CODE TIME TECHNOLOGIES

Abassi RTOS

User's Guide

Table of Contents

List of Figures

List of Tables

1 Introduction

This document is the User's Guide for the Abassi RTOS, and provides all the information the reader needs on how to configure and use the RTOS. It was written to deliver to the user all the information related to the Abassi RTOS, without turning into a tutorial on operating systems. The reader is assumed to have a good understanding of embedded and real-time concepts, general knowledge of RTOS, and a desire to understand the design process and design decisions that led to the Abassi RTOS.

While this document contains a lot of information on the internal operation of the Abassi RTOS, there is nothing in the way of an introduction to RTOS or OS concepts. Plenty of books and tutorials are available that discuss OS and RTOS. If the nomenclature used in this document differs from any reference work, a quick glance at the glossary below should be sufficient to remove these ambiguities.

1.1 Glossary

Adam & Eve	"Adam & Eve" is the name given in Abassi for the task associated to the "C" function `main()`. This name was chosen since this is always the first task created in an application, and from this task one or more tasks can be created.
Blocking	Blocking is a mechanism to stop a task when it needs access to an unavailable service. When a task gets blocked, it does not use any CPU and relinquishes the CPU to another task.
Component	An access point to a RTOS service.
Descriptor	A service descriptor is a single instance of class of operation. In Abassi, a descriptor is always referenced by the pointer to the data memory holding the information.
Event	Events are a synchronization mechanism holding multiple flags, where the flags can only be set. Tests are applied to the flags to validate the synchronization when a set of conditions is fulfilled.
Mailbox	First-in First-out data structure used to exchange information between two entities. A mailbox uses fixed size element exchanges, when a message queue uses variable size elements.
Mutex	(MUTual EXclusion) service, providing a mechanism to avoid simultaneous use of a shared resource.
Ready to run	State of a task where the task is ready to run but cannot use the CPU because another task is using it.
Resuming	Operation performed on a task to terminate the suspended state. When a task is in the suspended state, it does not use the CPU, nor can it react to synchronization services.
RTOS	Real-Time Operating System.
Running	Operation state of a task where it is the task using the CPU.
Semaphore	Service providing a synchronization mechanism.
Service	This is a class of operations based on one synchronization/exchange mechanism.
Stack	Last-in First-out data structure commonly used in processors. It is typically used to hold function call/return and local variable in re-entrant systems. In a RTOS, each task has its own stack that is used by the processor when the task is in the running state.
Suspending	Forcing a task to go into the suspended state where it will neither request nor use the CPU, and will not react to synchronization services.
Timer	RTOS service offering time based synchronization triggers.

2 Features

- Portable source code with minimal amount of assembly code
- Fully scalable: only the services required are compiled in
- The code can reside in ROM
- Can be re-started without reloading the code/data image
- Very small code size
- Very low interrupt latency
- Data used by the kernel services can be allocated dynamically, or statically, or at compile time
- Fully preemptive
- Cooperative mode emulation
- Versatile semaphores, features are selectable on per semaphore basis:
    - Counting semaphores or binary semaphores
    - Priority ordered or first come first served ordered
- Reentrant Mutexes, features are selectable on per Mutex basis:
    - Priority ordered or first come first served ordered
    - Task suspension postponed upon Mutex lock
- Recursive Mutex deadlock detection
- Mutex unlocking can be restricted to the locker only
- Priority Inheritance on Mutexes to eliminate the priority inversion issue
- Intelligent Priority Ceiling on Mutexes to eliminate the priority inversion issue
- Mailboxes
- Events with AND & OR masks
- Grouping (semaphores and mailboxes)
- Run-time safe service creation
- Real-time tailored Priority Aging protection
    - Programmable highest priority on per task basis
    - Programmable maximum run time at a raised priority on per task basis
    - Programmable maximum wait time on per task basis
- Dynamic priority changes
- Can be configured for one task per priority to reduce the code size
- Multiple tasks at same priority:
    - On a first come first served basis
    - Round Robin
    - Programmable Round Robin time slice duration on per task basis
    - Co-existence of round robin and run to blocking/completion at same priority
- No limits on the number of Tasks / Priorities / Semaphores, etc. …

- Selectable search algorithm for the next task to run
- Optional per task arguments
- Time-out on components
- RTOS timer callback
- Timer Services (one shot & periodic)
  - Data write to memory (with data post-addition)
  - Function call (with argument post-addition)
  - Semaphore posting
  - Mutex unlocking
  - Mailbox writing (with data post-addition)
  - Event flag setting
- Dynamic memory block management Services
- Names can be associated to the resources (Tasks, Semaphores, Mailboxes, etc.…)
- Interrupts are not disabled by the kernel (except with nested interrupts: only a few instructions)
- All RTOS components are available in interrupt contexts
- Simple Interrupt handler attachment
- Hybrid stack for interrupts (most ports)
- Fast Interrupt (FIQ) support (most ports)
- Optional stack usage monitoring
- Optional out of memory monitoring
- Operational log for debugging
- Multi-threading & reentrance library protection (when supported by the library)

2.1 Limitations

There are a few features not available in the Abassi RTOS.

- Queues are not supported because too many flavors of them exist. Most of the time an application needs a finely tuned queue system to be code size and real-time efficient. Using the mailboxes service that is available in the Abassi RTOS, it is quite easy to tailor a queue management service fully adapted to the needs of an application. An example is given in Section 6.7.13.
- Once a service descriptor (Task, Semaphore, Mutex, Mailbox, etc.) has been created, it cannot be deleted. Not allowing the deletion of services eliminates unexpected resource locking, it reduces both the code complexity and size, and it eliminates possible long-term fragmentation issues when dynamic memory is used.
- Operating the Memory Management Unit (MMU) on processors with such peripherals is not supported. The reasoning behind not supporting MMUs is to keep the Abassi RTOS simple and real-time efficient. This RTOS has been tailored for embedded applications, not for workstations or servers; it is not a RTOS capable of replacing the class of OSes like the real-time LINUX operating system.

3 Overview

This section gives an overview of the Abassi RTOS. The design choices are explained so as to give the reader an understanding of the decisions made when Abassi was architected and implemented.

3.1 Design choices

The Abassi RTOS was architected and implemented with 5 main goals:

- To be fully portable with very little assembly language programming
- To be the most feature-rich and innovative RTOS available
- To be completely scalable
- To use as little code and data memory as possible
- To not disable interrupts

The following subsections describe some of the techniques used to achieve each one of these goals.

3.1.1 Portability

Portability is guaranteed by coding most of the Abassi RTOS in standard ANSI-C language (the 1999 standard is used [R1]). A very small number of compilers used in the ports are still not C99 standard compliant, more than a decade after the establishment of the standard. When some aspect of the ANSI-C language syntax is not supported, an alternate method is used. One example of these standard syntaxes not being handled is the initialization of a data structure using the "`.field=value`" expression. It was decided to not use this form of initialization. (There is no side effect other than making the code a bit less readable.)

On purpose, the data types used in the implementation of Abassi are `char`, `int`, and pointers, and data structures composed of any of the previous. Almost all variable used in the RTOS are of type `int`; `char` is only used for dealing with memory allocation and for character strings. When a variable can hold both an `int` and a pointer, the standard data type `intptr_t`, as defined in `stdint.h`, is used. The choice of using `int` everywhere instead of `int8_t`, `int16_t` or `int32_t` (which are also defined in `stdint.h`) was made on the basis that, for most processor/compiler pairs, the data type `int` is the native data size of the processor. Being the native data size, data movements and operations are always performed in a single instruction, and this helps fulfilling both the goal of small code size and fast operation.

3.1.2 Feature set

The Abassi RTOS was designed to be feature-rich and innovative. As such, it supports almost all the features found in other RTOS, and adds many key features that are unique to it:

- Priority Aging
- Per task programmable round robin time slice
- Co-existence of round robin and run until blocking/completion at the same priority
- Mutex deadlock detection
- Intelligent priority ceiling
- Task suspension postponed upon Mutex lock

3.1.3 Scalability

Scalability was achieved with the use of C pre-processor `#define` directives. This allows the Abassi RTOS to scale from handling a single task per priority with nothing else, which obviously delivers the smallest data size requirement and the smallest and fastest code, to the other extreme, where all available features are enabled. Due to the large number of combinations of build options, it is not realistic to build or supply individual libraries for every combination of options. Instead, the source code of the RTOS is supplied to the user, which allows full control of tuning the build options to fulfill the exact requirements of the application.

3.1.4 Code Size

Keeping the code size as small as possible offers multiple advantages. This is especially true when using ASICs, which often impose additional memory (both data and code) constraints. Additionally, the smaller is a piece of code, the better the chances of high performance on processors using a memory cache. Finally, depending on how the small code size was achieved, it translates into a lower number of instructions, therefore delivering better real time performance.

The technique used in the Abassi RTOS to keep the code size small was to eliminate all but one function call inside the kernel. One has to remember that calling a function typically implies pushing the argument(s) of the function (or setting the argument(s) in specific registers) and a return address on the stack (or register), plus the called function may need to create its own local context. Even worse, a function call always disrupts the optimizer because (on most compiler implementations) around half the registers are lost at the calling function level, as the called function is allowed to modify them. By eliminating function calls, it allows the optimizer (when capable of recognizing the condition) to use all the registers of the processor for most of the kernel code. Eliminating the extra operations required for a function call, and possibly using all the registers, translates most of the time into smaller, and also faster, code.

The internal RTOS functionality was decomposed into elementary operations, e.g. "remove a task from the blocked list", "add a task to a running list", etc. These elementary operations would typically be implemented by individual functions, which would then be used in higher level functions like "posting a semaphore". Instead, they are all directly implemented in the kernel without function calls, and it is done in a manner that code duplication is largely avoided. In the end, the Abassi kernel is single large function with conditional processing blocks. The processing blocks were grouped in a sort of successive approximation, and were ordered in such a way as to minimize the number of conditions to check for any operation by the kernel.

The sole function call in the kernel is used to perform the context switch, which must be coded in assembly language. Many "C" compilers allow direct insertion of assembler directives in the "C" code. However, the presence of assembly code in the "C" code can confound some optimizers. Even for compilers / optimizers that deal properly with the assembly code needed for the context switch, it was decided to keep this operation in an assembly file because some compilers have extra build configurations only set for the assembly file.

3.1.5 Data Size

Care has been taken to optimize the descriptors of the tasks, semaphores, mailboxes, etc. When features of the Abassi RTOS are disabled through the build options, their respective entries are eliminated from the descriptor, reducing the data memory requirements for each descriptor and the kernel operations. Significant time and effort was spent when defining the Abassi architecture to find ways to re-use the fields or variables for multiple purposes.

3.1.6 Interrupts not disabled

Any RTOS kernel possesses critical regions that need to be protected. The obvious way to protect these critical regions is to disable interrupts when these critical regions run. However, there are disadvantages to doing so. The first is that on some processor the enabling and disabling is not as straight-forward as simply setting/clearing a bit in the data memory space (memory mapped register), but instead involves inserting assembly code in the middle of "C" code, which sometimes confuses the optimizer. Doing so would then require an assembly-coded function, and the associated overhead of a function call. The second is that the re-enabling of the interrupts after the critical region must be conditional on the interrupts having been enabled in the first place. The third issue is for applications with very strict real-time requirements, and depends on the size of the critical region: when interrupts are disabled and an interrupt occurs, the interrupt processing is delayed, which could have a negative impact.

3.2 Services

This section gives an introduction to all the basic services supported by the Abassi RTOS.

The Abassi RTOS supports all the services that are required in a full featured RTOS:

- ➢ Tasks
- ➢ Semaphores
- ➢ Mutexes
- ➢ Event flags
- ➢ Mailboxes
- ➢ Timer
- ➢ Memory Block Management
- ➢ Interrupts

Many of these services use a descriptor (object) that needs to be created and initialized. In some RTOS, there is a never-ending case of ambiguity between either the use of the descriptor or the use of a pointer to the descriptor. The Abassi RTOS supports and makes visible pointers to the descriptors and nothing else; this is even true when a descriptor is statically created at compile time. This means pointers are the only descriptor type available and usable by the user, so there is no confusion on the data types. Every time the expression "descriptor" is used in this document it really means the pointer to the descriptor, as pointers to descriptors are the only access method.

3.2.1 Tasks

Tasks in the Abassi RTOS are what are commonly called "a thread" in the literature. A task is the elementary processing entity in the Abassi RTOS; it possesses its own stack and operates exactly as if it was it was an independent little processor having access to all memory and peripherals inside the full processor. The Abassi RTOS has no limits on how many tasks can exist in an application, nor is there a restriction on the task code size, stack size, or data size usage; only the data and code space available to the processor limits these numbers.

Tasks are always in one out of 4 possible states:

- Running
- Ready to Run
- Blocked
- Suspended

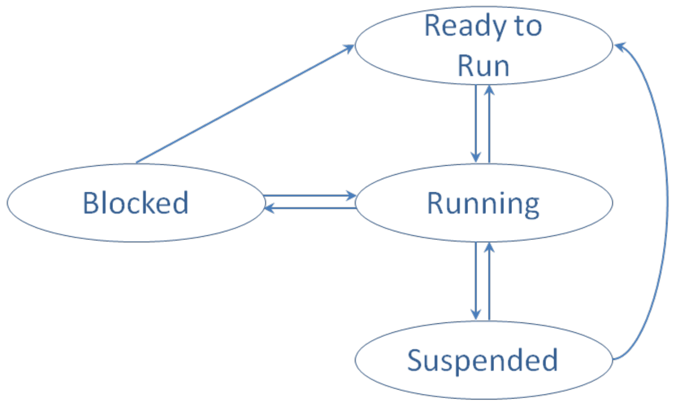

Figure 3-1 State Changes

Each task is assigned a priority. A priority value of 0 is the highest priority, and the larger the numerical value of the priority, the lower is the priority level (the maximum priority value is the maximum numerical value supported by a `signed int`, so typically 32767 for 16 bit integer or 2147483647 for 32 bit integer). The task with the highest priority (lowest numerical value) that is not blocked, suspended, or preempted by another task at the same priority, is the task that uses the CPU, and it is the task in the running state; at all time, there is one and only one task in the running state. All other tasks that are not blocked or suspended are in the ready to run state.

A blocked task is a task waiting for a service; this can be either a semaphore to be posted, a mutex to be unlocked, event flags to be set, room to deposit a message in a mailbox, or the availability of a message to be retrieved from a mailbox.

Multiple tasks can co-exist at the same priority, and depending on the build option, they either share the CPU time available through round robin, which is also called time slicing, or they can simply operate on a first come first served basis, or a mix of both. A task can be suspended only if it is in the running or ready to run state. When a task is blocked, it needs to get unblocked before the suspension occurs.

3.2.2 Semaphores

Semaphores are at the heart of the Abassi RTOS and are the only internal blocking mechanism. Every synchronization service uses a semaphore to block a task. Mutexes are simply semaphores in disguise. Events are implemented as a set of conditions attached to the private semaphore of a task that specify when to acquire or when to release the private semaphore. It is the same with mailboxes. And even suspending a task is simply blocking the task on its own private semaphore (this is the same semaphore used by the event flags, re-used to save on data memory usage).

All semaphores in the Abassi RTOS are counting semaphores: they keep track of the excess number of postings. Binary semaphores are also available; the accumulated count is zeroed upon the acquisition of the semaphore, so a regular counting semaphore is used for a binary semaphore, and it becomes a binary semaphore upon being acquiring.

By default, tasks that are blocked on a semaphore will become unblocked in a priority based ordering: the highest priority task that is blocked on a semaphore is the first to be unblocked on the first posting, no matter when that highest task got blocked. If the same semaphore blocks multiple tasks with the same priority, the task that was the first to try to acquire the semaphore will become the first to get unblocked.

The Abassi RTOS optionally supports the configuration of semaphores to make them operate in a *First Come First Served* mode. When configured in this mode, the priority of the tasks blocked on the semaphore is not taken into account when determining which task gets unblocked first. The first task to get unblocked when a semaphore is operating in *First Come First Served* mode is the first task that tried to acquire the semaphore, no matter what its priority is. The selection of *Priority* mode or *First Come First Served* mode is done on a per semaphore basis (and, implicitly, per mutex and per mailbox).

3.2.3 Mutexes

In the previous section it was stated that mutexes are semaphores in disguise. A mutex is simply a binary semaphore with an initial count of the accumulated postings of 1 instead of 0, as in a regular semaphore. Mutexes are a synchronization mechanism used to control access to shared resources, making sure only one task can access the resource at any time. When a task has acquired a mutex, this means it has access to the shared resource, and all other tasks that try to acquire the mutex will be blocked until the task that has access to the resource (the owner of the mutex) releases (unlocks) the mutex.

Another characteristic of the Abassi mutexes is that they are fully re-entrant mutexes. A re-entrant mutex is a mutex that can be locked multiple times by the owner (the same number of unlockings must be performed to release the lock on the mutex). This characteristic allows the use of a single mutex for multiple shared resources that themselves also use a common resource. The multiple locks on a mutex translate into multiple acquisitions, which is not possible on a semaphore. So, for the mutexes, the count value of the semaphore is allowed to become negative in order to track the number of recursive locks performed by the owner of the mutex.

Mutexes are the root cause of the so-called *Priority Inversion* problem in multi-tasking applications. The Abassi RTOS optionally supports either automatic *Priority Inheritance* or *Priority Ceiling* to eliminate the *Priority Inversion* problem. Abassi Priority Ceiling is completely automatic; there is no need to set the mutex ceiling priority, the priority is determined at run time. More details on Priority Inheritance / Ceiling are given in Section 7.

Another problem that can exist with mutexes is a mutex deadlock. Abassi optionally support the run-time discovery of mutex deadlock, and when a deadlock is detected, the lock request is cancelled and an error condition is reported to the caller. See Section 9 for more information on mutex deadlock detection.

3.2.4 Event Flags

When the event flags service is enabled through the build options, it gives every task an event register set that can be used to synchronize the task through event flags. Event flags can be set by any tasks in the application, and typically, a task will be blocked until the desired combination of flags in its register is true.

The flag condition is a minimalist "sum of products". Two masks define the condition, an AND mask and an OR mask. The AND mask condition is true when all flags matching the bits set to 1 in the AND mask have been set in the event register. The OR mask condition is true when any of the flags matching the bits set to 1 in the OR mask have been set in the event register. When one of the two mask conditions is true, the event is declared valid. When an event is valid, the task owning the event register is unblocked if it was blocked on the event. The number of flags held in the event register is determined by the `int` size used by the compiler; this is typically 16 or 32 bits.

3.2.5 Mailboxes

Mailboxes are queues (first-in first-out buffers) with fixed size elements, and this service is optionally supported by the RTOS through the build options. There should always be a single reader of the mailbox, as in the real life postal system, but any task can write to the mailbox. The Abassi RTOS does not enforce the single reader requirement and can properly operate with multiple readers (except when the reader is an interrupt handler). When the mailbox is empty, the reader(s) can be blocked until a message is deposited into the mailbox. When the mailbox is full, the writer(s) can be blocked until there is room to put a new message in the mailbox.

3.2.6 Timer

Time based operations are supported by the Abassi RTOS when enabled through the appropriate build options. The time-based operations cover round robin, timeout on a service, and blocking a task for a fixed duration. All blocking services have the capability to block a task forever (until the normal unblocking occurs), to not block the task at all, or to block the task for a maximum time, called the expiry time. The timer resource used in the Abassi RTOS is not hour - minute - second based, but tick based, which is another way to indicate it is based on a simple ever incrementing counter (with roll-over). If needed, components are available to convert timer tick units into/from milliseconds or seconds, abstracting the internals of the timer from the application.

Also involving the timer is an optional timer service module, which gives the application access to generic timer facilities. The timer services can be used to perform a delayed operation or periodic operations. With selected timer services, when the operation is periodic, it is possible to add an offset to the argument of the operation. For example, a function can be periodically called, and after each periodic call, the function argument gets its value updated by a specified value.

3.2.7 Memory Block Management

Abassi supplies a memory block management module, allowing an application to create as many memory block pools as required. Each memory block pool holds a number of memory blocks, all with the same number of bytes per block, selectable on a per pool basis. The memory block management service offers simple "alloc" and "free" type components, and tasks can get blocked upon exhaustion of the memory block pool.

NOTE: Because the memory block management service can be used inside an interrupt, it is not possible to support this service without disabling / enabling the interrupts in the kernel. The interrupt enabling / disabling only affects the memory block management section in the kernel; even when the memory block management service is part of Abassi, none of the other services disable interrupts.

3.2.8 Interrupts Handlers

Interrupt handlers are easily attached to an application using the Abassi RTOS. This is done with the `OSisrIntall()` (Section 6.8.6) component, where all there is to do is to indicate the interrupt number and the function to use as the interrupt handler. From a user point of view, an interrupt function is not much different from any other function; all resources of the Abassi RTOS are available as is. There is no need to declare the function with non-standard keywords like `interrupt` or `using`.

There are a few exceptions, obviously, such as not having an interrupt handler waiting on a semaphore or trying to lock a mutex for example; the use of these exceptions in an application is not verified by the RTOS, and it is the responsible of the designer to not use any of these exceptions.

3.3 Constraints and Don'ts

There are very few limitations in the Abassi RTOS. The main one relates to the need to always have a task in the running state in the application. The other constrains are either related to the build choices, or the internal architecture, or obvious caveats applicable to any RTOS.

3.3.1 Idle Task

The Abassi RTOS requires there to always be a task in the running state in the application, which is the origin of the concept of the Idle Task. The Idle Task (or an application equivalent) is always the lowest priority task in the application, and this task cannot be in the blocked or suspended state. This means none of the following components can be used in the Idle Task if the expiry timeout specified in the arguments is non-zero:

- `SEMwait()`
- `SEMwaitBin()`
- `MTXlock()`
- `EVTwait()`
- `MBXget()`
- `MBXput()`
- `GRPwait()`

And the following components should never be used:

- `TSKsleep()`
- `TSKselfSusp()`
- `TSKsuspend()` (when the task to suspend is the Idle Task)
- `MTXlock()` (if priority inheritance or priority ceiling is enabled)

The Idle Task can be used as a regular task, performing any type of operations needed in the application, as long as it always remains either in the ready to run or running state. The Idle Task is also ideal to put a processor into idle mode or power saving mode (even though the Idle Task is in the running state, this does not mean it needs to actively be consuming CPU cycles). It is not necessary to create and use the Idle Task, but it is necessary to always have a running task in the application.

3.3.2 Interrupts

The constraints on components used in an interrupt context are not unique to this RTOS: no RTOS component that could block a task can be used in an interrupt handler. If such components are used, the most likely outcome is the running task that has been interrupted will become blocked. The components that are not authorized to use a timeout are the same as the one listed in the previous section. These components can still be used in an interrupt as long as the timeout value is set to zero. It was decided to not add code to override a non-zero value, in order to not create a new type of problem that the user would not be aware. Others components don't make sense to use in an interrupt.

The following table lists the components that can safely and meaningfully be used in an interrupt, as long as the timeout value is zero:

Table 3-1 Components usable in an interrupt

Component	Purpose / Extra Information
`SEMwait`	Decrement a semaphore count / Timeout value must be 0
`SEMwaitBin`	Flush the accumulated count of a semaphore / Timeout value must be 0
`MBXput`	Deposit a message in a mailbox / Delayed operation / Timeout value is ignored
`MBXget`	Retrieve a message from a mailbox / Timeout value ignored
`MTXunlock`	Unlock a mutex
`MBLKalloc()`	Retrieve a block of memory from a memory block pool

The second to last item, `MTXunlock()` needs a bit of explanation on how to correctly use it in an interrupt. This component should not be used blindly in an interrupt handler. But, with careful planning, using it in an interrupt handler can speed-up the reaction time of the application.

An example of the proper use of unlocking a mutex in an interrupt would be the case when a shared peripheral needs to be protected by a mutex. If the peripheral generates an interrupt after completion of its operations, then it makes sense to unlock the mutex directly in the interrupt handler. Performing the unlocking operation in the interrupt means that no task accessing the peripheral can unlock the mutex; all they can do is to lock the mutex.

The implementation described in the above example speeds-up the reaction time because it eliminates the extra step of having the mutex locker to run and then unlock the mutex, which would then unblock the next task trying to lock the mutex.

BEWARE: If one understands sufficiently well the problem of mutex deadlock, the above example is a perfect candidate to create a mutex deadlock, under certain conditions.

Kernel requests performed during an interrupt are queued for processing outside the interrupt context (except for the `MBXget()` component, Section 6.7.5). This means that any kernel request performed during an interrupt will be processed a bit later. As a side effect, the return value or result of using such a component is always zero (Success) as the kernel request is always successfully queued. The operation is still performed, but with a small delay. The return value always being zero was changed in Abassi V1.262.234, mAbassi V1.76.75, and uAbassi V1.34.24. The return value in these versions and following indicate if the queuing has been successful or not.

The component `MBXget()` (and `MBXput()` in Abassi V1.262.234 and mAbassi V1.76.75, and following) is/are an exception to the above, as special code is used when the request is performed in an interrupt handler. The addition of this special code was deemed necessary. For example, reading a mailbox in an interrupt may be required when the interrupt is triggered by a peripheral that needs to be fed new data, assuming the new data is held in a mailbox.

The added code creates a new constraint, which is typically not an issue: two or more nested interrupt handlers cannot read the same mailbox. This means that if multiple interrupt handlers at the same priority read the same mailbox, then it is safe. What is not safe is if two or more interrupt handlers at different priorities read the same mailbox; then there is a risk of having mailbox data duplication or loss. As previously stated, a mailbox service should normally always have a single reader.

The following table lists the components that should never be used in an interrupt, as they apply to and affect the running task. These are not restrictions unique to the Abassi RTOS; these operations simply do not make sense in an interrupt.

Table 3-2 Components to never use in an interrupt (1)

Component	Issue
`MTXlock`	If the mutex is already locked, a pairing `MTXlock` / `MTXunlock` done in an interrupt will not lock the mutex as the timeout is 0, but it will unlock the mutex, not making the locker aware of the loss of lock.
`EVTwait`	If the flag conditions are valid, the flags will be moved into the receive register without the task owning the event flags being aware of the fact. This is almost equivalent to losing flags setting.
`TSKsleep`	Will put the currently running task in sleep.
`TSKselfSusp`	Will suspend the currently running task.
`GRPwait`	Cannot operate inside an interrupt. It will also report failure if used in an interrupt, no matter what are the value of the `Timeout` and `All` arguments

The following table lists another group of components that should never be used in an interrupt, as they deal with resource creation / allocation. These components manipulate one or more of Abassi's global resources as they rely on a mutex (locking with infinite timeout) to give the calling task exclusive access to the resource. As a mutex locking / unlocking is performed in these components, none of them can be used in an interrupt.

Table 3-3 Components to never use in an interrupt (2)

Component	Issue
`TSKcreate`	`MTXlock` issue described in Table 3-2.
`SEMopen`	`MTXlock` issue described in Table 3-2.
`SEMopenFCFS`	`MTXlock` issue described in Table 3-2.
`MTXopen`	`MTXlock` issue described in Table 3-2.
`MTXopenFCFS`	`MTXlock` issue described in Table 3-2.
`MBXopen`	`MTXlock` issue described in Table 3-2.
`MBXopenFCFS`	`MTXlock` issue described in Table 3-2.
`TIMopen`	`MTXlock` issue described in Table 3-2.
`MBLKopen`	`MTXlock` issue described in Table 3-2.
`MBLKopenFCFS`	`MTXlock` issue described in Table 3-2.
`GRPaddMBX`	`MTXlock` issue described in Table 3-2.
`GRPaddSEM`	`MTXlock` issue described in Table 3-2.
`GRPaddSEMbin`	`MTXlock` issue described in Table 3-2.
`GRPrmAll`	`MTXlock` issue described in Table 3-2.
`GRPrmMBX`	`MTXlock` issue described in Table 3-2.
`GRPrmSEM`	`MTXlock` issue described in Table 3-2.

3.3.3 Task Suspension

There is a generic component named `TSKsuspend()` (Section 6.3.29) used to suspend a task. If a task to suspend is blocked, the suspension only occurs when the task becomes ready to run. Additionally, a critical safety feature is added in the Abassi RTOS: a task will not go into the suspended state until it has unlocked all the mutexes it owns.

The verification that the task to suspend does not own mutexes is not performed when the component `TSKselfSusp()` (Section 6.3.18) is used. Proper care must be taken when using the component `TSKselfSusp()` otherwise some tasks in the application may remain blocked indefinitely if they try to acquire a lock on a mutex still locked by a self-suspended task.

3.3.4 Single Task per Priority

There is an issue to be mindful of when the Abassi RTOS is configured for a single task per priority: no two tasks can have the same priority. This is straightforward, but if the priority inversion protection feature is enabled in the Abassi RTOS build, then extreme care must be taken when assigning the priority to the tasks in an application. For more information consult section 7, on priority inheritance / priority ceiling.

If two or more tasks operate at the same priority in a build set for a single task per priority, all tasks at that priority, except one, will "disappear" from the application. This is not equivalent to suspending a task: the tasks are in the ready to run state but will not run. They are still there, but can't become blocked, running or suspended.

3.4 Distribution Contents

The Abassi RTOS source code distribution always has a minimum of 3 files:

`Abassi.h`	The Abassi RTOS definition file
`Abassi.c`	The Abassi RTOS code
`Abassi_???_???.?`	The processor / compiler specific assembly file

Most of the distributions have code examples for specific hardware platforms, and some processor/compiler ports may also include device drivers. Consult the processor/compiler port document that applies to your target application.

3.5 C++

Abassi can be used in a C++ environment. As Abassi is entirely coded in "C", and not "C++", care must be taken when attaching functions to the Abassi services (this is the only restriction involved when using Abassi in C++). The following functions must be declared with "C" linkage in a C++ environment:

- The function attached to a task when using `TSK_STATIC()` (Section 6.3.2)
- The function attached to a task when using `TSKcreate()` (Section 6.3.5)
- The handler attached to an interrupt through `OSisrInstall()` (Section 6.8.6)
- The function attached to a timer service when using `TIMfct()` (Section 6.9.5)

The RTOS timer callback function `TIMcallBack()` (Section 6.7.16) is already declared with "C" linkage; therefore nothing special is required when creating the callback function, as long as the file `Abassi.h` is included in the file where the function `TIMcallBack` is located; if not, the function `TIMcallBack()` must be declared with "C" linkage.

4 Configuration

The Abassi RTOS is fully configurable: services and modules that are not required/used are not compiled-in and the respective entries in the task, semaphore, mutex or mailbox descriptors are not present. The inclusion of a feature is controlled by the use of `#define` in "C"; the token used in these `#define` are called "build options". There are two ways to specify these `#define`: directly in the file `Abassi.h`, or with typically the option `–D` or `-d` on the compiler command line, most of the time as part of a make file. The first method, which defines the build options in the file `Abassi.h` is desirable when a single instance of the Abassi RTOS is built (or multiple instances of the RTOS are built with exactly the same configuration). The second method is preferable when building multiple platforms that require different configurations of the Abassi RTOS. The latter case would happen on a multi-processor platform where each processor uses a different configuration of the Abassi RTOS. Moving the definition out of the `Abassi.h` file into the make file allows the build process to keep a single instance of the `Abassi.h` file.

To use the definitions in `Abassi.h`, all there is to do is to not define the build option `OS_DEF_IN_MAKE` (Section 4.1.2) on the compiler command line. This will force the build process to use all the definitions in `Abassi.h`. If the build option `OS_DEF_IN_MAKE` is defined (the definition value is not important) on the compiler command line (most likely in a make file), all the build options that are defined in the file `Abassi.h` are completely ignored, but the make file must define all of them on the compiler command line. If some build options haven't been defined, building the Abassi RTOS will report at compile time which build options have not been defined. When build options have invalid values or there are conflicts between related build options, error messages are generated during the build operation.

There are some other internal build options not really accessible to the user. These internal build options are processor / compiler specific and are set to values that match the processor / compiler capabilities when applicable, and/or their values are set to generate code that minimizes both the real-time CPU usage and/or the code and data memory usage.

NOTE: Each of the build options (except `OS_DEF_IN_MAKE`, Section 4.1.2) described in the sub-sections of 4.1 must be defined. This is true even if a build option is internally overloaded because of the value of another build options. It has been decided for enforce this because, first, it is easier to work with a standard template of definitions. And second, because when a build option is modified, if this build option was provoking an overload of another option, error messages will not look like suddenly appearing from nowhere because build options that are not overloaded anymore are not defined.

NOTE: MISRA-C:2004 compliance makes the use of `#undef` not acceptable. As some build options are internally overloaded, it was necessary to internally use a different build options when overloading occurred. To simplify the identification of the build options, all internal build options that are used have the `OS_` part of the token name replaced by `OX_`. This means it is strongly suggested when a build option is used in the application code that the token name `OX_` should be used instead of the one with `OS_`.

4.1 Build Options

The following sub-sections describe each of the build options that must be defined, and the meaning of their value. The next section, Section 4.2, gives a step by step explanation to assist the reader in setting the build option values according to the needs of their application.

4.1.1 OS_ALLOC_SIZE

Depending on the setting of many build options, dynamic memory allocation may be the source of the data memory utilized when creating services. When dynamic memory allocation is used and this build option is set to a value of zero, the standard "C" library function `malloc()` is the memory allocator.

When the build option `OS_ALLOC_SIZE` is positive, then `OSalloc()` (Section 6.14.3) uses special code to extract memory from an area of size `OS_ALLOC_SIZE` bytes that has been reserved at compile / link time. As memory allocated from that pool of memory is never returned to the pool, this method delivers a more efficient use of the data memory than `malloc()` does with the heap memory. An important add-on is that when this build option is positive, the memory allocator is protected by a mutex, making Abassi's internal allocator multithread-safe; this is not always the case when `malloc()` is directly used, which is when this build option is set to zero. (Some compilers have access to multithread-safe libraries when others don't.)

This build option cannot be set to a negative value; if it is, an error message will be generated at compile time.

4.1.2 OS_DEF_IN_MAKE

`OS_DEF_IN_MAKE` is not really a build option that configures the features of the Abassi RTOS; it is a token with the purpose of overriding all the build options defined inside the file `Abassi.h`. This token should only be defined on the compiler command line. Defining it, no matter what value is assigned to it, is an indication to ignore all the build options that are declared in the file `Abassi.h`. Overloading the definitions in `Abassi.h` is useful when applications targeted to multiple processors with different RTOS configurations are built from a common source code.

4.1.3 OS_CHECK_DESC

The build option `OS_CHECK_DESC` has been added in 2019 releases. When defined and set to a non-zero value it makes Abassi to check the validity of descriptors it is provided. This is done through the insertion of a marker in the descriptors that uniquely identify each type of descriptors. When an invalid or worng descriptor type is detected Abassi goes into the `OStrap()` facility (Sect 6.14.6).

4.1.4 OS_COOPERATIVE

The build option `OS_COOPERATIVE` make the Abassi RTOS kernel operate in a cooperative mode instead of preemptive. Setting this build option to a non-zero value configures the kernel to operate in the cooperative mode. For more information on Abassi's cooperative mode, refer to Section 11.

When `OS_COOPERATIVE` is set to a non-zero value, the build option `OS_ROUND_ROBIN` (Section 4.1.36) is internally forced to a zero value, as round robin cannot exist in a cooperative RTOS. Also, when `OS_COOPERATIVE` is set to a non-zero value, the build option `OS_SEARCH_ALGO` (Section 4.1.38) is internally forced to a negative value, so the kernel uses a 2-dimensional linked list to determine the next task that will run when a task relinquishes the CPU.

4.1.5 OS_EVENTS

The Abassi RTOS optionally offers the event flags service as one of the inter-task synchronization mechanisms. Setting this build option to a non-zero value configures the build to include the code that supports this service. Events are described in detail in Section 6.6.

4.1.6 OS_FCFS

Setting a non-zero value for the build option `OS_FCFS` adds the capability to make semaphores (and all other blocking services) unblock tasks on a *First Come First Served* ordering instead of default *Priority* ordering. This does not require all services to operate in *First Come First Served* mode; the type of ordering is specified when creating the service and it can also be notified during run-time on a per case basis. As previously explained, since the only blocking mechanism in the Abassi RTOS are semaphores, then mutexes and mailboxes also gain the capability of unblocking tasks in a *First Come First Served* ordering when this build option is set to a non-zero value.

4.1.7 OS_GROUP

The build option `OS_GROUP` is not required to be defined. When it is defined and set to a non-zero value, it adds support for groups of triggers (Section 6.11). Grouping triggers allows a task to block on reading one or more mailboxes and / or waiting for one or more semaphores. When defined and set to a positive value, it informs the Abassi RTOS to reserve memory at compile / link time. This memory is used to hold special descriptors. A positive value assigned to `OS_GROUP` specifies the total number of trigger descriptors available. When a negative value is assigned to `OS_GROUP`, then the memory needed to hold the trigger descriptors is obtained during run-time through the `OSalloc()` service (Sections 4.1.1 and 6.14.3). Setting the build option `OS_GROUP` to a value of zero disables the support of group of triggers. Note that when `OS_GROUP` is positive, it specifies the maximum total number of triggers, not the maximum total number of groups.

Availability:

Abassi:	Version 1.250.223 and up
mAbassi:	Version 1.60.59 and up
µAbassi:	Unsupported

4.1.8 OS_GRP_XTRA_FIELD

The build option `OS_GRP_XTRA_FIELD` is new in Abassi version 1.264.239 and mAbassi version 1.82.80. When defined and set to a positive value, it informs Abassi to add in the group descriptors `OS_GRP_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the group descriptor (of type `GRP_t`). This build option is ignored if the build option `OS_GROUP` (Section 4.1.7) is not defined or if defined and set to 0. The application usable entries in `XtraData[]` are the first `OS_GRP_XTRA_FIELD` entries; if Abassi internally needs to use entries in `XtraData[]` it adds extra entries and accesses the entries at and above index `OS_GRP_XTRA_FIELD`.

4.1.9 OS_HASH_ALL

The build option `OS_HASH_ALL` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for all named services and task. The size of the hashing tables is the value defined by `OS_HASH_ALL` and it must be a power of 2 (unsigned). When `OS_HASH_ALL` is defined and set to an unsigned non-zero value, all other hashing table build options are ignored. This build option is ignored if the built options `OS_RUNTIME` (Section 4.1.37) or `OS_NAMES` (Section 4.1.28) are zero.

When runtime creation and names are supported (build options `OS_RUNTIME` and `OS_NAMES`), Abassi by default keeps the service and task names in a linked list, one linked list per service / tasks. If an application creates a large number of the same type of service, then the opening of the service or the extraction of the task by name could take a long time due to the need to traverse the linked list. When hashing tables are used, the service and task names are kept in multiple linked lists; the number of linked lists is the value assigned to `OS_HASH_xxx`. Statistically, the search time is `1/OS_HASH_xxx` the time required using a single linked list.

4.1.10 OS_HASH_MBLK

The build option `OS_HASH_MBLK` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named memory blocks. The size of the hashing tables is the value defined by `OS_HASH_MBLK` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28), or `OS_MEM_BLOCK` (Section 4.1.23) are zero. See section 13 for further information about the hashing tables.

4.1.11 OS_HASH_MBX

The build option `OS_HASH_MBX` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named mailboxes. The size of the hashing tables is the value defined by `OS_HASH_MBX` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28), or `OS_MAILBOX` (Section 4.1.18) are zero, or `OS_HASH_ALL` (Section 4.1.9) is non-zero. See section 13 for further information about the hashing tables.

4.1.12 OS_HASH_MUTEX

The build option `OS_HASH_MUTEX` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named mutexes. The size of the hashing tables is the value defined by `OS_HASH_MUTEX` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28) are zero, or `OS_HASH_ALL` (Section 4.1.9) is non-zero. See section 13 for further information about the hashing tables.

4.1.13 OS_HASH_SEMA

The build option `OS_HASH_SEMA` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named semaphores. The size of the hashing tables is the value defined by `OS_HASH_SEMA` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28) are zero, or `OS_HASH_ALL` (Section 4.1.9) is non-zero. See section 13 for further information about the hashing tables.

4.1.14 OS_HASH_TASK

The build option `OS_HASH_TASK` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named tasks. The size of the hashing tables is the value defined by `OS_HASH_TASKK` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28) are zero, or `OS_HASH_ALL` (Section 4.1.9) is non-zero. See section 13 for further information about the hashing tables.

4.1.15 OS_HASH_TIMSRV

The build option `OS_HASH_TIMSRV` is not required to be defined. When it is defined and set to an unsigned, non-zero value, it configures Abassi to use hashing tables for the named timer services. The size of the hashing tables is the value defined by `OS_HASH_TIMSRV` and it must be a power of 2 (unsigned). This build option is ignored if the build options `OS_RUNTIME` (Section 4.1.37), or `OS_NAMES` (Section 4.1.28), or `OS_TIMER_SRV` (Section 4.1.59) are zero, or `OS_HASH_ALL` (Section 4.1.9) is non-zero. . See section 13 for further information about the hashing tables.

4.1.16 OS_IDLE_STACK

This build option has a double meaning as it controls, first, if a dedicated Idle Task function is supplied by the application or not, and second, when the Idle Task function is supplied, what is the size of the stack to allocate to this task. When `OS_IDLE_STACK` is set to a positive value, the Abassi RTOS creates at start-up the environment for an Idle Task function supplied by the application and it allocates a stack size of `OS_IDLE_STACK` bytes to the task. If this build option is zero, the Idle Task environment is not created, but the application must create the equivalent of an Idle Task, and this task must have its priority set to the lowest level amongst all other tasks (see `OS_PRIO_MIN`, Section 4.1.34). This build option cannot be set to a negative value; if it is, an error message will be generated at compile time.

4.1.17 OS_LOGGING_TYPE

When set to a value of zero, no logging of the internal operations is performed by the Abassi RTOS.

When this build option is set to a numerical value of 1, the Abassi RTOS outputs ASCII messages for each one of the operations it performs. The output device is specified by the definition of `OSputchar()` (Section 6.14.5); this function uses the same function prototype as the standard "C" library function `putchar()`.

Note: Activating ASCII logging creates a significant impact on the real-time performance of the kernel. The real-time performance of the kernel will definitely degrade.

When this build option is set to a numerical value greater than 1, the Abassi RTOS inserts packets of numerical information into a circular buffer sized to the build option value. The sizing is the number of packets the buffer can hold.

The logging can be run-time started/stopped, and when the circular buffer is used, can be dumped over the same ASCII output mechanism as used for the ASCII dump (`OS_LOGGING_TYPE` set to a value of 1) or dumped anywhere else by using the logging buffer reader API.

When `OS_LOGGING_TYPE` is non-zero, the build option `OS_NAMES` (Section 4.1.28) is internally enabled (forced to a non-zero value), activating the naming of all services.

4.1.18 OS_MAILBOX

The Abassi RTOS optionally offers the possibility of using mailbox services as a data exchange and synchronization mechanism. Setting this build option to a non-zero value includes the code during the build to support this service. If `OS_MAILBOX` is set to zero, meaning the mailbox service is not supported, then the build options `OS_STATIC_BUF_MBX` (Section 4.1.45) and `OS_STATIC_MBX` (Section 4.1.47) must be set to a value of zero.

4.1.19 OS_MBX_XTRA_FIELD

The build option `OS_MBX_XTRA_FIELD` is new in Abassi version 1.264.239 and mAbassi version 1.82.80. When defined and set to a positive value, it informs Abassi to add in the mailbox descriptors `OS_MBX_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the mailbox descriptor (of type `MBX_t`). This build option is ignored if the build option `OS_MAILBOX` (Section 4.1.18) is set to 0. The application usable entries in `XtraData[]` are the first `OS_MBX_XTRA_FIELD` entries; if Abassi internally needs to use entries in `XtraData[]` it adds extra entries and accesses the entries at and above index `OS_MBX_XTRA_FIELD`.

4.1.20 OS_MBLK_XTRA_FIELD

The build option `OS_MBLK_XTRA_FIELD` is new in Abassi version 1.264.239 and mAbassi version 1.82.80. When defined and set to a non-zero value, it informs Abassi to add in the memory block descriptors `OS_MBLK_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the memory block descriptor (of type `MBLK_t`). This build option is ignored if the build option `OS_MEM_BLOCK` (Section 4.1.23) is not defined or if defined and set to 0. The application usable entries in `XtraData[]` are the first `OS_MBLK_XTRA_FIELD` entries; if Abassi internally needs to use entries in `XtraData[]` it adds extra entries and accesses the entries at and above index `OS_MBLK_XTRA_FIELD`.

4.1.21 OS_MAX_PEND_RQST

The value of this build option sets the dimension of an internal queue used to absorb all kernel requests performed during interrupts. One must be aware that bursts of continuous back-to-back interrupts can starve the kernel of CPU, to the point where it is not able to completely empty this internal queue between each interrupt. So, when selecting the value for this build option, one must take in account the worst case of all interrupts performing the maximum number of requests to the kernel at once; and due to the way the queue is implemented (chosen to minimize code size and to optimize real-time efficiency), there is always one entry that is never used. For example, if the worst total number of back-to-back requests is 10, `OS_MAX_PEND_RQST` must be set to 11 or higher.

If the value of this build option is set to an exact power of 2, code space and CPU cycle savings are achieved.

This build option cannot be set to a value of less than 2; if this happens, an error message will be generated at compile time.

For MISRSA-C:2004 compliance the suffix `U`, indicating an unsigned integer, must be appended to the numerical value specified.

4.1.22 OS_MBXPUT_ISR

The build option `OS_MBXPUT_ISR` was added in Abassi V1.262.234 and mAbassi V1.76.75 (and following). If the build option `OS_MBXPUT_ISR` is not defined, it is assume it has a value of zero. When defined and set to a non-zero value, it enables the possibility to get a valid return value when using `MBXput()` (Section 6.7.9) in an ISR. This means when `MBXput()` is called from an interrupt handler, if it returns a zero value, then the message will be added to the mailbox. If it return a non-zero zero, it means either the mailbox is full or the interrupt request queue is full, meaning the message will not land in the mailbox.

Please refer to the section on `MBXput()` (Section 6.7.9) because setting the build option `OS_MBXPUT_ISR` to a non-zero value does not enable the validation of the return value in an ISR; this feature must be enable on individual mailboxes.

4.1.23 OS_MEM_BLOCK

The Abassi RTOS optionally offers the possibility of managing pools of memory blocks as a data retrieval and synchronization mechanism. The build option `OS_MEM_BLOCK` is one of the few options that is not required to be defined. When defined and set to a non-zero value it includes the code during the build to support this service. If `OS_MEM_BLOCK` is set to zero, meaning the memory block management service is not supported, then the build options `OS_STATIC_BUF_MBLK` (Section 4.1.44) and `OS_STATIC_MBLK` (Section 4.1.46) must be set to a value of zero.

4.1.24 OS_MIN_STACK_USE

The build option `OS_MIN_STACK_USE` configures the kernel and the RTOS timer tick interrupt handler to minimize the amount of stack they use. When this build option is non-zero, all, but two `int`, local variables used by the kernel are declared static, therefore none of the later use stack room. Enabling this feature is mostly useful on target devices with very small data memory. But when the auto variables are set to static, it is quite likely that the run-time performance of the kernel will suffer a bit.

To give an idea to what is the memory saving, assuming a 32 bit processor with 32 bit int and 32 bit pointer, enabling `OS_MIN_STACK_USE` on an application with all the Abassi features enabled, reduces the room required on the stack for the local variables from 84 bytes down to 8 bytes. This translates a reduced stack requirement for each task (as all task access the kernel) of 76 bytes. For the timer tick, still considering the same 32 bit processor, a saving of 12 bytes is achieved. For an application with N tasks, the saving on the stack size is N times 88 (76+12) bytes.

4.1.25 OS_MTX_DEADLOCK

When set to a non-zero value, this build option activates the recursive mutex deadlock protection feature. Simple stated, a mutex deadlock condition occurs if a task tries to lock a mutex that is locked by another task that is blocked on a mutex locked by the first task. The reality is a bit more convoluted because the task locking the mutex can be blocked on another mutex, which is locked by another task, which is blocked on a mutex… and after traversing many mutex / locker pairs then the locker of a mutex that blocks a task in the chain can be the first task that was trying to lock the original mutex.

The mutex deadlock protection detects this condition and does not allow the locking/blocking, and instead reports an error to the caller. More information on mutex deadlock protection is available in Section 9.

Since 2019 when the build option `OS_MTX_DEADLOCK` is negative it will go into the RTOS trap code (See section 6.14.6) instead of returning an error when locking the mutex at fault..

4.1.26 OS_MTX_INVERSION

Setting this build option to a non-zero value includes the code to support one of two commonly used mechanisms to prevent the problem of priority inversion, created under certain conditions when a task blocks on a mutex. When this build option is positive, the priority inheritance mechanism is activated in the build. When this build option is negative, Abassi's intelligent priority ceiling mechanism is activated instead.

The case where only some mutexes would be selected to have protection against priority inversion while others do not is not supported by the Abassi RTOS. Also, the priority inversion protection is either priority inheritance or intelligent priority ceiling; the two mechanisms cannot co-exist in a build, since they were determined during the conception of Abassi as two mutually exclusive mechanisms.

When `OS_MTX_INVERSION` is set to a non-zero value, the build option `OS_PRIO_CHANGE` (Section 4.1.33) is internally forced to a non-zero value in order to allow dynamic priority changes. This is necessary because the priority inheritance and priority ceiling mechanisms rely on raising the priority level of tasks.

If `OS_MTX_INVERSION` is set to a value either greater than 999 or less than -999, it then becomes possible to enable/disable the priority inversion on a per mutex basis.

More information in priority inversion protection is available in Section 7.

4.1.27 OS_MTX_OWN_UNLOCK

The build option `OS_MTX_OWN_UNLOCK` is not required to be defined. When defined and set to a non-zero value, it adds code to restrict the unlocking of a mutex to the mutex owner (that's the task that currently locks the mutex). When the value assigned to `OS_MTX_OWN_UNLOCK` is positive, all mutexes are under the unlocking protection. When the value assigned to `OS_MTX_OWN_UNLOCK` is negative, mutexes are by default under the unlocking protection and the feature can be enabled and disabled with the `MTXignoreOwn()` and `MTXcheckOwn()` components. When this feature is enabled on a mutex, a non-owner task unlocking that mutex is a do-nothing operation, with report of an error.

Availability:

Abassi:	Version 1.245.219 and up
mAbassi:	Version 1.55.55 and up
μAbassi:	Unsupported

4.1.28 OS_NAMES

Names ("C" strings) can be attached to individual tasks, semaphores, mutexes, and mailboxes to identify them at run-time. Setting this build option to a value of zero does not attach names to services; setting it to a non-zero value performs the attachment.

There are three reasons why one would choose to use named services. First, it simplifies the debugging when using the logging facilities; without names, all services could only be identified with the pointer to their descriptor ("C" pointers). Second, it allows the application to not use global variables to access the services. Third, when opening a named semaphore, mutex or mailbox, the Abassi RTOS uses a run-time safe opening technique. The run-time safe feature is such that if the semaphore, mutex or mailbox already exists, the already existing descriptor is returned instead of creating the service. This feature removes the requirement to assign at the design stage which task will create the service: at run-time, the first task using the creation component will be the one that creates the service, and all other tasks running afterward, when using the creation component for the same service, will get the descriptor to the already existing service.

Setting the build option to a positive value creates a local copy of the string in the service descriptor, and a negative value memorizes the pointer to the string in the service descriptor. The latter can save memory on the condition that none of the individual names share memory, e.g. if `sprintf()` is used to create the names, but the same destination string is re-used, then all names will be the same.

If the build option `OS_LOGGING_TYPE` (Section 4.1.17) is set to a non-zero, the build option `OS_NAMES` is internally forced to a non-zero value in order for the logging to report meaningful information.

When OS_NAMES is set to a non-positive value, the build option OS_STATIC_NAME (Section 4.1.48) is internally forced to a value of zero, as there is no need to reserve memory for the different names.

4.1.29 OS_NESTED_INTS

If the target processor supports nested interrupts or if the RTOS interrupt dispatcher, implemented in the assembly file, has been configured to create nesting interrupts, then this build option must be set to non-zero. This is required because even though the Abassi RTOS does not disable interrupts, this statement is true only with non-nested interrupts. A very small critical region exists when interrupts are nested and this build option insert a tiny interrupt protection code. The interrupt dispatcher in the assembly file of many processors can be set to operate with either non-nested, or nested interrupt. The OS_NESTED_INTS build option must be set the same as how the assembly file set-up configures the interrupt dispatcher in there.

On some processors, this build option is overloaded, as it is internally forced to the only mode implemented in the interrupt handler. Consult the processor / compiler porting documents for your target platform to get the full information on what is available.

4.1.30 OS_OUT_OF_MEMORY

The build option OS_OUT_OF_MEMORY is one of the few options that is not required to be defined. When defined and set to a non-zero value, it adds code to verify possible memory allocation issues. Most of the problems encountered when configuring Abassi for an application are related to the amount of memory and the number of services allocated through the OS_STATIC_???? build options. Defining the build option OS_OUT_OF_MEMORY inserts dedicated code that will stop the application exactly where a memory problem occurs. As it is expected this feature is only enabled for debugging, it is assumed a debugger is used and as such, the specific service that is running out of memory can be determined by looking at the source code. The trap code is a macro named OS_CHK_OOM(); if the application is stopped at such a statement, then an out of memory condition was encountered. The build option that must be upgraded is stated in comments.

Alternatively, if the logging facilities (See sections 4.1.17 and 6.12) are enabled, then an error message indicating the build option to upgrade is generated before stopping the application.

When an out of memory condition is detected, the processor enters and remains in the function OStrap(). Refer to OStrap() (Section 6.14.6) for a list of the errors that are trapped.

4.1.31 OS_PERF_MON

The build option OS_PREF_MON is new in Abassi version 1.255.277 and mAbassi version 1.67.66. When defined and set to a non-zero value, it adds a callback to the function PerfMon() immediately before the optional callback OSpreContext() (See section 4.1.32), which is before the context switch.

The performance monitoring facilities collect statistics on all task operations. To do so, a timer is used and through the value assigned to OS_PERF_MON, can be selected between the RTOS timer tick (G_OStimCnt; See section 6.7.15) or a port specific fine time resolution timer:

OS_PERF_MON == 0 :	No performance monitoring
OS_PERF_MON > 0 :	RTOS timer period / OS_PERF_MON
OS_PERF_MON < 0 :	Port Specific timer period / -OS_PERF_MON
OS_PERF_MON == 0x7FFFFFFFL :	Port specific timer set-up by Abassi without call to PerfMon()

Refer to section 6.13 for more information on the performance monitoring facilities.

4.1.32 OS_PRE_CONTEXT

The build option `OS_PRE_CONTEXT` is new in Abassi version 1.255.277 and mAbassi version 1.67.66. When defined and set to a non-zero value, it adds a callback to the function `OSpreContext()` immediately before a context switch. This could be used, for example, for per task data memory bank selection or to add statistics collection, alike the performance monitoring (See section 4.1.31). Application specific data can be held in the task descriptors through the use of the build option `OS_TASK_XTRA_FIELD` (See section 4.1.55).

The function prototype `OSpreContext()` is:

```
void OSpreContext(TSK_t *TaskNext, TSK_t *TaskNow);
```

The argument `TaskNext` is the next task to run, and `TaskNow` is the current running task that will be pre-empted or block upon context switch.

The callback `OSpreContext()`operates inside the kernel, therefore it is not possible to use any Abassi services in the callback. If a service call is performed in the callback, the application will most likely misbehave, even crash. As it is operating inside the kernel, the callback will never be pre-empted (except by interrupts) and in the case of the multi-core mAbassi, switching between when in the callback cannot happen.

4.1.33 OS_PRIO_CHANGE

If the build option `OS_PRIO_CHANGE` is non-zero, it enables the Abassi RTOS to allow the run-time change of priority tasks.

If either the build option `OS_MTX_INVERSION` (Section 4.1.26) or the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) are non-zero, the value of `OS_PRIO_CHANGE` is internally forced to a non-zero value, as both the mutex priority inversion protection and the task starvation protection mechanisms rely on modifying the priority of tasks during run-time.

4.1.34 OS_PRIO_MIN

The value of this build option informs the Abassi RTOS what is the highest numerical value it needs to handle in the application; the largest numerical priority value is the lowest priority level in the application. The Idle Task, or its equivalent, must always be set to the lowest priority; therefore, the numerical value of the Idle Task priority must be set to `OS_PRIO_MIN`. No task can have its priority set to a numerical value larger than `OS_PRIO_MIN`; if this restriction is not respected, it will most likely provoke a crash of the Abassi RTOS.

4.1.35 OS_PRIO_SAME

To enable the Abassi RTOS to support multiple tasks running at the same priority, this build option must be set to a non-zero value. If it is set to zero and multiple tasks at the same priority are created, duplicate priority tasks will never run and will "disappear" from the application.

If either the build option `OS_ROUND_ROBIN` (Section 4.1.36) or `OS_STARVE_WAIT_MAX` (Section 4.1.43) are non-zero, the value of `OS_PRIO_SAME` is internally forced to a non-zero value.

4.1.36 OS_ROUND_ROBIN

Setting this build option to a non-zero value includes the code for round robin (also known as time slicing). The value indicates the maximum period in microseconds for the time slice. This is the maximum continuous CPU time allocated to a running task, when one or more tasks at the same priority are ready to run.

If the value of this build option is negative, it enables the run-time setting of the maximum allowed CPU time per slice on a per task basis. Upon creation, all tasks have their time slice set to `|OS_ROUND_ROBIN|` (absolute value), and this value can be modified during run-time for selected tasks. When the build option is negative, it also becomes possible to set-up the time slice of some task such that they run until completion/blocking, while others tasks at the same priority remain under round robin CPU distribution.

When round robin is enabled, the value assigned to the build option `OS_PRIO_SAME` (Section 4.1.35) is internally overloaded. The value of this build option must be equal or greater than the value of `OS_TIMER_US` (Section 4.1.59); if this condition is not respected, the numerical value of `OS_ROUND_ROBIN` is internally forced to the value of `OS_TIMER_US`, meaning the round robin time slice duration is a single timer tick.

If the build option `OS_COOPERATIVE` (Section 4.1.4) is non-zero, `OS_ROUND_ROBIN` is internally overloaded and forced to a value of zero.

4.1.37 OS_RUNTIME

When this build option is set to a non-zero value, the Abassi RTOS supports the creation of tasks, semaphores, mutexes and mailboxes at run-time. If this build option is positive, the Abassi RTOS still allows the creation of services defined at compile time for tasks (using the macro `TSK_STATIC()`, Section 6.3.2), semaphores (with `SEM_STATIC()`, Section 6.4.2), mutexes (with `MTX_STATIC()`, Section 6.5.2), and `MBX_STATIC()`, Section 6.7.2). When all services are created at compile/link time, with `TSK_STATIC()`, `SEM_STATIC()`, `MTX_STATIC()` and `MBX_STATIC()`, then a significant code size reduction is achieved by setting this build option to zero.

If this option is negative, the services are created at run-time, but the creation of services at compile time is not available. For MISRA-C:2004 compliance, this build option should be set to a negative value.

4.1.38 OS_SEARCH_ALGO

The build option `OS_SEARCH_ALGO` selects the algorithm used to determine the next task to run when the currently running task becomes blocked or suspended. The next task to run could be either a task at the same priority or lower priority.

For most algorithms, an array is used to hold the head of linked lists of tasks ready to run at the same priority. The advantage of the array is that it makes the insertion of a task that became unblocked quite fast. The index is the priority, and all there is left to do is to insert the task in the linked list. But when the running task becomes blocked or suspended, this array needs to be traversed to determine the next task to run. So, the array helps at speeding up the operations after a task gets unblocked, but slows down the operations when a task gets blocked.

Consult the processor/compiler porting documents for your target platform to obtain all the information on how to set this build option in an optimal manner according to your application.

If the build option `OS_COOPERATIVE` (Section 4.1.4) is non-zero, `OS_SEARCH_ALGO` is internally overloaded and forced to a negative value, since in cooperative mode the most efficient search is the 2-dimensional linked list.

4.1.38.1 OS_SEARCH_ALGO == 0

The basic search, when the build option value is zero, simply traverses the array holding the information for each possible priority in the system (`OS_PRIO_MIN`, Section 4.1.34) checking entry after entry until the first lower priority task ready to run is found.

4.1.38.2 OS_SEARCH_ALGO == 1

When the build option is set to 1, the search uses a faster algorithm. An array of bytes (`char`) holds 8 bits indicating if the corresponding priority (priority value == Bit# + 8 * Byte#) has a task ready or not. This `char` array is traversed to find the first non-zero byte entry, which indicates a ready to run task amongst the group of 8. Therefore, for K*8 tasks, a maximum of K iterations is performed for the first step. Once the group of 8 priorities with a ready to run task is found, the basic search is used, starting from the associated index.

4.1.38.3 OS_SEARCH_ALGO > 1

When this build option is greater than 1, the search uses a successive approximation algorithm. First, an array of type `int` holds a bit field indicating if the corresponding priority has a task ready to run or not. This array is traversed to find the first entry with a ready to run task. For example, if the `int` holds 32 bits, for K*32 tasks, a maximum of K iterations is performed for the first step. Second, for `int` of 2^N bit, the entry of the array holds the information for 2^N consecutive priorities. Using successive approximation requires N iterations to find the next task to run. The value to assign to `OS_SEARCH_ALGO` is the power of two over which the search is to be performed; e.g. for 16 bits, a value of 4, and for 32 bits, a value of 5. It is very important to not exceed the size of an `int`: a smaller value is OK.

Note: If the processor does not have a barrel shifter, i.e. the capability to shift data by more than a one bit in a single cycle, using `OS_SEARCH_ALGO` with a value greater than 1 will probably deliver a slower search than if it was set to 1.

4.1.38.4 OS_SEARCH_ALGO < 0

When this build option is negative, the algorithm used does not rely on an array, but instead maintains a 2-dimensional linked list. The array used by the other algorithms is replaced by one dimension of the 2D linked list.

The 2D linked list delivers opposite performance to what the array provides: determining the next task to run is the fastest amongst all algorithms, but unblocking a task is slower, and the task switches during round robin require more CPU.

4.1.39 OS_SEM_XTRA_FIELD

The build option `OS_SEM_XTRA_FIELD` is new in Abassi version 1.264.239 and mAbassi version 1.82.80. When defined and set to a non-zero value, it informs Abassi to add in the semaphore / mutex descriptors `OS_SEM_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the semaphore / mutex descriptor (of type `SEM_t` and `MTX_t`). The application usable entries in `XtraData[]` are the first `OS_SEM_XTRA_FIELD` entries; if Abassi internally needs to use entries in `XtraData[]` it adds extra entries and accesses the entries at and above index `OS_SEM_XTRA_FIELD`.

4.1.40 OS_STACK_CHECK

The build option `OS_STACK_CHECK` is one of the few options that is not required to be defined. When defined and set to a non-zero value, it adds code to perform the monitor the stack usage of each task. If a task encounters a stack overflow, right before a context switch, a logging message is generated (if logging enable through `OS_LOGGING_TYPE`, see Section 4.1.17) and the application is stopped: the interrupts are globally disabled and an infinite loop is entered. This method of trapping the stack overflows was selected as the most likely need for the stack monitoring is during the development and debugging phase. One must remember it is still possible that an application crash happens before the monitoring feature can trap the stack overflow as the overflow could corrupt anything in the application. When an overflow is trapped, the descriptor of task that has suffered the stack overflow is held in the RTOS internal global variable `G_OStaskNow`.

When the stack monitoring is enable, this feature also give access to components to get the maximum stack size that has been used and the minimum unused stack size (`TSKstkFree()` and `TSKstkUsed()`, see section 6.3.27 and 6.3.28). Using these two components can greatly ease the process of selecting the optimal stack size for each task.

Releases on or after 2019 add the capability to check the stack of the current (or all cores in the case of mAbassi) every time the kernel is entered (this does not apply when a kernel request is performed in an interrupt). If the build option `OS_STACK_CHECK` is set to a positive value, only the stack of the task to be blocked before the context switch is checked as was done in the original implementation. If the build option `OS_STACK_CHECK` is set to a negative value, then the stack(s) is (are) is also checked every time the kernel is entered. A total of `-OS_STACK_CHECK` words entries at the top of the stack are checked, up to a maximum of 8 entries. If the value of `OS_STACK_CHECK` is less than -8 the number of words checked remains at 8.

When a stack overflow is detected, the processor enters and remains in the function `OStrap()`. Refer to `OStrap()` (Section 6.14.6) for a list of the errors that are trapped.

4.1.41 OS_STARVE_PRIO

When the task starvation protection (Modified Priority Aging) feature is enabled with the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) set to a non-zero value, `OS_STARVE_PRIO` specifies the priority threshold of the task starvation protection. All tasks operating at a priority lower than (or numerical value larger than) `OS_STARVE_PRIO` become protected against the problem of task starvation. When these tasks are in the ready to run state, but never achieve the running state, they get their priority increased by one level every `OS_STARVE_WAIT_MAX` timer ticks (up to the priority `OS_STARVE_PRIO`). This priority increase occurs until they reach the running state, and run long enough. Once their priority is `OS_STARVE_PRIO` the task remains at that priority until it runs long enough.

If the value of this build option is negative, it enables the run-time setting of the priority threshold on a per task basis. When negative, upon creation, all tasks have their priority threshold value set to `|OS_STARVE_PRIO|` (absolute value), and this value can be modified during run-time for selected tasks.

Tasks at the lowest priority (`OS_PRIO_MIN`, Section 4.1.34) are never under starvation protection. This was done since the Idle Task is the lowest priority task and, in many power sensitive applications, the Idle Task is the task that puts the processor into sleep mode. Having the Idle Task under starvation protection would periodically put the processor into sleep when it shouldn't be.

If the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) is set to zero, which means the starvation protection feature is not part of the build, then `OS_STARVE_PRIO` is internally forced to a zero value.

4.1.42 OS_STARVE_RUN_MAX

When the task starvation protection feature is enabled with the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) set to a non-zero value, `OS_STARVE_RUN_MAX` specifies the maximum number of timer ticks a task is allowed to run at an increased priority.

If the value of this build option is negative, it enables the run-time setting of the maximum run time on a per task basis. Upon creation, all tasks have their maximum run time value set to `|OS_STARVE_RUN_MAX|` (absolute value), and this value can be modified during run-time for selected tasks.

If the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) is set to zero, which means the starvation protection feature is not part of the build, then `OS_STARVE_RUN_MAX` is internally forced to a zero value.

If the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) is set to non-zero, which means the starvation protection feature is part of the build, then `OS_STARVE_RUN_MAX` must be set to a non-zero value.

4.1.43 OS_STARVE_WAIT_MAX

This build option enables the task starvation protection when it is set to a non-zero value. A non-negative value specifies the maximum number of timer ticks a task can remain at the same priority level when stuck in the ready to run state. If the task never achieves the running state during this time, the task gets its priority level raised, and the sequence goes again until the task reaches the running state or the maximum priority level specified by `OS_STARVE_PRIO` (Section 4.1.41).

When `OS_STARVE_WAIT_MAX` is set to a non-zero value, both build options `OS_PRIO_CHANGE` (Section 4.1.33) and `OS_PRIO_SAME` (Section 4.1.35) are internally forced to a non-zero value in order to allow dynamic priority changes and the existence of tasks at the same priority. This is necessary because the task starvation protection relies on raising the priority level of tasks one level at a time, and because of that, two or more tasks will sometimes have the same priority.

If the value of this build option is negative, it enables the run-time setting of the maximum wait time on a per task basis. Upon creation, all tasks will have their maximum wait time value set to `|OS_STARVE_WAIT_MAX|`, (absolute value), and this value can be modified during run-time for selected tasks.

If this build option is non-zero, the build option `OS_TIMER_US` (Section 4.1.59) must be set to the correct value used in the application otherwise an error message will be generated at compile time.

4.1.44 OS_STATIC_BUF_MBLK

The build option `OS_MEM_STATIC_BUF_MBLK` is one of the few options that is not required to be defined. When defined and set to a positive value it informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the memory used for the buffers of all the memory block management descriptor. The value specifies the total number of bytes to reserve for all memory block pool created in the application. Refer to section 6.10 for more details on how to determine the minimum memory size that must be reserved. If this build option is not defined or is defined and zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time when the memory block management service are part of the build when `OS_MEM_BLOCK` (Section 4.1.23) is set to a non-zero value.

If this build option is non-zero, then the build option `OS_STATIC_MBLK` (Section 4.1.46) must also be set to a non-zero value.

If the build option `OS_MEM_BLOCK` (Section 4.1.23) is zero, this build option is internally forced to zero.

4.1.45 OS_STATIC_BUF_MBX

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the buffers of all mailboxes. The value specifies the total number of buffer elements (or messages) to reserve, and it must be set to a value greater or equal to the sum of all mailbox buffer sizes. If this build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time.

If this build option is non-zero, then the build option `OS_STATIC_MBX` (Section 4.1.47) must also be set to a non-zero value.

If the build option `OS_MAILBOX` (Section 4.1.18) is zero, this build option is internally forced to zero.

4.1.46 OS_STATIC_MBLK

The build option `OS_MEM_STATIC_MBLK` is one of the few options that is not required to be defined. When defined and set to a positive value it informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the descriptors of all the memory block management; this is not the memory used by the buffers in the pool, it is the memory to hold the descriptor of the memory pools. The value specifies the total number of memory block management descriptors to reserve, and it must be set to a value greater or equal to the total number mailboxes in the application. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time when the memory block management service are part of the build when `OS_MEM_BLOCK` (Section 4.1.23) is set to a non-zero value.

If the build option `OS_MEM_BLOCK` (Section 4.1.23) is zero, this build option is internally forced to zero

4.1.47 OS_STATIC_MBX

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the descriptors of all mailboxes. The value specifies the total number of mailbox descriptors to reserve, and it must be set to a value greater or equal to the total number mailboxes in the application. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time .

If the build option `OS_MAILBOX` (Section 4.1.18) is zero, this build option is internally forced to zero

This build option is not related in any way to the macro `MBX_STATIC()` (Section 6.7.2). The latter creates and initializes a mailbox at compile/link time, while the build option `OS_STATIC_MBX` reserves the memory used to create mailboxes at run time.

4.1.48 OS_STATIC_NAME

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the names of all tasks, semaphores, mutexes and mailboxes. These services are named when the build option `OS_NAMES` (Section 4.1.28) is non-zero. The memory reserved by `OS_STATIC_NAME` is used when names strings are copied; i.e. when the build option `OS_NAMES` is positive. The value of the build option `OS_STATIC_NAME` specifies the total number of characters (including the terminating null characters) to reserve, and it must be greater or equal to the total memory needed for all the names in the application. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run-time.

If the build option `OS_NAMES` is set to zero, then `OS_STATIC_NAME` is internally forced to zero.

4.1.49 OS_STATIC_SEM

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the semaphore and mutex descriptors. The value specifies the total number of semaphore and mutex descriptors to reserve, and it must be greater or equal to the total number semaphores and mutexes in the application. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time.

This build option is not related in any way to the macros `SEM_STATIC()` (Section 6.4.2) or `MTX_STATIC()` (Section 6.5.2). `SEM_STATIC` and `MBX_STATIC` create and initialize a semaphore or mutex at compile/link time, while the build option `OS_STATIC_MBX` reserves the memory used to create these services at run-time.

4.1.50 OS_STATIC_STACK

Setting this build option to a positive value informs the Abassi RTOS to reserve the memory needed by the stacks of all task (excluding Adam & Eve and the Idle Task; the size of the Idle Task stack is specified with the build option `OS_IDLE_STACK`, see Section 4.1.16) at compile/link time. The value specifies the number of `char` to reserve. There is no overhead to include to this number; e.g. if the application has 9 tasks requiring 1024 `char` each, the minimum value to specify is 9216 (9*1024). Specifying a larger value will waste memory. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the stack memory at run-time.

The stack size reserved when using the special macro `TSK_STATIC()` (Section 6.3.2) that creates and initializes tasks at compile/link does not use of the memory reserved by `OS_STATIC_STACK`.

4.1.51 OS_STATIC_TASK

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the task descriptors, and it must be greater or equal to the total number of tasks in the application. The tasks associated to the functions `IdleTask()` and `main()` must be included in this value. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run

This build option is not related in any way to the macro `TSK_STATIC()` (Section 6.3.2). The latter creates and initializes a task at compile/link time, while the build option `OS_STATIC_TASK` reserves the memory used to create tasks at run time.

Note: If the build option `OS_STATIC_TASK` is set to zero and the build option `OS_IDLE_TASK` (section 4.1.16) is non-zero, provision must be made to make sure enough data space is available for the allocation of the memory required by the task descriptor used by the Idle Task. This specifically applies when `OS_ALLOC_SIZE` (Section 4.1.1) is non-zero, then Idle Task descriptor will use memory from this pre-allocate memory pool.

4.1.52 OS_STATIC_TIM_SRV

Setting this build option to a positive value informs the Abassi RTOS to reserve memory at compile/link time. This memory is used to hold the timer service descriptors. The value specifies the total number of timer services descriptors to reserve, and it must be greater or equal to the total number timer services in the application. If the build option is zero, `OSalloc()` (Section 6.14.3) is used to dynamically allocate the memory at run time.

This build option is not related in any way to the macro `TIM_STATIC()` (Section 6.9.1). `TIM_STATIC` creates and initializes a timer service at compile/link time, while the build option `OS_STATIC_TIM_SRV` reserves the memory used to create the timer services at run-time.

4.1.53 OS_SYS_CALL

The build option `OS_SYS_CALL` is optional and only needed when the Abassi System Calls Layer is included in an application. When the System Calls Layer is one of the components of an application, then `OS_SYS_CALL` must be defined and set to a non-zero value. For more information on the System Calls Layer, refer to [R6].

4.1.54 OS_TASK_SUSPEND

If the application needs to put tasks in the suspended state, then this build option must be set to a non-zero value. When tasks only suspend themselves, there is no need to turn on this feature; it is only needed when Task A suspends Task B. When this build option is set to a non-zero value, a special feature is supported in the Abassi RTOS that ensures when a task is to be suspended that it does not go into the suspended state until it has unlocked all the mutexes it owns.

4.1.55 OS_TASK_XTRA_FIELD

The build option `OS_TASK_XTRA_FIELD` is new in Abassi version 1.255.277 and mAbassi version 1.67.66. When defined and set to a non-zero value, it informs Abassi to add in the task descriptors `OS_TASK_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the task descriptor (of type `TSK_t`). The field `XtraData`[] is always located at the beginning of the task descriptor data structure, right after the preserved stack pointer entry; therefore, if access to `XtraData[]` must be performed in an assembly coded module, the location is not affected by any of the build options.

There are some ports that could use the first few entries of the field `XtraData[]`, although all `OS_TASK_XTRA_FIELD` entries are available, the first available entry may not be at index 0. One example is with GCC using the newlib "C" library; if re-entrance protection is enabled by setting the port specific build option `OS_NEWLIB_REENT` to a non-zero value, then the first entry in `XtraData[]` is internally used by Abassi. The "C" define `OX_TASK_1ST_XTRA_FIELD` is provided by Abassi to indicate the index of the first free entry in `XtraData[]`; entries in `XtraData[]` at indexes less than `OX_TASK_1ST_XTRA_FIELD` are not available and must never be modified.

4.1.56 OS_TIM_EXTRA_FIELD

The build option `OS_TIM_XTRA_FIELD` is new in Abassi version 1.264.239 and mAbassi version 1.82.80. When defined and set to a non-zero value, it informs Abassi to add in the timer services descriptors `OS_TIM_XTRA_FIELD` scratch pad entries. These entries are of standard "C" type `intptr_t`, meaning they can hold either an `int` or any types of pointer. The data is accessed through the field `XtraData[]` in the timer service descriptor (of type `TIM_t`). This build option is ignored if the build option `OS_TIMER_SRV` (Section 4.1.59) is set to 0. The application usable entries in `XtraData[]` are the first `OS_TIM_XTRA_FIELD` entries; if Abassi internally needs to use entries in `XtraData[]` it adds extra entries and accesses the entries at and above index `OS_TIM_XTRA_FIELD`.

4.1.57 OS_TIMEOUT

The Abassi RTOS offers the option of using expiry timeouts when a task is waiting for a semaphore, trying to lock a mutex, waiting for an event, or retrieving/depositing from/in a mailbox. Setting this build option to a non-zero value includes the code to support this feature during the compilation.

When the timeout argument of components that require such information has a positive value positive, and `OS_TIMEOUT` is set to a value of zero, it remaps the positive timeout value into a value of zero. When `OS_TIMEOUT` is negative, the positive timeout value arguments are remapped to a negative value. More information is given in Section 4.2.11.1 on the purpose of these two ways for disabling the timeouts.

If this build option is a positive value, the build option `OS_TIMER_US` (Section 4.1.59) must be set to the correct value used in the application, otherwise an error message will be generated at compile time.

4.1.58 OS_TIMER_CB

This build option instructs the Abassi RTOS to perform a callback to a user function when the timer facilities (see `OS_TIMER_US`, Section 4.1.59) are enabled. This function must be named `TIMcallBack()` (Section 6.7.16). A value of zero disables the timer callback facilities. A positive value specifies the period of the callback; e.g. if `OS_TIMER_CB` is set to a value of 4, it will make the timer service call `TIMcallBack()` once every 4 timer periods, as specified by `OS_TIMER_US` build option. If the build option `OS_TIMER_CB` is positive, `OS_TIMER_US` must be set to the correct value.

Note: The callback function always operates within the timer interrupt context.

4.1.59 OS_TIMER_SRV

This build option includes the code to support the timer services. The timer services allow an application to perform a delayed operation or periodic operations.

4.1.60 OS_TIMER_US

If the timer facilities are used (see `OS_ROUND_ROBIN`, Section 4.1.36, `OS_STARVE_WAIT_MAX`, Section 4.1.43, `OS_TIMEOUT`, Section 4.1.57, and `OS_TIMER_CB`, Section 4.1.58), this build option specifies, in microseconds, the period of the timer; a value of zero disables the timer facilities.

The value specified must be positive.

4.1.61 OS_USE_TASK_ARG

Each task can be supplied with a pointer to any type of information. This feature is useful when the same code/function is used to implement multiple tasks. A non-zero value for this build option enables this capability.

Most RTOS supply this information as the argument to the task function. For code compatibility when this feature is either enabled or disabled, the information is not exchanged through the function arguments, but it is exchanged using the `TSKsetArg()` (Section 6.3.19) and `TSKgetArg()` (Section 6.3.6) components.

4.1.62 OS_WAIT_ABORT

The build option `OS_WAIT_ABORT` is an option that does not need to be defined. When defined and set to a non-zero value, it make available components to force an un-blocking of all the tasks blocked on a service. This is alike forcing a timeout expiry on all task blocked on a service, even if the blocking was a forever blocking. The components made available are `SEMabort()`, `MTXabort()`, `EVTabort()` and `MBXabort()`. If the build option `OS_WAIT_ABORT` is zero or it is not defined, then these four components are not available.

4.2 Build Option Selection

This section explains, step by step, how to set each of the build options to configure the Abassi RTOS to fulfill all the needs of the target application. Reading through every one of the following sub-section should be sufficient for the reader to understand what values to set for each of the build options described in the previous section.

4.2.1 Cooperative

If it is desired to have the Abassi RTOS operate in a cooperative mode (emulation of a cooperative RTOS), set the build option `OS_COOPERATIVE` to a non-zero value, as shown in the following table:

Table 4-1 Build option cooperative mode

```
#define OS_COOPERATIVE   1   /* Operate in cooperative mode, not preemptive         */
```

To have the RTOS operate in its native preemptive mode, set the build option `OS_COOPERATIVE` to a value of zero, as shown in the following table:

Table 4-2 Build option for preemptive mode (native)

```
#define OS_COOPERATIVE   0   /* Operate in preemptive  mode, not cooperative        */
```

4.2.2 Priority Span

When an application is architected to use a RTOS, one key decision to make is to select the number of different priority levels the application will require for all the tasks. Once this number is determined, the Abassi RTOS is informed of that value through the build option `OS_PRIO_MIN`:

Table 4-3 Build option OS_PRIO_MIN

```
#define OS_PRIO_MIN    PPP    /* Lowest priority numerical value                    */
```

The value `PPP` to set for the build option `OS_PRIO_MIN` is the numerical value of the lowest task priority required in the application, which is not exactly the number of priority levels. As the highest priority level has a numerical value of 0, if an application requires `N` distinct priorities, then the build option `OS_PRIO_MIN` must be set to `N-1`. `OS_PRIO_MIN` is the numerical priority value at which the Idle Task operates.

4.2.3 One or multiple tasks at same priority

Abassi offers support of either a single task per priority or multiple tasks at the same priority. When a single task per priority is selected, some code and CPU usage reduction is achieved. But if either round robin or the task starvation protection mechanism is needed, the Abassi RTOS must be configured for multiple tasks at the same priority. When any of the two previous features is activated, Abassi internally overrides the `OS_PRIO_SAME` build option to force the system to use multiple tasks per priority.

If the system is configured for a single task per priority, and the priority inversion protection is enabled, one must be very careful on the priority assigned to each task. More explanations are given in the Priority Inversion section (Section 7) on how to deal with priority inversion protection in an application with one task per priority.

To configure the RTOS to only support a single task per priority, the build option `OS_PRIO_SAME` must be set to a value of zero, as shown in the following table:

Table 4-4 Build option for one task per priority

```
#define OS_PRIO_SAME    0    /* Do not support multiple tasks at the same priority   */
```

If the application requires multiple tasks at the same priority, set the build option `OS_PRIO_SAME` to a non-zero value, as shown in the next table:

Table 4-5 Build option for many tasks at the same priority

```
#define OS_PRIO_SAME    1    /* Support multiple tasks at the same priority          */
```

4.2.4 Task suspension

Having Task A capable of suspending Task B is an optional feature in Abassi. It was decided to make it optional as many applications do not need to suspend tasks.

Abassi offers a unique feature when a task is to be suspended: if the task to suspend locks one or more mutexes, the suspension is postponed until the task relinquishes all locks on mutexes. This feature safeguards the application against creating a mutex deadlock. If the application does not need to suspend tasks, then disabling this feature saves code and CPU.

To configure the RTOS to <u>not</u> support task suspension, the build option `OS_TASK_SUSPEND` must be set to a value of zero, as shown in the following table:

Table 4-6 Build option without task suspension

```
#define OS_TASK_SUSPEND   0  /* Tasks cannot be suspended                            */
```

If the application needs to suspend tasks, then set `OS_TASK_SUSPEND` to a non-zero value, as shown in the next table:

Table 4-7 Build option with task suspension

```
#define OS_TASK_SUSPEND   1  /* Tasks can be suspended                              */
```

When the task suspension feature is not part of a build, it is still possible for a task to self-suspend. Having the task still able to self-suspend was determined to be allowable as it is assumed that when the application will need to have a task self-suspend, the operation will be only performed when the task does not lock any mutexes.

4.2.5 Mailboxes

The mailbox service is an optional feature of the Abassi RTOS. If the application would benefit by having access to a mailbox system, then set the `OS_MAILBOX` built option to a non-zero value, as shown in the following table:

Table 4-8 Build option to include mailboxes

```
#define OS_MAILBOX       1    /* Enable the handling of mailboxes                   */
```

If there is no need for mailboxes in the application, set the build option `OS_MAILBOX` to zero, as shown in the following table:

Table 4-9 Build option to not include mailboxes

```
#define OS_MAILBOX       0    /* Disable the handling of mailboxes                  */
```

4.2.6 Events

Events flags are a simple synchronization mechanism, and are another optional feature of the Abassi RTOS. If the application would benefit by using event flags, then set the `OS_EVENTS` built option to a non-zero value, as shown in the following table:

Table 4-10 Build option to include event flags

```
#define OS_EVENTS       1    /* Enable the handling of event flags                  */
```

If there is no need for event flags in the application, set the build option `OS_EVENTS` to zero, as shown in the following table:

Table 4-11 Build option to not include event flags

```
#define OS_EVENTS       0    /* Disable the handling of event flags                 */
```

4.2.7 First Come First Served

By default, all blocking services unblock tasks in a priority ordering. This means that if two or more tasks are blocked, when they try to access the same unavailable service, the first task to get unblocked upon resource availability is the task with the highest priority amongst all blocked task. This mode of unblocking can be modified to operate on a *First Come First Served* and enabled on a case by case basis. When a resource is configured to operate in the *First Come First Served* mode, the first task that got blocked is the first task to get unblocked. To be able to have services operate in a *First Come First Served* mode, the build option `OS_FCFS` must be set to a non-zero value, as shown in the next table:

Table 4-12 Build option to support FCFS

```
#define OS_FCFS    1         /* Support the FCFS unblocking mode                    */
```

If there is no need for *First Come First Serve* unblocking ordering in the application, set the build option `OS_FCFS` to zero, as shown in the following table:

Table 4-13 Build option to not support FCFS

```
#define OS_FCFS    0         /* Do not support the FCFS unblocking mode             */
```

4.2.8 Task Arguments

It is possible to exchange information between tasks through the "Task Argument" feature. This is alike what most RTOS offer, which is an argument to the function attached to the task. In Abassi it was decided to not use the function arguments because if argument passing is never used, this means some code in the RTOS is useless. As this is optional, passing information through the function argument becomes problematic, as the function prototype is different if the feature is enable or disable. Instead, two RTOS components (see `TSKgetArg()`, Section 6.3.6) and `TSKsetArg()`, Section 6.3.19) are supplied to perform the exchange operation.

If there is a need to exchange arguments in the application, set the build option `OS_USE_TASK_ARG` to a non-zero value:

Table 4-14 Build option to support task arguments

```
#define OS_USE_TASK_ARG    1     /* Support task arguments                              */
```

If there is no need to exchange arguments in the application, set the build option `OS_USE_TASK_ARG` to zero:

Table 4-15 Build option to not support task Arguments

```
#define OS_USE_TASK_ARG    0     /* Do not support task arguments                       */
```

4.2.9 Data Memory

The Abassi RTOS requires memory to hold the information needed by the descriptors of all the services. This data memory can be distributed using 4 different methods. The first 3 methods are used for run-time creation of services, while the fourth method does not create the services at run-time but instead creates them at compile/link time:

- Dynamically allocated through the generic `malloc()` facility
- Dynamically allocated through a facility internal to Abassi
- Reserved memory pool per service
- Static services created at compile and link time

Each method has advantages and disadvantages.

- `malloc()` uses the generic "C" library memory allocation facility and requires almost no set-up. But, as malloc() is a is paired with `free()`, there is always extra memory allocated to manage the allocation / release of the memory blocks. In the case of Abassi, as this memory is never released, this extra memory becomes wasted data memory. If the target platform is tight on the data memory available, `malloc()` may not be a good choice. If there is plenty of data memory, then `malloc()` is quite likely the best choice. However, not all `malloc()` implementations are multithread-safe. When Abassi internally uses `malloc()`, a mutex is assigned to `malloc()` to protect it, but using `malloc()` anywhere else in the application is not protected. If it is necessary to make `malloc()` multithread-safe, see Section 12, or use `OSalloc` (Section 6.14.3) instead.
- Abassi's internal dynamic memory allocator operates exactly like `malloc()`, except it does not use extra memory as `malloc()` does. Therefore, compared to `malloc()`, the data memory is fully utilized without losses. The set-up for this memory allocation method is quite simple, but when the memory is exhausted, there is no report on the condition.
- Instead of using dynamic memory allocation to retrieve from a common memory pool the aggregate size needed for each service, dedicated memory pools can be created: one memory pool for the tasks, another for the semaphores/mutexes, and one for the mailboxes. Compared to the previous method, this method is a bit simpler to use, as it does not require the user to know what are the memory sizes of the different services. The disadvantage of this method is that because the memory is distributed amongst multiple pools, there is a need for very good planning of the all the services needed in the application in order to determine the optimal sizes to reserve.
- The compile/link time creation of the services is the easiest and most optimal method as there is never a danger of running out memory, and only the needed memory is reserved. The disadvantage is that, depending on the compiler, it may not be possible to restart the application without reloading the binary image on the processor. This situation happens if the compiler (or compiler configuration) does not reload the initialized data upon start-up. When the compiler (or compiler configuration) re-initializes data upon start-up, this means the pre-initialized data is kept in a different part of the data memory, translating into a doubling of the required memory.

4.2.9.1 Data Memory with malloc()

To use `malloc()` for the run-time creation of services, the build options must be set as shown in the following table. The comments on the right should be sufficient to allow the reader to understand what Abassi is doing internally.

Table 4-16 Build options for data memory allocation with malloc()

```
#define OS_ALLOC_SIZE        0    /* No memory reserved for Abassi's allocator      */
#define OS_RUNTIME           1    /* Run-time memory allocation / service creation   */
#define OS_STATIC_MBX        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_MBX_BUF    0    /* No memory reserved for this memory pool         */
#define OS_STATIC_NAME       0    /* No memory reserved for this memory pool         */
#define OS_STATIC_SEM        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_STACK      0    /* No memory reserved for this memory pool         */
#define OS_STATIC_TASK       0    /* No memory reserved for this memory pool         */
```

4.2.9.2 Data Memory with Abassi's allocator

To use Abassi's internal memory allocator for the run-time creation of services, the build options must be set as shown in the following table. The comments on the right should be sufficient to allow the reader to understand what Abassi is doing internally.

Table 4-17 Build options for Abassi's data memory allocator

```
#define OS_ALLOC_SIZE        NNN  /* Memory (bytes) reserved for Abassi's allocator  */
#define OS_RUNTIME           1    /* Run-time memory allocation / service creation   */
#define OS_STATIC_MBX        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_MBX_BUF    0    /* No memory reserved for this memory pool         */
#define OS_STATIC_NAME       0    /* No memory reserved for this memory pool         */
#define OS_STATIC_SEM        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_STACK      0    /* No memory reserved for this memory pool         */
#define OS_STATIC_TASK       0    /* No memory reserved for this memory pool         */
```

In the above table, the definition of `OS_ALLOC_SIZE`, set to `NNN`, must be a numerical value that fulfills the needs of the application, meaning a numerical value specifying a memory size greater or equal to the number of bytes (`char`) of all the descriptors, stacks, and name strings created in the application.

4.2.9.3 Data memory with memory pools

To make Abassi use individual memory pools for the run-time creation of services, the build options must be set as shown in the following table. The comments on the right should be sufficient to allow the reader to understand what Abassi is doing internally. Consult the previous section (Section 4.1) to obtain more details on the numerical values to select.

Table 4-18 Build options for multiple memory pools

```
#define OS_ALLOC_SIZE        0    /* Memory (bytes) reserved for Abassi's allocator  */
#define OS_RUNTIME           1    /* Run-time memory allocation / service creation   */
#define OS_STATIC_MBX        AAA  /* Number of mailboxes in the application          */
#define OS_STATIC_MBX_BUF    BBB  /* Total number of buffers for the mailboxes       */
#define OS_STATIC_NAME       CCC  /* Total number of bytes to hold the names         */
#define OS_STATIC_SEM        DDD  /* Total number of semaphores & mutexes in the app */
#define OS_STATIC_STACK      EEE  /* Total number of bytes for all task stacks       */
#define OS_STATIC_TASK       FFF  /* Total number of tasks in the application        */
#define OS_STATIC_TIM_SRV    GGG  /* Total number of timer services in the app       */
```

If a memory pool is set to zero (`AAA` -> `FFF`) and the corresponding service is created at run-time, then dynamic memory allocation is used as a fall back mechanism. If the build option `OS_ALLOC_SIZE` is non-zero, it will be Abassi's internal memory allocator that will be used; if `OS_ALLOC_SIZE` is zero, then `malloc()` is used.

4.2.9.4 Data memory at compile time

To not have Abassi create services at run-time, the build options must be set as shown in the following table. When Abassi is configured this way, a significant amount of code is eliminated. The comments on the right should be sufficient to allow the reader to understand what Abassi is doing internally.

Table 4-19 Build options for allocation at compile / link time

```
#define OS_ALLOC_SIZE        0    /* Memory (bytes) reserved for Abassi’s allocator  */
#define OS_RUNTIME           0    /* No Run-time memory allocation/service creation  */
#define OS_STATIC_MBX        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_MBX_BUF    0    /* No memory reserved for this memory pool         */
#define OS_STATIC_NAME       0    /* No memory reserved for this memory pool         */
#define OS_STATIC_SEM        0    /* No memory reserved for this memory pool         */
#define OS_STATIC_STACK      0    /* No memory reserved for this memory pool         */
#define OS_STATIC_TASK       0    /* No memory reserved for this memory pool         */
#define OS_STATIC_TIM_SRV    0    /* No memory reserved for this memory pool         */
```

4.2.10 Idle Task

The Abassi RTOS needs to always have one or more tasks ready to run / running at the lowest priority level. This task is known as the Idle Task, meaning it is a task that absorbs the left over CPU when the application has nothing else to process. Abassi can automatically create the Idle Task, or the application can “manually” create it. The choice is selected with the build option `OS_IDLE_STACK`.

To inform Abassi to not create the Idle Task, the `OS_IDLE_STACK` build option must be set to zero, as shown in the following table:

Table 4-20 Build option to not create the Idle Task

```
#define OS_IDLE_STACK       0    /* Do not automatically create the Idle Task        */
```

To inform Abassi to create the Idle Task and make it ready to run, the `OS_IDLE_STACK` build option must be set to a non-zero value, as shown in the following table:

Table 4-21 Build option to create the Idle Task

```
#define OS_IDLE_STACK     KKK    /* Automatically create the Idle Task               */
```

The `KKK` in the definition must be set to the numerical value specifying how many bytes (or `char`) to reserve for the stack of the Idle Task.

4.2.11 Timer

The Abassi timer facility is required when an application uses any of the following services:

- Expiry timeout on blocking services
- Round Robin
- Task starvation protection
- Timer services

If none of the above services are needed in the application, then Abassi has no need for the timer facility, and the build options should be set as shown in the following table:

Table 4-22 Build options when the RTOS timer is not used

```
#define OS_ROUND_ROBIN       0    /* No round robin                              */
#define OS_STARVE_PRIO       0    /* No starvation protection                    */
#define OS_STARVE_RUN_MAX    0    /* No starvation protection                    */
#define OS_STARVE_WAIT_MAX   0    /* No starvation protection                    */
#define OS_TIMEOUT           0    /* No timeout on blocking services             */
#define OS_TIMER_CB          0    /* No timer, so no timer callback              */
#define OS_TIMER_SRV         0    /* No timer, so no timer services              */
#define OS_TIMER_US          0    /* The timer is not used                       */
```

If no timer dependent features are required, you can skip to Section 4.2.12, as the build options for timeout, round robin, task starvation protection, and timer services are described in the following 4 sub-sections.

If the timer facility is required because one or more of the features requiring the timer are used in the application, the build options for the timer should be set as shown in Table 4-23. The build options that were specified in the previous table, that are not present in Table 4-23, are described in the next 4 sub-sections.

Table 4-23 Build options when the RTOS timer is used

```
#define OS_TIMER_CB       NNN     /* Timer period to perform the callbacks          */
#define OS_TIMER_US       MMM     /* Timer is used and its period is MMM µs         */
```

In the above example, `MMM` must be set to the numerical value that specifies the period of the timer in microsecond units. If there is no need for a callback, set `NNN` to zero; otherwise, set `NNN` to the timer tick period at which the application needs the callback. For example, if a callback is needed at every timer tick, set `NNN` to 1; if a callback is needed only once every 10 timer ticks, set `NNN` to the value of 10.

One may wonder why `OS_TIMER_US` requires a value in microsecond units and not a simple Boolean flag, since Abassi does not configure the timer peripheral that generates the timer interrupts. The numerical value of `OS_TIMER_US` is used by the time converter components (Section 6.7.17) to give the designer access to traditional time durations. Specifying time related constants using the converter components instead of using the number of timer ticks makes the application independent from the timer period. This is important if the timer period has to be changed at any time, since the application code can remain the same.

4.2.11.1 Timeout

The availability of timeouts on blocking services is controlled by the `OS_TIMEOUT` build option. If timeouts are used in the application, set the `OS_TIMEOUT` build option to a non-zero value, as shown in the next table:

Table 4-24 Build option when timeouts are used

```
#define OS_TIMEOUT        1       /* Enable timeouts on blocking services           */
```

When the timeout argument of components that require such information is positive, and timeouts are disabled, it is possible to either convert the positive timeout arguments into an infinite timeout (same as setting the argument to a negative value) or to remap them to no timeout (same as setting the argument to a value of 0). If `OS_TIMEOUT` is zero, positive timeout arguments are remapped to a zero argument. When `OS_TIMEOUT` is set to a negative value, positive timeout arguments are remapped to a negative argument. The purpose of this offering is to allow the disabling of timeouts in an application that was initially developed using positive timeouts as a debugging mechanism. Setting `OS_TIMEOUT` to a value of zero remaps the positive arguments to an abnormal condition, where the service should always be available. With `OS_TIMEOUT` set to a negative value, it remaps the positive arguments into a normal condition, where the service is not always immediately available but it will always become available within an acceptable time window.

If timeouts are not needed by the application, set the `OS_TIMEOUT` build option to a non-positive, as shown in the two next tables:

Table 4-25 Build option when timeouts are not used (+ve map to 0)

```
#define OS_TIMEOUT        0       /* No timeouts on blocking services             */
```

Table 4-26 Build option when timeouts are not used (+ve map to -ve)

```
#define OS_TIMEOUT        -1      /* No timeouts on blocking services             */
```

4.2.11.2 Round Robin

When the RTOS timer is enabled, it is possible to apply round robin to tasks at the same priority that are ready to run. If the build option `OS_COOPERATIVE` is non-zero, round robin is not available, therefore set the build option `OS_ROUND_ROBIN` to a value of zero, as shown in the following table, and skip this section.

Table 4-27 Round Robin setting when `OS_COOPERATIVE` is non-zero

```
#define OS_ROUND_ROBIN    0       /* Round Robin is not active                   */
```

The round robin can be configured in two ways: either all tasks are allocated the same time slice duration, or the time slice duration can be individually set for each task. Round robin is enabled when the build option `OS_ROUND_ROBIN` is set to a non-zero value. When the value is positive, all tasks are allocated a fixed time slice duration in microseconds units, specified in the definition. When the value is negative, all tasks upon creation have a default time slice duration of the absolute of the value, in microsecond units, but can be later modified at run time on a per task basis.

Table 4-28 No Round Robin

```
#define OS_ROUND_ROBIN    0       /* Round Robin is not active                   */
```

Table 4-29 Fixed Round Robin

```
#define OS_ROUND_ROBIN    UUU     /* Round Robin time slices are fixed at UUU us     */
```

Table 4-30 Programmable Round Robin

```
#define OS_ROUND_ROBIN    -UUU    /* RR time slice programmable, default is UUU us   */
```

When round robin is enabled in the run-time configurable mode, it becomes possible to have co-existence of round robin and running until blocking/completing for tasks at the same priority level (see Section 10).

4.2.11.3 Task Starvation Protection

When the RTOS timer is enabled, it is possible to activate a modified Priority Aging mechanism that is unique to Abassi. This custom priority aging is called Task Starvation Protection and is configured with three build options:

- ➢ OS_STARVE_PRIO
- ➢ OS_STARVE_RUN_MAX
- ➢ OS_STARVE_WAIT_MAX

The first build option, OS_STARVE_PRIO (Section 4.1.41), indicates the priority that tasks under starvation protection cannot exceed. As the task starvation feature increases the priority of a starved task until it runs, this build option specifies the priority level that cannot be exceeded. The build option is the numerical value of the priority.

The build option OS_STARVE_RUN_MAX (Section 4.1.42) specifies the maximum number of timer ticks a task under starvation protection will run at an increased priority.

Finally OS_STARVE_WAIT_MAX (Section 4.1.43) specifies the maximum number of timer ticks a task under starvation protection will wait at a priority level before having its priority level increased by 1 level.

Each of these build options can be set to a positive or negative value. When the build option value is positive, the value applies to all tasks and it cannot be modified at run-time. When the build option is negative, all tasks upon creation have their task starvation parameter set to the absolute of the build option, and the corresponding parameter is run-time modifiable on a per task basis.

Table 4-31 Task Starvation Protection

```
#define OS_STARVE_PRIO      PPP    /* Threshold priority                          */
#define OS_STARVE_RUN_MAX   RRR    /* Maximum run time at promoted priority       */
#define OS_STARVE_WAIT_MAX  WWW    /* Maximum wait time at same priority          */
```

Setting the build option OS_STARVE_PRIO to the same value as the build option OS_PRIO_MIN is set to will disable the task starvation protection. This special condition is handled in such a way that the task starvation protection code is not included in the build. On the other side, setting OS_STARVE_PRIO to negative OS_PRIO_MIN specifies that all tasks, when created, are not under task starvation protection. But, as the negative value indicates run-time modification of the parameter, the code for the task starvation protection is part of the build and individual tasks can be set to be under the task starvation protection.

NOTE: Tasks at the lowest priority (OS_PRIO_MIN) are never put under starvation protection. This is design intent as it provides a "class" of low priority tasks that are not put under protection when the starvation build options are positive.

4.2.11.4 Timer Services

When the RTOS timer is enabled, it is possible to include an optional "timer services" module. This module provides a simple way to add the functionality where operations can be delayed or performed periodically. To include the code and API for the timer services, set the build option OS_TIMER_SRV to a non-zero value, as shown in the following table:

Table 4-32 Timer Services

```
#define OS_TIMER_SRV        1      /* Timer services are included in the build      */
```

4.2.12 Priority inversion protection

Abassi supports two of the most common priority inversion protection mechanisms: priority inheritance and priority ceiling, with enhancements.

To disable the priority inversion protection mechanism, set the build option `OS_MTX_INVERSION` to a value of zero, as shown in the following table:

Table 4-33 Priority Inversion Protection Disabled

```
#define OS_MTX_INVERSION    0    /* Priority Inversion Protection is de-activated    */
```

To enable priority inversion protection and use the priority inheritance mechanism, set the build option `OS_MTX_INVERSION` to a positive value, as shown in the following table:

Table 4-34 Priority Inheritance Enabled

```
#define OS_MTX_INVERSION    1    /* Priority Inheritance enabled                     */
```

To enable priority inversion protection and use Abassi's intelligent priority ceiling mechanism, set the build option `OS_MTX_INVERSION` to a negative value, as shown in the following table:

Table 4-35 Priority Ceiling Enable

```
#define OS_MTX_INVERSION    -1    /* Priority Ceiling enabled                        */
```

When the priority inversion protection is enabled, it is either the priority ceiling mechanism or the priority inheritance mechanism. These two mechanisms are mutually exclusive. Also, when one of the two mechanisms is enabled, all mutexes in the application are under priority inversion protection; it is not possible to apply the protection only on some mutexes and not others.

4.2.13 Mutex Deadlock protection

Abassi is capable of detecting a deadlock condition when a task tries to lock a mutex. A detailed description on how Abassi's mutex deadlock protection operates is given in Section 9. To disable the mutex deadlock protection, set the build option `OS_MTX_DEADLOCK` to a value of zero as shown in the following table:

Table 4-36 Disabling Mutex Deadlock Protection

```
#define OS_MTX_DEADLOCK    0     /* Mutex deadlock protection is NOT active         */
```

To activate the mutex deadlock protection, set the build option `OS_MTX_DEADLOCK` to a non-zero value as shown in the following table:

Table 4-37 Enabling Mutex Deadlock Protection

```
#define OS_MTX_DEADLOCK    1     /* Mutex deadlock protection is active           */
```

4.2.14 Interrupt Queue sizing

In Abassi, in order for the kernel to never disable interrupts, and to allow the interrupt handler to operate as fast as possible, when a kernel request is performed in an interrupt, the kernel request is queued for processing outside of the interrupt context. Doing so shortens the duration of the interrupt processing and minimizes the time required to enter the kernel when back-to-back interrupts occur. The requests are queued, and the queue needs to be sized large enough to absorb the worst case of accumulation of kernel requests. The word *accumulation* used here is important as the sizing for the worst case is not the largest number of kernel requests done inside a single interrupt handler. One must understand that the queue is filled when interrupts perform requests to the kernel, and these interrupts could occur non-stop, back-to-back. The kernel retrieves the requests from the queue outside interrupt contexts, and depending on the rate of arrival of the interrupts, the kernel may not be fast enough to empty all requests between interrupts.

Don't be scared by this worst case situation, as it is really a worst case because the kernel typically processes requests from interrupts within a few hundreds of CPU cycles. On most processors, this worst case would require interrupts occurring constantly every 2 to 3 microseconds to starve the kernel[1]. One way or another, as the queue elements are fairly small, it does not hurt to oversize this queue intentionally.

Table 4-38 Interrupt queue sizing

```
#define OS_MAX_PEND_RQST   NN     /* Size of the queue for the ISR kernel requests   */
```

The numerical value NN must be set to one more than the strict minimum, because the extra one is needed to implement the queue for maximum performance. So, if the application can have a maximum of 5 requests pending, then the queue size must be set to a value of 6. Also, setting the value to a power of 2 is highly desirable as this reduces the number of CPU cycles needed to insert and retrieve the request to/from the queue. In the previous example, the queue size would be set to 8, capable of holding 7 unprocessed requests.

As an initial guideline, setting the queue to a value of 16 or 32 should be sufficient in most applications, assuming the number of interrupt sources can be counted on your fingers.

4.2.15 Logging

To observe Abassi's internal reaction to kernel requests, it is possible to enable a logging facility. If logging is not needed, set the build option `OS_LOGGING_TYPE` to a value of zero, as shown in the following table:

Table 4-39 Disabling the logging facilities

```
#define OS_LOGGING_TYPE   0     /* Kernel operations are not logged               */
```

[1] If an application has to continuously handle interrupts occurring constantly at such a fast rate, it may be preferable to revert to the ubiquitous *loop* to implement such an application instead of utilizing a RTOS.

There are two ways to extract the logging performed by the RTOS. The first and most simple method is to have the kernel send the information over an ASCII output device. This logging mechanism is enabled by setting the build option `OS_LOGGING_TYPE` to 1, as shown in the following table. The output device is accessed through the RTOS component `OSputchar()` (Section 6.14.5).

Table 4-40 Activating the ASCII logging dump

```
#define OS_LOGGING_TYPE   1     /* Kernel operations are logged through OSputchar()  */
```

The other logging mechanism collects the individual requests and reactions of the kernel in packets, deposited in a circular buffer. The circular buffer content can be delivered to an output device at anytime, using the logging buffer reader API. This type of logging is enabled by setting the build option `OS_LOGGING_TYPE` to a value greater than 1, and the specified value informs Abassi about the circular buffer size to reserve for the logging. The size indicated by the build option is the number of logging packets, not the size in bytes (the packets hold 3 `int` and 1 pointer).

Table 4-41 Activating the buffered logging

```
#define OS_LOGGING_TYPE   NN     /* Kernel operations logged in a circular buffer    */
```

4.3 Build Examples

There are a lot of options when configuring the Abassi RTOS, and at the beginning, selecting the correct values for the desired build configuration can be a challenge. The previous section should have eliminated much of the confusion. To further assist, a few examples are given in the following sub-sections.

4.3.1 Minimum feature set

A RTOS built with the minimum feature set offers unnamed semaphores, and unnamed tasks that cannot have the same priority. In this example, the RTOS is build to supports 5 different priorities. The descriptors are created using `TSKcreate()` (Section 6.3.2), `SEMopen()` (Section 6.4.5) or `MTXopen()` (Section 6.5.11) using `malloc()`, with dynamic memory allocation, and the application supplies the Idle Task function named `IdleTask()`.

Table 4-42 Build with minimum feature set

```
#define OS_ALLOC_SIZE         0      ← ==0, OS_RUNTIME & OS_STATC_???: use malloc
#define OS_COOPERATIVE        0
#define OS_EVENTS             0
#define OS_FCFS               0
#define OS_IDLE_STACK         512    ← IdleTask() created with stack of 512 bytes
#define OS_LOGGING_TYPE       0
#define OS_MAILBOX            0
#define OS_MAX_PEND_RQST      32     ← ISR queue is 31 requests deep
#define OS_MTX_INVERSION      0
#define OS_NAMES              0
#define OS_NESTED_INTS        0
#define OS_PRIO_CHANGE        0
#define OS_PRIO_MIN           4      ← Allowed priority values are 0,1,2,3,4
#define OS_PRIO_SAME          0
#define OS_ROUND_ROBIN        0
#define OS_RUNTIME            1      ← Services are created at run time (malloc)
#define OS_SEARCH_ALGO        0
#define OS_STATIC_BUF_MBX     0      ← Services are created at run time (malloc)
#define OS_STATIC_MBX         0      ← Services are created at run time (malloc)
#define OS_STATIC_NAME        0      ← Services are created at run time (malloc)
#define OS_STATIC_SEM         0      ← Services are created at run time (malloc)
#define OS_STATIC_STACK       0      ← Services are created at run time (malloc)
#define OS_STATIC_TASK        0      ← Services are created at run time (malloc)
#define OS_TASK_SUSPEND       0
#define OS_TIMEOUT            0
#define OS_TIMER_CB           0
#define OS_TIMER_US           0
#define OS_USE_TASK_ARG       0
```

4.3.2 Minimum feature with static memory creation

The next example creates a RTOS with exactly the same feature set as the previous example, except this time the task's descriptor, stack, semaphores, and mutexes descriptors are created during run-time using memory reserved at compile/link time. Dynamic memory allocation through `OSalloc()` (Section 6.14.3) is not used anymore. The example reserves memory for 5 tasks, 3 semaphores/mutexes, and 4096 bytes for the task stacks.

Table 4-43 Build with static memory

```
#define OS_ALLOC_SIZE        0      ← See OS_RUNTIME / OS_STATIC_???: memory pools
#define OS_COOPERATIVE       0
#define OS_EVENTS            0
#define OS_FCFS              0
#define OS_IDLE_STACK        512
#define OS_LOGGING_TYPE      0
#define OS_MAILBOX           0
#define OS_MAX_PEND_RQST     32
#define OS_MTX_INVERSION     0
#define OS_NAMES             0
#define OS_NESTED_INTS       0
#define OS_PRIO_CHANGE       0
#define OS_PRIO_MIN          4
#define OS_PRIO_SAME         0
#define OS_ROUND_ROBIN       0
#define OS_RUNTIME           1      ← at runtime, Memory pools non zero
#define OS_SEARCH_ALGO       0
#define OS_STATIC_BUF_MBX    0
#define OS_STATIC_MBX        0
#define OS_STATIC_NAME       0
#define OS_STATIC_SEM        3      ← The application can have up to 3 semaphores
#define OS_STATIC_STACK      4096   ← Memory available for all stacks is 4096 bytes
#define OS_STATIC_TASK       5      ← The application can have up to 5 tasks
#define OS_TASK_SUSPEND      0
#define OS_TIMEOUT           0
#define OS_TIMER_CB          0
#define OS_TIMER_US          0
#define OS_USE_TASK_ARG      0
```

4.3.3 Minimum feature with compiled time creation

The next example again creates a RTOS with exactly the same feature set as the previous example, except this time the task descriptors, stack, and semaphores descriptors are created and initialized at compile time using the macro definitions `TSK_STATIC()` (Section 6.3.2), `SEM_STATIC()` (Section 6.4.2) and `MTX_STATIC` (sect 6.5.2). Run-time memory allocation/set-up is not used anymore.

Table 4-44 Build with compile time memory

```
#define OS_ALLOC_SIZE        0      ← see OS_RUNTIME / OS_STATIC_???: compile time
#define OS_COOPERATIVE       0
#define OS_EVENTS            0
#define OS_FCFS              0
#define OS_IDLE_STACK        512
#define OS_LOGGING_TYPE      0
#define OS_MAILBOX           0
#define OS_MAX_PEND_RQST     32
#define OS_MTX_INVERSION     0
#define OS_NAMES             0
#define OS_NAMES             0
#define OS_NAMES             0
#define OS_NESTED_INTS       0
#define OS_PRIO_CHANGE       0
#define OS_PRIO_MIN          4
#define OS_PRIO_SAME         0
#define OS_ROUND_ROBIN       0
#define OS_RUNTIME           0      ← is now 0
#define OS_SEARCH_ALGO       0
#define OS_STARVE_WAIT_MAX   0
#define OS_STARVE_PRIO       0
#define OS_STARVE_TIME_MAX   0
#define OS_STATIC_BUF_MBX    0
#define OS_STATIC_MBX        0
#define OS_STATIC_NAME       0
#define OS_STATIC_SEM        0      ← is now 0
#define OS_STATIC_STACK      0      ← is now 0
#define OS_STATIC_TASK       0      ← is now 0
#define OS_TASK_SUSPEND      0
#define OS_TIMEOUT           0
#define OS_TIMER_CB          0
#define OS_TIMER_US          0
#define OS_USE_TASK_ARG      0
```

4.3.4 Adding the timer

The next example again creates a RTOS with exactly the same feature set as the previous example, except this time the timer service is enabled. That is, timeouts are made available to the components that can block tasks, and an application specific timer callback is used. The timer is configured to operate once every 10 *m*s., the timer callback is configured to be entered once every second, and the round robin time slicing allocates a maximum of 50 *m*s. per task. Having enabled the round robin feature, `OS_PRIO_SAME` is automatically set internally but it is still set here for consistency.

Table 4-45 Build with timer

```
#define OS_ALLOC_SIZE         0
#define OS_COOPERATIVE        0
#define OS_EVENTS             0
#define OS_FCFS               0
#define OS_IDLE_STACK         512
#define OS_LOGGING_TYPE       0
#define OS_MAILBOX            0
#define OS_MAX_PEND_RQST      32
#define OS_MTX_INVERSION      0
#define OS_NAMES              0
#define OS_NESTED_INTS        0
#define OS_PRIO_CHANGE        0
#define OS_PRIO_MIN           4
#define OS_PRIO_SAME          1      ← Tasks can have the same priority
#define OS_ROUND_ROBIN        50000  ← Round robin once every 50 ms (50000 µs)
#define OS_RUNTIME            0
#define OS_RUNTIME            0
#define OS_RUNTIME            0
#define OS_SEARCH_ALGO        0
#define OS_STARVE_WAIT_MAX    0
#define OS_STARVE_PRIO        0
#define OS_STARVE_TIME_MAX    0
#define OS_STATIC_BUF_MBX     0
#define OS_STATIC_MBX         0
#define OS_STATIC_NAME        0
#define OS_STATIC_SEM         0
#define OS_STATIC_STACK       0
#define OS_STATIC_TASK        0
#define OS_TASK_SUSPEND       0
#define OS_TIMEOUT            1      ← Component expiry timeout enable
#define OS_TIMER_CB           100    ← 1 s. period (once every 100 timer ticks)
#define OS_TIMER_US           10000  ← Timer period 10 ms (10000 µs)
#define OS_USE_TASK_ARG       0
```

4.3.5 Adding many features

The next example again creates a RTOS with exactly the same feature set as the previous example, except this time many features have been enabled, and the run-time dynamic memory allocation is used.

Table 4-46 Build with many features

```
#define OS_ALLOC_SIZE        0      ← With OS_RUNTIME & OS_STATIC_???: use malloc
#define OS_COOPERATIVE       0
#define OS_EVENTS            1      ← Event flags available
#define OS_FCFS              1      ← First Come First Served available
#define OS_IDLE_STACK        512
#define OS_LOGGING_TYPE      0
#define OS_MAILBOX           1      ← Mailboxes available
#define OS_MAX_PEND_RQST     32
#define OS_MTX_INVERSION     0
#define OS_NAMES             1      ← Services have a name attached to them
#define OS_NESTED_INTS       0
#define OS_PRIO_CHANGE       1      ← Priority can be changed at run time
#define OS_PRIO_MIN          4
#define OS_PRIO_SAME         1
#define OS_ROUND_ROBIN       50000
#define OS_RUNTIME           1      ← Services created at run time
#define OS_SEARCH_ALGO       0
#define OS_STARVE_WAIT_MAX   0
#define OS_STARVE_PRIO       0
#define OS_STARVE_TIME_MAX   0
#define OS_STATIC_BUF_MBX    0
#define OS_STATIC_MBX        0
#define OS_STATIC_NAME       0
#define OS_STATIC_SEM        0
#define OS_STATIC_STACK      0
#define OS_STATIC_TASK       0
#define OS_TASK_SUSPEND      1      ← Tasks can be suspended at run-time
#define OS_TIMEOUT           1
#define OS_TIMER_CB          100
#define OS_TIMER_US          10000
#define OS_USE_TASK_ARG      1      ← Tasks can have argument passed to them
```

5 Quick Start

Here is a small example with only 3 tasks. In `main()`, the Abassi RTOS is started by calling `OSstart()`. When `OSstart()` has initialized everything and returns, the `main()` function becomes attached to the highest priority task in the application (priority numerical value is 0). Typically, all tasks would be created in `main()`.

In this example, 2 tasks and 1 semaphore are created. This example can use any set of build options that were described in the previous section.

Table 5-1 Quick Start Example

```
#include "Abassi.h"

SEM_t *SemHi;

int main()
{
   OSstart();                        /* RTOS initialization                  */

   SemHi = SEMopen("Sem Hi");        /* 1 semaphore in this application      */

   TSKcreate("Lo Prio", 2, 1024, &FctLo, 1);
   TSKcreate("Hi Prio", 1, 1024, &FctHi, 1);

   TSKsetPrio(TSKmyID, OS_PRIO_MIN); /* Adam & Eve is converted to the Idle   */

   for(;;);                          /* Infinite loop as Idle                */
}

void FctLo(void)                     /* Low priority task                    */
{
   for (;;) {
       puts("Low priority running");
       SEMpost(SemHi);
   }
}

void FctHi(void)                     /* High priority task                   */
{
   for (;;) {
       puts("High priority running");
       SEMwait(SemHi, -1);           /* -1 means waiting forever             */
   }
}
```

One semaphore is created by using the component `SEMopen()` with the name of the semaphore, `"Sem Hi"`. A `NULL` pointer could also have been used to create an unnamed semaphore. Two tasks are created by using the component `TSKcreate()`. The arguments for `TSKcreate()` are from left to right: the name of the task, the priority of the task, the stack size of the task, the function to use for the task, and a Boolean indicating if the task is created in the ready to run state or in the suspended state. `NULL` pointers could have been used instead of the names `"Lo Prio"` and `"Hi Prio"` to create unnamed tasks. Tasks are requested to be created in the ready to run state, therefore `TSKresume()` is not needed to put them in that state. At this point, both tasks remain in the ready to run state because the current task is Adam & Eve, which is the highest priority task; all other tasks are at lower priority. Setting Adam & Eve priority to the lowest priority level in the application allows the next highest lower priority task (than Adam & Eve) to run.

`FctHi()` is the highest priority ready to run task, so it becomes the running task, waiting forever on the semaphore `SemHi`, which hasn't been posted. `FctHi()` then blocks on the semaphore `SemHi`. The next priority ready to run task is `FctLo()`, which becomes the running task. The semaphore `SemHi` is posted in `FctLo()`, which unblocks `FctHi()`, pre-empting `FctLo()`, and remains ready to run. This sequence continues indefinitely.

6 Components

This section describes every one of the components the Abassi RTOS services offer. Many of these components are not real functions but are defined as a call to `Abassi()`, with the appropriate arguments. `Abassi()` is the sole function, providing all kernel functionality. Due to the complexity of the code in `Abassi()`, this function is not described in this document. The components can be a real function, a simple definition, or a more complex macro definition. Inline functions are not used as they are not MISRA-C:2004 compliant.

Some components should never be used in interrupts, and others can be used with some care. For more information about using components in an interrupt handler, refer to section 3.3.2.

6.1 Component Type

Components are implemented using many different programming techniques. The following sub-sections described the possible ways components can be implemented and the short-comings.

The selection of what method to use to implement a component was determined with a single goal: minimizing the overall code size of an application, and, similarly, the CPU requirements. All kernel requests are implemented as "C" pre-processor definitions, thereby hiding the syntax for the kernel request. Almost all other components are implemented using macro definitions. This was selected as the preferred method since almost all components require only a few "C" statements. If, instead, a function was used, the overall code size generated by the compiler would have most likely been significantly bigger than what the macro delivers. This is due to the need for a function call set-up, which is typically not as lightweight as inline code.

As an example of this, have a look at the two snippets of code below, both implementing a memory byte filling operation:

Table 6-1 Function call

```
memset(Ptr, 0, 60);
```

Table 6-2 Code from a macro definition

```
do {
int _ii;
char *_Pch=(char *)(Ptr);

    for(_ii=0 ; _ii<60 ; _ii++) {
        *_Pch++ = 0;
    }

} while(0)
```

On some compilers/processors, both snippets of code are generated with exactly the same code size. When the code size is identical or very close, there is no advantage to using a function. Because of their extremely low complexity, almost all Abassi components fall under this advantage of a macro definition vs. a function call. (In reality, the library function `memset()` may have been optimized for real-time, using 32 or 64 bit write to memory instead of 1 char per iteration, but the spirit of this example remains.)

6.1.1 Atomic Macro

An Atomic macro is a "C" definition (or a macro definition) that performs a direct request to the kernel. It can be a simple definition remapping one-to-one the component arguments into the kernel function arguments, or it can be a macro definition with some pre-processing. Atomic macros components are tagged as safe and unsafe. An unsafe Atomic macro component is simply an unsafe macro, where one or more arguments are used multiple times in the definition, meaning they should never have a modifier such as ++.

6.1.2 Function

A function component is simply a component implemented as a "C" function, or it could be a function wrapped under a macro definition that maps one-to-one the arguments of the macro to the function arguments.

6.1.3 Macro

A macro is a "C" definition (or a macro definition) that does not perform a request to the kernel. It can be a simple definition remapping one-to-one the component arguments into the kernel function arguments, or it can be a macro definition with some pre-processing. Macros components are tagged as safe and unsafe. An unsafe macro component is simply an unsafe macro, where one or more arguments are used multiple times in the definition, meaning they should never have a modifier such as ++.

6.1.4 Data Access

A data access component is a component that either reads from or writes to a data variable. The variable can be a field in a service descriptor, or it can be one of the kernel variables.

6.2 System components

This section described all system-wide components. The system components that are described in the following sub-sections are:

Table 6-3 System Component list

Section	Name	Description
6.2.2	`OSstart`	Initialization of the Abassi RTOS

6.2.1 Description

There is a single system-wide component: `OSstart()`, which sets-up the Abassi RTOS operations. This component initializes all the global variables to allow the Abassi RTOS to be restarted without reloading the application binary image. It attaches the information needed by the Abassi RTOS to identify `main()` as a task with the highest priority (numerical value of 0), and it optionally creates and resumes the Idle Task (according to the build option `OS_IDLE_STACK`, Section 4.1.16).

`OSstart()` is the only component required to establish the Abassi RTOS environment and it must be called before any Abassi RTOS component is used.

6.2.2 OSstart

Synopsis

```
#include “Abassi.h”

void OSstart(void);
```

Description

The component `OSstart()`initializes the Abassi RTOS. It is the first component to activate before using any other components of the Abassi RTOS. This is typically done as one of the first statements in `main()`. In `OSstart()`, all internal variables used by the Abassi RTOS are initialized, `main()` is set-up to be the running task, which is set to the highest priority, and if the build option `OS_IDLE_STACK` is non-zero, the application supplied function `IdleTask()` is assigned to a task with the lowest priority (`OS_PRIO_MIN`) and is put into the ready to run state.

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Many build options affects the internal operations performed by `OSstart()`. From the user point of view, this component is always used the same way, no matter what are the build options.

Notes

Interrupts cannot be enabled until after the component `OSstart()` has been used. And when interrupts are to be enabled, the component `OSintOn()` should be used.

See also

`OS_IDLE_STACK` (Section 4.1.16)
`OS_PRIO_MIN` (Section 4.1.34)
`OSintOn()` (Section 6.8.5)

6.3 Task Components

This section describes all components related to the tasks. The task components described in the following sub-sections are:

Table 6-4 Task Component list

Section	Name	Description
6.3.2	`TSK_STATIC`	Create a task at compile / link time
6.3.3	`TSK_SETUP`	Initialization of a task that was created at compile / link time
6.3.4	`return`	Returning from a task and suspend it
6.3.5	`TSKcreate`	Create a task at runtime
6.3.6	`TSKgetArg`	Retrieve the argument in a task
6.3.7	`TSKgetID`	Obtain the descriptor of another task
6.3.8	`TSKgetPrio`	Get the current priority of a task
6.3.9	`TSKgetRR`	Get the round robin slice time used by a task
6.3.10	`TSKgetStrvPrio`	Get the starvation priority threshold used by a task
6.3.11	`TSKgetStrvRunMax`	Get the starvation maximum run time allowed to a task
6.3.12	`TSKgetStrvWaitMax`	Get the starvation maximum wait time allowed to a task
6.3.13	`TSKisBlk`	Report if a task is in the blocked state
6.3.14	`TSKisRdy`	Report if a task is in the ready to run state
6.3.15	`TSKisSusp`	Report if a task is in the suspended state
6.3.16	`TSKmyID`	Obtain the descriptor of the currently running task
6.3.17	`TSKresume`	Resume a suspended task
6.3.18	`TSKselfSusp`	Self-suspend a task
6.3.19	`TSKsetArg`	Set the argument to a task
6.3.20	`TSKsetPrio`	Set the current priority of a task
6.3.21	`TSKsetRR`	Set the round robin slice time used by a task
6.3.22	`TSKsetStrvPrio`	Set the starvation priority threshold used by a task
6.3.23	`TSKsetStrvRunMax`	Set the starvation maximum run time allowed to a task
6.3.24	`TSKsetStrvWaitMax`	Set the starvation maximum wait time allowed by a task
6.3.25	`TSKsleep`	Set the current priority of a task
6.3.26	`TSKstate`	Report the state of a task: ready to run, blocked or suspended
6.3.27	`TSKstkFree`	Report the minimal amount of unused space on a task stack
6.3.28	`TSKstkUsed`	Report the maximum amount of used space on a task stack
6.3.29	`TSKsuspend`	Suspend a task
6.3.30	`TSKtimeoutKill`	Terminate an active timeout that blocks a task
6.3.31	`TSKtout`	Change an active timeout that blocks a task

6.3.32	`TSKyield`	Yield the CPU

6.3.1 Description

The task components give access to all the resources needed to create, to modify the operation of, or to retrieve information on tasks in an application. To exist, a task must first be created with the `TSKcreate()` component or the `TSK_STATIC()` and `TSK_SETUP()` components. When a task is created, a stack, a priority level, and a name (ignored if the build option `OS_NAMES` (Section 4.1.28) is zero) are assigned to it. Newly created tasks are either in the suspended state or the ready to run / running state; the after-creation state is selected with an argument to `TSKcreate()` or `TSK_SETUP()`. To make the task running or ready to run when it was created in the suspended state, the component `TSKresume()` is used.

6.3.2 TSK_STATIC

Synopsis

```
#include "Abassi.h"

TSK_STATIC(VarName, TskName, Prio, StackSize, Fct);
```

Description

`TSK_STATIC()` is a special component that creates a task and initializes the task descriptor. It is a macro definition that creates a static object, so none of the arguments has a real data type. The task is not created at run time; it is created at compile/link time.
It is not possible to completely initialize the stack of a task when it is created using `TSK_STATIC()`. Therefore, it is absolutely necessary to use the component `TSK_SETUP()` during run-time for each and every tasks that has been created using `TSK_STATIC()`.

Availability

Always.

Arguments

`VarName`	Name of the variable holding the pointer to the task descriptor to create / initialize. This variable name is the pointer to the task descriptor used by all task related components. This is a variable name, therefore do not put double quotes around the name.
`TskName`	Task name. This is not the variable name, but is the name attached to the task. As it is a "C" string, double quotes around the name are required. `G_OSnoName` , and not `NULL`, should be used for an unnamed task.
`Prio`	Priority to assign to the task (0 is highest, `OS_PRIO_MIN` the lowest) .
`StackSize`	Number of `char` allocated to the task's stack.
`Fct`	Function to use for the task; this is not a pointer to the function, it is the name of the function. The function prototype must be declared before `TSK_STATIC()` uses it.

Returns

N/A

Component type

Macro (unsafe)

Options

If the build option `OS_NAMES` is set to a value of zero, the argument `TskName` is ignored but must still be supplied.

Notes

When a task has been created and initialized with `TSK_STATIC()`, and access to the task descriptor is required in another file, the task descriptor must be imported as shown in the following example:

Table 6-5 Usage of TSK_STATIC with TSK_SETUP

```
/* File #1 */

TSK_STATIC(MyTask, "Task Name", 4, 512, &TaskFct);


/* File #2 */

extern TSK_t *MyTask;
```

A task created and initialized with `TSK_STATIC()` will not be part of the search done with `TSKgetID()`. If the component `TSKgetID()` is used with the name of such a task, the return value will be `NULL`, unless another task with the exactly the same name was created using `TSKcreate()`.
If tasks are created using the `TSK_STATIC()` component, it may not be possible to restart the application without reloading the binary image on the processor. This situation happens if the compiler (or compiler configuration) does not reload the initialized data upon start-up.

If Abassi is used in a C++ environment, the function attached to the task (the argument `Fct`) must be declared with "C" linkage (see section 3.5), not as a regular C++ function.

See also

`OS_IDLE_STACK` (Section 4.1.16)
`OS_NAMES` (Section 4.1.28)
`OS_PRIO_MIN` (Section 4.1.34)
`OS_RUNTIME` (Section 4.1.37)
`OSstart()` (Section 6.2.2)
`TSK_SETUP()` (Section 6.3.3)
`TSKcreate()` (Section 6.3.5)
`TSKgetID()` (Section 6.3.7)
`TSKresume()` (Section 6.3.17)
`G_OSnoName` (Section 6.14.2)

6.3.3 TSK_SETUP

Synopsis

```
#include "Abassi.h"

void TSK_SETUP(VarName, State);
```

Description

`TSK_SETUP()` is a special component that completes the initialization of a task statically created and initialized at compile/link time with `TSK_STATIC()`. This component must always be used when a task is created and initialized with the component `TSK_STATIC()`. It completes the initialization and optionally resumes the task from the suspended state. Contrary to `TSK_STATIC`(), which is a static declaration, `TSK_SETUP()` is a real-time operation, therefore it must be run before any other component is used on the task. If the task isn't resumed with `TSK_SETUP()`, then `TSK_SETUP()` must run before the task is resumed with `TSKresume()`.

Availability

Always.

Arguments

`VarName`	Name of the variable for the task descriptor. This must be exactly the same name as the variable name (first argument) that was used in `TSK_STATIC()`. This is a variable name therefore do not put double quotes around the name.
`State`	State of the task after initialization:
	`0` The task is set in the suspended state
	`non-0` The task is set in the ready to run or running state

Returns

`void`

Component type

Macro (unsafe)

Options

Notes

See also

`TSK_STATIC()` (Section 6.3.2)
`TSKresume()` (Section 6.3.17)

6.3.4 return

Synopsis

```
return;
```

Description

When the statement `return` is used in the function that was attached to a task, the task gets suspended.

Availability

Always

Arguments

N/A

Returns

N/A

Component type

N/A

Options

Notes

Never use the "C" statement `return` in the function `main()`, as the task associated to `main()` is completely different from all other tasks. The function `main()` is not attached to a task, but the task Adam & Eve is attached to the function. If `return` is used in Adam & Eve, the application will exit, which means all operations will be terminated.
Depending on the setting of the build option `OS_TASK_SUSPEND`, either the component `TSKselfSusp()` or the component `TSKsuspend()` is used after the return. A task where a return was performed can always be resumed with the component `TSKresume()`.

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKresume` (Section 6.3.17)
`TSKselfSusp` (Section 6.3.18)
`TSKsuspend` (Section 6.3.29)

6.3.5 TSKcreate

Synopsis

```
#include "Abassi.h"

TSK_t *TSKcreate (const char *Name, int Prio, int StackSize,
                  void(*Fct)(void), int State);
```

Description

`TSKcreate()` is the run-time component needed to create a new task. When a task is created, its descriptor is allocated and initialized, a function is assigned to the task, a priority is given to it, and a stack is allocated, initialized and attached to the task. After being created, if the task was requested to be created in the suspended state, then it should later be moved to the running or ready to run state with the component `TSKresume()`.

Availability

`TSKcreate()` is only available when the build option `OS_RUNTIME` is non-zero.

Arguments

`Name`	Name to associate to the task. `NULL` can be used for an unnamed task.
`Prio`	Priority to assign to the task (a value of 0 is the highest priority) .
`StackSize`	Number of `char` allocated to the task's stack.
`Fct`	Pointer to the function to attach to the task.
`State`	State of the task after initialization: `0` The task is created in the suspended state `non-0` The task is created in the ready to run or running state

Returns

Task descriptor

Component type

Function
- Cannot be used in an interrupt -

Options

If the build option `OS_NAMES` is set to zero, the argument `Name` is ignored but must still be provided.
If the build option `OS_STATIC_TASK` is non-zero, the task descriptor uses memory that was allocated/reserved at compile/link time instead of memory dynamically allocated at run-time.
If the build option `OS_STATIC_STACK` is non-zero, the stack allocated to the task uses memory that was allocated/reserved at compile/link time instead of memory dynamically allocated at run-time.

Starting with Abassi version 1.273.262, mAbassi version 1.94.97, and µAbassi version 1.42.30, an optional extra argument is supported. When the argument `StackSize`, the base of the stack buffer is supplied by the caller instead of being internally allocated by `TSKcreate()`. The function prototype then becomes:

```
TSK_t *TSKcreate (const char *Name, int Prio, int StackSize,
                  void(*Fct)(void), int State, void *Buffer);
```

The stack size is – `StackSize`. The buffer specified by the argument `Buffer` must be dimensioned to at least – `StackSize` bytes. If the target platform requires stack alignment, Abassi will internally make sure the stack is properly aligned event if the supplied buffer isn't.

Notes

When using a statically allocated stack, as specified with `OS_STATIC_STACK`, the numerical value of `StackSize` should always be set to an exact multiple of 4 or 8 (a multiple of 8 is suggested). Many processors require access to memory aligned to the size of the data to access. By allocating the stack size in multiples of 4 or 8, the processor alignment restrictions will be respected. This is not required when using dynamic memory allocation based on `OSalloc()`, as the latter guarantees proper alignment for all data types. This constraint of stack size being a multiple of 4 or 8 could have been added in the code of `TSKcreate()`, but it was decided not to for the two following reasons. First, it adds code and second, when a designer sets the value of the build option `OS_STATIC_STACK`, the rounding up of the amount of memory to allocate to a multiple of 4 or 8 could prematurely exhaust the reserved stack memory without knowledge of the designer.
The numerical value for the argument `Prio` must never be greater than the value of the build option `OS_PRIO_MIN`.

If Abassi is used in a C++ environment, the function attached to the task (the argument `Fct`) must be declared with "C" linkage (see section 3.5), not as a regular C++ function.

See also

`OS_NAMES` (Section 4.1.28)
`OS_PRIO_MIN` (Section 4.1.34)
`OS_RUNTIME` (Section 4.1.37)
`OS_STATIC_STACK` (Section 4.1.50)
`OS_STATIC_TASK` (Section 4.1.51)
`OSalloc()` (Section 6.14.3)
`TSKresume()` (Section 6.3.17)

6.3.6 TSKgetArg

Synopsis

```
#include "Abassi.h"

void *TSKgetArg(void);
```

Description

`TSKgetArg()` is a component that extracts an argument, which has been forwarded to a task with the component `TSKsetArg()`. A generic `void` pointer is all that is forwarded to the task. Therefore, this pointer, once casting has been applied to it, can refer to any type and size of data. Using a forwarding approach, instead of passing arguments (alike the `argc` and `argv[]` for `main()`) to the task's function, was retained to first minimize the complexity of passing arguments to a task, but mainly to allow dynamic changes of a task arguments. This could be useful, for example, when suspending/resuming a task in order to modify its behavior. Task arguments are very useful when a single function implements more than a one task which operates a bit differently from task to task.

Availability

`TSKgetArg()` is only available when the build option `OS_USE_TASK_ARG` is non-zero.

Arguments

`void`

Returns

Last pointer used when the component `TSKsetArg()` was applied on the task.

Component type

Data access
- Meaningless in an interrupt -

Options

Notes

Any data type can be forwarded to a task using this component. One has to remember it is the pointer that is forwarded, not the contents. Therefore the contents should not be modified after having forwarded the pointer with `TSKsetArg()`, unless a dynamic change in the task functionality is required.

See also

`OS_USE_TASK_ARG` (Section 4.1.61)
`TSKsetArg()` (Section 6.3.19)

6.3.7 TSKgetID

Synopsis

```
#include "Abassi.h"

TSK_t *TSKgetID(const char *Name);
```

Description

`TSKgetID()` is the component to use in order to obtain the task descriptor of a named task. One may desire to make use of this component to eliminate the sharing of variables (task descriptors) across multiple files.

Availability

`TSKgetID()` is only available when the build options `OS_NAMES` and the build option `OS_RUNTIME` are both non-zero.

Arguments

`Name`	Name of the task to obtain the descriptor. If `Name` is a `NULL` pointer or an empty string (`""`), then this component returns the running task's descriptor; this is the same behavior as using the `TSKmyID()` component.

Returns

This component returns the descriptor of the named task. If there are no tasks that have the name specified by the argument, a `NULL` pointer is returned. If two or more tasks have been created with the same name, the descriptor returned is that of last task created (time-wise).

Component type

Function

Options

Notes

The pre-defined name for the task Adam & Eve, the task associated to `main()`, is `"A&E"` and the name given to the Idle Task, that uses the function `IdleTask()`, is `"Idle"`.
Tasks created with the `TSK_STATIC()` component are not part of the search performed by `TSKgetID()`.

See also

`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`TSK_STATIC` (Section 6.3.2)
`TSKcreate()` (Section 6.3.5)
`TSKmyID()` (Section 6.3.16)

6.3.8 TSKgetPrio

Synopsis

```
#include "Abassi.h"

int TSKgetPrio(TSK_t *Task);
```

Description

`TSKgetPrio()` is the component used to obtain the current priority of a task. The priority returned is the current priority of the task, which is not necessary the priority assigned to it when it was created. The priority could differ because either the priority inversion protection and/or the task starvation protection is operating on the task.

Availability

Always.

Arguments

`Task`	Descriptor of the task to retrieve the current priority value.

Returns

The numerical value of the task's priority.

Component type

Data access

Options

Notes

The priority of the running task can be obtained with:

Table 6-6 Retrieving the priority of the running task

```
Priority = TSKgetPrio(TSKmyID());
```

See also

`TSKmyID()` (Section 6.3.16)
`TSKgetID()` (Section 6.3.7)
`TSKsetPrio()` (Section 6.3.20)

6.3.9 TSKgetRR

Synopsis

```
#include "Abassi.h"

unsigned int TSKgetRR(TSK_t *Task);
```

Description

`TSKgetRR()` is the component used to obtain the current maximum time slice duration a task can use when part of a round robin group of tasks. The returned value is in number of timer tick units.

Availability

`TSKgetRR()` is only available when the build options `OS_ROUND_ROBIN` and `OS_TIMER_US` are non-zero.

Arguments

`Task`	Descriptor of the task to retrieve the time slice duration value from.

Returns

The maximum number of timer ticks a task can use when involved in round robin.
When `OS_ROUND_ROBIN` is positive, the value returned is always `OS_ROUND_ROBIN` divided by `OS_TIMER_US`.

Component type

Data access

Options

Notes

If the build option `OS_ROUND_ROBIN` is positive, the value returned is always the value `OS_ROUND_ROBIN` divided by `OS_TIMER_US`.
If the build option `OS_ROUND_ROBIN` is negative, the value returned is negative `OS_ROUND_ROBIN` divided by `OS_TIMER_US`, if `TSKsetRR()` hasn't been applied on the task, otherwise it is the last value used with `TSKsetRR()` was applied on the task.

See also

`OS_ROUND_ROBIN` (Section 4.1.36)
`OS_TIMER_US` (Section 4.1.59)
`TSKsetRR()` (Section 6.3.21)

6.3.10 TSKgetStrvPrio

Synopsis

```
#include "Abassi.h"

int TSKgetStrvPrio(TSK_t * Task);
```

Description

`TSKgetStrvPrio()` is the component used to obtain the starvation priority threshold used by the task specified in the argument. The starvation priority threshold is the priority numerical value at which a task under starvation protection will stop getting its running priority level increased.

Availability

`TSKgetStrvPrio()` is only available when the build option `OS_STARVE_WAIT_MAX` is non-zero.

Arguments

`Task`	Descriptor of the task to retrieve the starvation priority threshold.

Returns

Starvation priority threshold the task uses when it goes under starvation protection.
When `OS_STARVE_PRIO` is positive, the value returned is always `OS_STARVE_PRIO`.

Component type

Data Access

Options

Notes

When the build option `OS_STARVE_PRIO` is positive, the value returned is always the value `OS_STARVE_PRIO` is set to.
If the build option `OS_STARVE_PRIO` is negative, the value returned is negative `OS_STARVE_PRIO` if `TSKsetStrvPrio()` hasn't been applied on the task, otherwise it's the last value used with `TSKsetStrvPrio()` when applied on the task.

See also

`OS_STARVE_PRIO` (Section 4.1.41)
`OS_STARVE_WAIT_MAX` (Section 4.1.43)
`TSKsetStrvPrio` (Section 6.3.22)

6.3.11 TSKgetStrvRunMax

Synopsis

```
#include "Abassi.h"

int TSKgetStrvRunMax(TSK_t *Task);
```

Description

`TSKgetStrvRunMax()` is the component used to obtain the maximum running time a task can run when under the starvation protection mechanism. The value reported is in number of timer tick units.

Availability

`TSKgetStrvRunMax()` is only available when the build option `OS_STARVE_WAIT_MAX` is non-zero.

Arguments

`Task`	Descriptor of the task to retrieve the maximum run time when under starvation protection.

Returns

The maximum number of timer ticks a task can use when it has its priority level increased by to the task starvation protection mechanism.
When `OS_STARVE_RUN_MAX` is positive, the value returned is always `OS_STARVE_RUN_MAX`.

Component type

Data access

Options

Notes

When the build option `OS_STARVE_RUN_MAX` is positive, the value returned is always the value `OS_STARVE_RUN_MAX` is set to.
If the build option `OS_STARVE_RUN_MAX` is negative, the value returned is negative `OS_STARVE_RUN_MAX` if `TSKsetStrvRunMax()` hasn't been applied on the task, otherwise it's the last value used with `TSKsetStrvRunMax()` when applied on the task.

See also

`OS_STARVE_RUN_MAX` (Section 4.1.42)
`OS_STARVE_WAIT_MAX` (Section 4.1.43)
`TSKsetStrvRunMax` (Section 6.3.23)

6.3.12 TSKgetStrvWaitMax

Synopsis

```
#include "Abassi.h"

int TSKgetStrvWaitMax(TSK_t *Task);
```

Description

`TSKgetStrvWaitMax()` is the component used to obtain the maximum time a task remains at the same priority when supervised by the starvation protection mechanism. If the task has not run for the required run time duration, the priority level of the task gets increased after the maximum waiting time. The value reported is in number of timer tick units.

Availability

`TSKgetStrvWaitMax()` is only available when the build option `OS_STARVE_WAIT_MAX` is non-zero.

Arguments

`Task`	Descriptor of the task to retrieve the maximum wait time when under starvation protection.

Returns

The maximum number of timer ticks a task remains at the same priority level when supervised by starvation protection mechanism.
When `OS_STARVE_WAIT_MAX` is positive, the value returned is always `OS_STARVE_WAIT_MAX`.

Component type

Data access

Notes

When the build option `OS_STARVE_WAIT_MAX` is positive, the value returned is always the value `OS_STARVE_WAIT_MAX` is set to.
If the build option `OS_STARVE_WAIT_MAX` is negative, the value returned is negative `OS_STARVE_WAIT_MAX` if `TSKsetStrvWaitMax()` hasn't been applied on the task, otherwise it's the last value used with `TSKsetStrvWaitMax()` when applied on the task.

See also

`OS_STARVE_WAIT_MAX` (Section 4.1.43)
`TSKsetStrvWaitMax` (Section 6.3.24)

6.3.13 TSKisBlk

Synopsis

```
#include "Abassi.h"

int TSKisBlk(TSK_t *Task);
```

Description

`TSKisBlk()` reports if a task is currently in the blocked state or not.

Availability

Always

Arguments

`Task`	Descriptor of the task to obtain information on its current state.

Returns

`0`	The task is not in the blocked state
`non-0`	The task is in the blocked state

Component type

Macro (unsafe)

Options

Notes

When the component `TSKsuspend()` (if `OS_TSK_SUSPEND` is non-zero) is applied on a task, this task will remain in the blocked state until it becomes running and then it will get immediately suspended. So until the task ran, the state of the task reported will be the blocked state, not the suspended state.

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKisRdy()` (Section 6.3.14)
`TSKisSusp()` (Section 6.3.15)
`TSKstate()` (Section 6.3.26)
`TSKsuspend()` (Section 6.3.29)

6.3.14 TSKisRdy

Synopsis

```
#include "Abassi.h"

int TSKisRdy(TSK_t *Task);
```

Description

`TSKisRdy()` reports if a task is currently in the ready to run state or not.

Availability

Always

Arguments

`Task`	Descriptor of the task to obtain information on its current state.

Returns

`0`	The task is not in the ready to run state
`non-0`	The task is in the ready to run state

Component type

Macro (safe)

Options

Notes

When the component `TSKsuspend()` (if `OS_TASK_SUSPEND` is non zero) is applied on a task that locks one or more mutexes, this task will remain in the ready to run / running state until it does not lock any mutexes, and will then immediately get suspended. So until the task releases all the locks it applies on mutexes, the state of the task reported will be the ready to run / running state.

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKisBlk()` (Section 6.3.13)
`TSKisSusp()` (Section 6.3.15)
`TSKstate()` (Section 6.3.26)
`TSKsuspend)` (Section 6.3.29)

6.3.15 TSKisSusp

Synopsis

```
#include “Abassi.h”

int TSKisSusp(TSK_t *Task);
```

Description

`TSKisSusp()` reports if a task is currently suspended or not.

Availability

Always

Arguments

`Task`	Descriptor of the task to obtain information on its current state.

Returns

`0`	The task is not suspended
`non-0`	The task is suspended

Component type

Macro (unsafe)

Options

Notes

When the component `TSKsuspend()` (if `OS_TASK_SUSPEND` is non zero) is applied on a task that locks one or more mutexes, this task will remain in the ready to run / running state until it does not lock any mutexes, and will then immediately get suspended. So until the task releases all the locks it applies on mutexes, the state of the task reported will be the ready to run / running state.

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKisBlk()` (Section 6.3.13)
`TSKisRdy()` (Section 6.3.14)
`TSKstate()` (Section 6.3.26)
`TSKsuspend)` (Section 6.3.29)

6.3.16 TSKmyID

Synopsis

```
#include "Abassi.h"

TSK_t *TSKmyID(void);
```

Description

`TSKmyID()` is the component used to obtain the pointer to the descriptor of the running task.

Availability

Always.

Arguments

`void`

Returns

This component returns the descriptor of the running task.

Component type

Data access
- Meaningless in an interrupt -

Options

Notes

Contrary to the component `TSKgetID()`, this component does not rely on task names (if `OS_NAMES` is non-zero), which means it is always available.

See also

`OS_NAMES` (Section 4.1.28)
`TSKgetID()` (Section 6.3.7)

6.3.17 TSKresume

Synopsis

```
#include "Abassi.h"

void TSKresume(TSK_t *Task);
```

Description

In order to change the state of a task from the suspended state to the running or ready to run state, `TSKresume()` is the component to use.

Availability

Always

Arguments

`Task`	Descriptor of the task to resume.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

When tasks are created in the suspended state, with `TSK_STATIC()/TSK_SETUP()` or `TSKcreate()`, `TSKresume()` must be used on these tasks before they can operate or before any other components are used on them; the same applies when a task is in the suspended state because `TSKsuspend()` or `TSkselfSusp()` was applied on it.
If a task is not suspended, or is not in the process of getting suspended, using `TSKresume()` on it will do nothing. Applying multiple `TSKresume()` on a task does not create cumulative action. This means no matter how many `TSKresume()` are applied on a task, the effect remains the same as if `TSKresume()` was used only once.
When a task that was required to get suspended is still ready to run / running because it locks one or more mutexes, applying `TSKresume()` on such a task cancels the suspension in progress.

See also

`TSKcreate()` (Section 6.3.5)
`TSKselfSusp()` (Section 6.3.18)
`TSKstate()` (Section 6.3.26)
`TSKsuspend()` (Section 6.3.29)

6.3.18 TSKselfSusp

Synopsis

```
#include "Abassi.h"

void TSKselfSusp(void);
```

Description

`TSKselfSusp()` is a component that changes the state of the running task from the running state to the suspended state. Contrary to `TSKsuspend()`, `TSKselfSusp()` immediately suspends the running task. See **Notes** below for the pitfalls.

Availability

Always.

Arguments

`void`

Returns

`void`

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

Notes

`TSKselfSusp()` does not use the same internal mechanism to suspend a task as the component `TSKsuspend()` uses (if `OS_TASK_SUSPEND` is non-zero). The later suspends a task only when all the mutexes the task locks have been released. In the case of `TSKselfSusp()`, the running task gets immediately suspended. Therefore, when `TSKselfSusp()` is used, all mutexes locked by the running task should be unlocked before using `TSKselfSusp()`. Not doing so will block other tasks when they try to lock the mutex(es) owned by the suspended task. These tasks will remain blocked on the mutex(es) until the suspended task is resumed and then unlocks the mutex(es).

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKresume()` (Section 6.3.17)
`TSKsuspend()` (Section 6.3.29)

6.3.19 TSKsetArg

Synopsis

```
#include "Abassi.h"

void TSKsetArg(TSK_t *Task, void *Ptr);
```

Description

`TSKsetArg()` is the component to use to forward arguments, indicated by `Ptr`, to the task, specified by `Task`. Using a forwarding approach, instead of passing argument to the task function, was first retained to minimize the complexity of passing arguments to a task, but also to allow dynamic changes of a task argument. This could be useful, for example, when suspending/resuming a task in order to modify its behavior.

Availability

`TSKsetArg()` is only available when the build option `OS_TASK_ARG` is non-zero.

Arguments

`Task`	Descriptor of the task to assign the argument.
`Ptr`	Pointer to any type of argument to pass to the target task.

Returns

`void`

Component type

Data access

Options

Notes

Any data type can be forwarded to a task using this component. One has to remember it is the pointer that is forwarded, not the contents. Therefore the contents should not be modified after using `TSKsetArg(),` unless a dynamic change in the task functionality is desired.

See also

`OS_TASK_ARG` (Section 4.1.61)
`TSKgetArg()` (Section 6.3.6)

6.3.20 TSKsetPrio

Synopsis

```
#include "Abassi.h"

void TSKsetPrio(TSK_t *Task, int Prio);
```

Description

`TSKsetPrio()` is a component used to dynamically change the priority of a task.

Availability

`TSKsetPrio()` is only available when the build option `OS_PRIO_CHANGE` is non-zero.

Arguments

`Task`	Descriptor of the task to set the priority.
`Prio`	New priority of the task, a numerical value of 0 is the highest priority.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

The highest priority of a task in an application is always 0. The lowest priority value is defined by the build option `OS_PRIO_MIN`. Therefore, the range of values for the argument `Prio` must be bounded between 0 and `OS_PRIO_MIN`. Setting `Prio` to any values outside this range will quite likely crash the Abassi RTOS.
If the priority of a task currently involved in a mutex priority inversion protection is modified, the change of priority is immediately applied, possibly modifying the priority of the task locking the mutex.
If the priority of a task that is currently supervised by the starvation protection is modified, this change of priority, no matter what is the new priority level, triggers the starvation protection mechanism to momentary stop monitoring that task. The task is put back into the starvation queue, if the new priority level is below the starvation priority threshold.

See also

`OS_PRIO_CHANGE` (Section 4.1.33)
`OS_PRIO_MIN` (Section 4.1.34)
`TSKgetPrio()` (Section 6.3.8)

6.3.21 TSKsetRR

Synopsis

```
#include "Abassi.h"

void TSKsetRR(TSK_t *Task, unsigned int Time);
```

Description

`TSKsetRR()` is the component used to dynamically change the round-robin time slice duration allocated to a task.

Availability

`TSKsetRR()`is only available when the build option `OS_TIMER_US` is non-zero, and the build option `OS_ROUND_ROBIN` is negative.

Arguments

`Task`	Task to set a new round-robin time slice value.
`Time`	Time slice duration, in number of timer ticks. If `Time` is zero, the task runs until blocked, or preempted, or relinquishes the CPU through the `TSKyield()` component.

Returns

`void`

Component type

Data access

Options

Notes

Upon creation, all tasks are allocated a time slice duration, indicated in microseconds, which is the absolute value of `OS_ROUND_ROBIN`. When the component `TSKsetRR()` is used on a task, it reconfigure the round robin time slice duration allocated to the task. Note that `TSKsetRR()` requires the time in number of timer ticks, and not in microseconds, as the units for the time slice duration. The time conversion components can be used to convert seconds or fraction into number of timer ticks.
When the component `TSKsetRR()` is applied by and to the running task, and the running task is currently involved in round robin, the new value takes effect when the next timer tick interrupt occurs.
When the argument `Time` has a value of zero, if the task is preempted, it will not relinquish the CPU to another task at the same priority when the preemption ends. To relinquish the CPU to another task at the same priority, the task must become blocked or voluntarily relinquish the CPU through the `TSKyield()` component.

See also

`OS_ROUND_ROBIN` (Section 4.1.36)
`OS_TIMER_US` (Section 4.1.59)
`OS_MS_TO_TICK` (Section 6.7.17.3)
`OS_SEC_TO_TICK` (Section 6.7.17.5)
`TSKyield` (Section 6.3.32)

6.3.22 TSKsetStrvPrio

Synopsis

```
#include "Abassi.h"

void TSKsetStrvPrio(TSK_t *Task, int Prio);
```

Description

`TSKsetStrvPrio()` is the component that sets the starvation priority threshold used by the task specified in the argument. The starvation priority threshold is the priority at which a task under starvation protection will stop getting its running priority level increased.

Availability

`TSKsetStrvPrio()` is only available when the build option `OS_STARVE_WAIT_MAX` is set to a non-zero value and the build option `OS_STARVE_PRIO` is set to a negative value.

Arguments

`Task`	Descriptor of the task to set a new starvation priority threshold.
`Prio`	Numerical value of the new priority threshold of the task. If `Prio` is set to `OS_PRIO_MIN` or higher (a higher numerical value is a lower priority), the task starvation protection is disabled on this task.

Returns

`void`

Component type

Data Access

Options

Notes

The highest priority of a task in an application is always 0. The lowest priority value is defined by the build option `OS_PRIO_MIN`. With this component, there is no restriction on what the value the argument `Prio` can be set to. If the numerical value of `Prio` is greater or equal to the value of the regular run-time priority of the task specified in the argument `Task`, it informs the kernel to disable the task starvation protection for the task. As a simple guideline, if it is desired to disable the task starvation protection on a task, setting its starvation priority threshold to `OS_PRIO_MIN` is the best way to achieve this.

If the `TSKsetStrvPrio()` component is used on the task which is the task currently being supervised by the task starvation mechanism, three things can happen:

- The new priority level is the same or is above the current priority threshold: the operation of the starvation protection mechanism remain the same but, with the new priority threshold.
- The new priority level is below the current priority threshold and above the run-time priority of the task: the task remains at its currently promoted priority until it succeed at reaching the running state and is removed from supervision.
- The new priority level is equal or below the current run-time starvation priority of task: the task will be removed from supervision when it has expired the maximum waiting time at the current promoted priority.

See also

`OS_PRIO_MIN` (Section 4.1.34)
`OS_STARVE_PRIO` (Section 4.1.41)
`OS_STARVE_WAIT_MAX` (Section 4.1.43)

6.3.23 TSKsetStrvRunMax

Synopsis

```
#include "Abassi.h"

void TSKsetStrvRunMax(TSK_t *Task, int Time);
```

Description

`TSKsetStrvRunMax()` is the component that sets the maximum running time when a task is running due to the starvation protection mechanism. The task selected to set the maximum run-time value is specified by the argument `Task` and the maximum starvation run-time is specified by the argument `Time`. The numerical value to set `Time` must be specified in number of timer ticks unit.

Availability

`TSKsetStrvRunMax()` is only available when the build option `OS_STARVE_WAIT_MAX` is non-zero and the build option `OS_STARVE_RUN_MAX` is negative.

Arguments

`Task`	Descriptor of the task to set the maximum starvation run time when under starvation protection.
`Time`	Maximum starvation protection run time, indicated in number of timer ticks units. If `Time` value is zero or negative, is it is the same as setting `Time` to 1.

Returns

`void`

Component type

Data access

Options

Notes

If the `TSKsetStrvRunMax()` component is used on the task that is the task currently being supervised by the task starvation mechanism, three things can happen:

- The task has not ran yet since it is has been under starvation protection: when the task will run, it will use the new value set with `TSKsetStrvRunMax()`.
- The task has been running, but for less time than the new value: the task will be running according the previous setting for the maximum run time.
- The task has been running for a longer time than the new value (the new value is less than it was before): the task will be running according the previous setting for the maximum run time.

See also

`OS_STARVE_RUN_MAX` (Section 4.1.42)
`OS_STARVE_WAIT_MAX` (Section 4.1.43)
`TSKgetStrvRunMax` (Section 6.3.11)

6.3.24 TSKsetStrvWaitMax

Synopsis

```
#include "Abassi.h"

void TSKsetStrvWaitMax(TSK_t *Task, int Time);
```

Description

`TSKsetStrvWaitMax()` is the component to use to set the maximum time a task remains at the same priority level when it is under the starvation protection mechanism. The task selected to set the maximum starvation wait time value is specified by the argument `Task`, and the time by the argument `Time`. The value to set `Time` to must represent a number of timer ticks units.

Availability

`TSKsetStrvWaitMax()` is only available when the build option `OS_STARVE_WAIT_MAX` is negative.

Arguments

`Task`	Descriptor of the task to set the maximum wait time when it is under starvation protection.
`Time`	Maximum starvation protection wait time, indicated in number of timer ticks units. If `Time` value is zero or negative, is it is the same as setting `Time` to 1.

Returns

`void`

Component type

Data access

Options

Notes

If the `TSKsetStrvWaitMax()` component is used on the task which is the task currently being supervised by the task starvation mechanism, then it will always complete the wait time it was originally programmed to. Then, when a priority level promotion occurs, it will start using the new wait time settings.

See also

`OS_STARVE_WAIT_MAX` (Section 4.1.43)

6.3.25 TSKsleep

Synopsis

```
#include "Abassi.h"

int TSKsleep(int Time);
```

Description

`TSKsleep()` is a component that temporary blocks the running task for a fixed amount of time. It is possible to abort a task sleep in progress by using the `TSKtimeoutKill()` component.

Availability

`TSKsleep()` is only available if the build option `OS_TIMEOUT` and the build option `OS_TIMER_US` are both non-zero.

Arguments

`Time`	Number of timer ticks to force the task to remain in the blocked state. If `Time` is zero, it is a do nothing operation. If `Time` is negative, `TSKsleep()` is replaced by the component `TSKselfSusp()`.

Returns

`0`	The task was unblocked before reaching the total of `Time` timer ticks. This happens if the component `TSKtimeoutKill()` was applied on the task.
`non-0`	The task became normally unblocked after the requested count of timer ticks.

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

If the build option `OS_TIMEOUT` is zero, then when the argument `Time` is set to a positive value, `TSKsleep()` behaves the same as if the `Time` argument had been set to zero.
If the build option `OS_TIMEOUT` is a negative value, then when the argument `Time` is set to a positive value, `TSKsleep()` behaves the same as if the component `TSKselfSusp()` had been used.

Notes

When the argument `Time` is negative, in order to remain consistent with the meaning of a negative timeout as used in all other blocking services, `TSKsleep()` remaps to `TSKselfSusp()`, which then exhibits exactly the same pitfalls `TSKselfSusp()` has. Refer to the description of `TSKselfSusp()` to gain awareness .

See also

`OS_TIMEOUT` (Section 4.1.57)
`OS_TIMER_US` (Section 4.1.59)
`TSKselfSusp()` (Section 6.3.18)
`TSKtimeoutKill()` (Section 6.3.30)

6.3.26 TSKstate

Synopsis

```
#include "Abassi.h"

int TSKstate(TSK_t *Task);
```

Description

`TSKstate()` is a component that reports the current state of a task.

Availability

Always.

Arguments

`Task`	Descriptor of the task to obtain the operating state

Returns

`0`	The task is in the ready to run state
`Negative`	The task is in the suspended state
`Positive`	The task is in the blocked state

Component type

Macro definition (unsafe)

Options

Notes

When the component `TSKsuspend()` (when `OS_TASK_SUSPEND` is non-zero) is applied on a blocked task, this task will remain in the blocked state until it becomes running which will then get it immediately suspended. So until the task ran, the state of the task reported will be the blocked state.
When the component `TSKsuspend()`(when `OS_TASK_SUSPEND` is non-zero) is applied on a task that locks one or more mutexes, this task will remain in the ready to run / running state until it does not lock any mutexes, which will then get it immediately suspended. So until the task gets rid of all the locks it applies on mutexes, the state of the task reported will be the ready to run / running state.

See also

`TSKisBlk()` (Section 6.3.13)
`TSKisRdy()` (Section 6.3.14)
`TSKisSusp()` (Section 6.3.15)
`TSKsuspend()` (Section 6.3.29)

6.3.27 TSKstkFree

Synopsis

```
#include "Abassi.h"

int TSKstkFree(TSK_t *Task);
```

Description

This is the component used to obtain a measurement on the minimum space that has never been used on a task's stack.

.

Availability

Only available when `OS_STACK_CHECK` is defined and set to non-zero

Arguments

`Task`	Descriptor of the task to monitor

Returns

The amount of stack of the task `Task` that has never been used (in bytes)
-1 : for the Adam & Eve task when its stack cannot be monitored

Component type

Function

Options

Notes

For most ports, the stack of the Adam & Eve task cannot be monitored. There are two reasons for that. The first one is the base and the top of the stack associated to the function `main()` is not always available at run time. The second and most important is that Adam & Eve is a special task, as it is not created from the ground up. Due to this, the task's stack is already in use and it becomes a bit convoluted to try to fill the un-used part of the stack with the monitoring data.

.

See also

`OS_STACK_CHECK` (Section 4.1.40)
`TSKstkUsed()` (Section 6.3.28)

6.3.28 TSKstkUsed

Synopsis

```
#include "Abassi.h"

int TSKstkUsed(TSK_t *Task);
```

Description

This is the component used to obtain a measurement on the maximum space that has been used on a task's stack.
.

Availability

Only available when `OS_STACK_CHECK` is defined and set to non-zero

Arguments

`Task`	Descriptor of the task to monitor

Returns

The amount of stack of the task `Task` that has been used (in bytes)
-1 : for the Adam & Eve task when its stack cannot be monitored

Component type

Function

Options

Notes

For most ports, the stack of the Adam & Eve task cannot be monitored. There are two reasons for that. The first one is the base and the top of the stack associated to the function `main()` is not always available at run time. The second and most important is that Adam & Eve is a special task, as it is not created from the ground up. Due to this, the task's stack is already in use and it becomes a bit convoluted to try to fill the un-used part of the stack with the monitoring data.
.

See also

`OS_STACK_CHECK` (Section 4.1.40)
`TSKstkFree()` (Section 6.3.27)

6.3.29 TSKsuspend

Synopsis

```
#include "Abassi.h"

void TSKsuspend(TSK_t *Task);
```

Description

This is the component used to put a task in the suspended state. The task to suspend may run for a while if it owns/locks mutexes. Only when all mutexes owned by the task are released does the suspending occurs.
If the task to suspend is in the blocked state, the task will have to reach the running state and then will immediately get suspended.

Availability

Only available when `OS_TASK_SUSPEND` is set to non-zero

Arguments

`Task`	Descriptor of the task to suspend.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

A task must become running before reaching the suspended state. Therefore if this component is applied on a task in the blocked state, the task will remain in the blocked state until it gets unblocked, then it will then immediately go into the suspended state.
As explained in **Description**, a task suspension triggered by `TSKsuspend()` happens only when the task to suspend has released all locks it has on mutexes. This restriction was added as a safety mechanism because if a task goes into the suspended state while it locks one or more mutexes, then other tasks in the application that try to get a lock on these mutexes will effectively becomes suspended due to the deadlock. The component `TSKselfSusp()` does not provide this protection.
Another safeguard applied to `TSKsuspend()` is related to the interrupts. When the interrupts are disabled, a task will not get suspended as long as the interrupts are disabled. This is added to protect against a task disabling for a short time the interrupt and while the interrupts are disabled, it gets forced by another one to be suspended. If this safety was not present, the interrupts would remain disabled until the task gets resumed.

See also

`OS_TASK_SUSPEND` (Section 4.1.54)
`TSKresume()` (Section 6.3.17)
`TSKselfSusp()` (Section 6.3.18)

6.3.30 TSKtimeoutKill

Synopsis

```
#include "Abassi.h"

void TSKtimeoutKill(TSK_t *Task);
```

Description

`TSKtimeoutKill()` is the component to use to prematurely unblock a task that has been put in the blocked state through the component `TSKsleep()`, or if the task is blocked with a timeout, waiting on a semaphore, a mutex lock, or reading/writing a mailbox.

Availability

`TSKtimeoutKill()` is only available if the build option `OS_TIMEOUT` and `OS_TIMER_US` are both non-zero

Arguments

`Task`	Descriptor of the task to unblock from an expiry timeout.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

Nothing happens if the task is not currently blocked through the use of the `TSKsleep()` component, or blocked with timeout through the use of the `SEMwait()`, `SEMwaitBin()`, `MTXlock()`, `EVTwait()`, `MBXput()`, or `MBXget()` components. This applies to the running task, a ready to run task, or a task suspended by `TSKsuspend()`.
If a task is blocked on any of the previous components with a negative timeout (meaning no expiry timeout), then `TSKtimeoutKill()`has absolutely no effect.

See also

`OS_TIMEOUT` (Section 4.1.57)
`OS_TIMER_US` (Section 4.1.59)
`EVTwait()` (Section 6.6.8)
`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)
`MTXlock()` (Section ·)
`SEMwait()` (Section 6.4.11)
`SEMwaitBin()` (Section 6.4.12)
`TSKsleep()` (Section 6.3.25)
`TSKtout()` (Section 6.3.31)

6.3.31 TSKtout

Synopsis

```
#include "Abassi.h"

void TSKtout(TSK_t *Task, int TimeOut);
```

Description

`TSKtout()` is the component to use to modify the timeout of a task blocked with a timeout, waiting on a semaphore, a mutex lock, or reading/writing a mailbox. When TSKtout() is used, the current time left in the timeout is replaced by the value provided by the argument `TimeOut`.

Availability

`TSKtout()` is only available if the build option `OS_TIMEOUT` and `OS_TIMER_US` are both non-zero

Arguments

`Task`	Descriptor of the task to unblock from an expiry timeout.
`TimeOut`	New expiry timeout.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

Nothing happens if the task is not currently blocked through the use of the `TSKsleep()` component, or blocked with timeout through the use of the `SEMwait()`, `SEMwaitBin()`, `MTXlock()`, `EVTwait()`, `MBXput()`, or `MBXget()` components. This applies to the running task, a ready to run task, or a task suspended by `TSKsuspend()`.
If a task is blocked on any of the previous components with a negative timeout (meaning no expiry timeout), then `TSKtout()`has absolutely no effect.

See also

`OS_TIMEOUT` (Section 4.1.57)
`OS_TIMER_US` (Section 4.1.59)
`EVTwait()` (Section 6.6.8)
`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)
`MTXlock()` (Section ·)
`SEMwait()` (Section 6.4.11)
`SEMwaitBin()` (Section 6.4.12)
`TSKsleep()` (Section 6.3.25)
`TSKtimeoutKill()` (Section 6.3.30)

6.3.32 TSKyield

Synopsis

```
#include "Abassi.h"

void TSKyield(void);
```

Description

`TSKyield()` is the component to use by the running task to yield the CPU to another ready to run task. The running task can yield the CPU under only 2 scenarios:

- The kernel is operating in the cooperative mode with the build option `OS_COOPERATIVE` set to a non-zero value. Under these conditions, the next task to run is the ready to run task with the highest or same priority level as the task yielding the CPU.
- The kernel is not in the cooperative mode. Under these conditions, the next task ready to run, which is at the same priority as the yielding task, will get the CPU.

Availability

Arguments

None

Returns

`void`

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

Notes

In the cooperative mode, the CPU is not relinquished when the component `TSKyield()` is used and there are no ready to run tasks that have a priority level equal or greater than the running task.
When not in the cooperative mode, the CPU is not relinquished if there are no other tasks ready to run at the same priority, or if the build option `OS_PRIO_SAME` is set to zero

See also

`OS_COOPERATIVE` (Section 4.1.4)
`OS_PRIO_SAME` (Section 4.1.35)

6.3.33 Examples

6.3.33.1 Static task

This example shows a snippet of code that creates a task at compile/link time. The variable name for the task descriptor is `Task1`, and its runtime name is `"Task #1"`. The run-time priority value is 1 (one level below the maximum priority level), and a stack of 512 bytes is allocated to the task, which operates as the function `FctTask1()`.

Table 6-7 Static Task definition example

```
#include "Abassi.h"

STATIC_TASK(Task1, "Task #1", 1, 512, FctTask1);

main()
{
   OSstart();                          /* RTOS set-up                                 */

   TSK_SETUP(Task1, 0);                /* TSK_SETUP must be used on all static tasks  */

   TSKresume(Task1);                   /* This task was created suspended             */

   …
```

6.3.33.2 OS_IDLE_STACK set to 0

This example shows code where the build option `OS_IDLE_STACK` is set to 0. This means `OSstart()` does not deal with the creation of the Idle Task, so the function `IdleTask()` does not need to be supplied by the application. Instead, the Adam & Eve task operating with `main()`, which is the task with the highest priority as set-up by `OSstart()`, gets its priority changed to `OS_PRIO_MIN`, which effectively convert it into the Idle Task. All there is to do is to use the `TSKsetPrio()` component for the priority change. When `TSKsetPrio()` is used, the task associated with `main()` is set to the lowest priority in the application.

Table 6-8 `OS_IDLE_STACK` set to 0 example

```
#include "Abassi.h"

main()
{
   OSstart();

   …                                      /* Perform any initialization required      */

   TSKcreate("Fct Hi", 3, 1024, &FctHi, 1); /* Create a task in ready to run state   */

   TSKsetPrio(TSKmyID(), OS_PRIO_MIN);    /* Is now operating at lowest priority      */

   …
```

6.4 Semaphore Components

This section describes all components related to the semaphores. The semaphore components described in the following sub-sections are:

Table 6-9 Semaphore Component list

Section	Name	Description
6.4.2	`SEM_STATIC`	Creation of a semaphore at compile / link time
6.4.3	`SEMabort`	Unblock all tasks blocked on a semaphore
6.4.4	`SEMnotFCFS`	Set a semaphore to operate in the *Priority* mode
6.4.5	`SEMopen`	Create a semaphore / obtain the descriptor of a semaphore
6.4.6	`SEMopenFCFS`	Create a semaphore to operate in *First Come First Served* mode Obtain the descriptor of a semaphore
6.4.7	`SEMpost`	Post a semaphore
6.4.8	`SEMpostAll`	Post a semaphore multiple times to unblock all tasks
6.4.9	`SEMreset`	Remove the excess count of a semaphore
6.4.10	`SEMsetFCFS`	Set a semaphore to operate in the *First Come First Served* mode
6.4.11	`SEMwait`	Wait on (Acquire) a semaphore
6.4.12	`SEMwaitBin`	Wait on (Acquire) a binary semaphore

6.4.1 Description

The semaphore service is the heart of the Abassi RTOS as semaphores are the sole blocking mechanism internally used. Semaphores are always counting semaphores, but they can operate as binary semaphore upon waiting. Also, by default, semaphores operate in a *Priority* mode, which means that when multiple tasks are blocked on the same semaphore, the task with the highest priority level is the first to get unblocked upon posting of the semaphore. Although by default they operate in *Priority* mode, individual semaphores can be optionally set to operate in a *First Come First Served* mode, where the first task blocked (time-wise) by a semaphore is the first task to get unblocked upon posting of the semaphore. A semaphore operating in the *First Come First Served* mode can be returned to the *Priority* mode at any time and vice-versa.

6.4.2 SEM_STATIC

Synopsis

```
#include "Abassi.h"

SEM_STATIC(VarName, SemName);
```

Description

`SEM_STATIC()` is a special component that creates a semaphore and initializes its descriptor. It is a macro definition creating a static object, so none of the arguments has a real data type. The semaphore is not created/initialized at run time; everything is done at compile/link time.

Availability

Always.

Arguments

`VarName`	Name of the variable holding the pointer to the semaphore descriptor to create / initialize. This is a variable name therefore do not put double quotes around the name.
`SemName`	Semaphore name. This is not the variable name, it is the name attached to the semaphore. As it is a "C" string, the double quotes around the name are required. `G_OSnoName` , and not `NULL`, should be used for an unnamed semaphore.

Returns

N/A

Component type

Macro (safe)

Options

If the build option `OS_NAMES` is set to a value of zero, the argument `SemName` is ignored but must still be supplied.

Notes

If one or more semaphores are created using the `SEM_STATIC()` component, it may not be possible to restart the application without reloading the binary image on the processor. This situation happens if the compiler (or compiler configuration) does not reload the initialized data upon start-up.
A semaphore created and initialized with `SEM_STATIC()` will not be part of the search done with `SEMopen()` or `SEMopenFCF()`.
A semaphore created with this component is always created to operate in the *Priority* mode. If the build option `OS_FCFS` is non-zero, and a semaphore created with `SEM_STATIC()` is targeted to operate in the *First Come First Served* mode, use the component `SEMsetFCFS()` on the semaphore in `main()`, once the component `OSstart()` has been used.

See also

`OS_FCFS` (Section 4.1.6)
`OS_NAMES` (Section 4.1.28)
`SEMopen()` (Section 6.4.5)
`SEMopenFCFS()` (Section 6.4.6)
`SEMsetFCFS()` (Section 6.4.10)
`G_OSnoName` (Section 6.14.2)

6.4.3 SEMabort

Synopsis

```
#include "Abassi.h"

void SEMabort(SEM_t *Sema);
```

Description

`SEMabort()` is a component that unblocks all tasks that are blocked on the semaphore `Sema`. This is alike applying the component `TSKtimeoutKill()` on all the tasks blocked on the semaphore `Sema` with the extra capability to also unblocked tasks blocked with an infinite timeout. `SEMabort()` has no effect on a task blocked on a group the semaphore is attached to.

Availability

`SEMabort()` is only available when the build option `OS_WAIT_ABORT` is defined and non-zero.

Arguments

`Sema`	Descriptor of the semaphore to unblock all the tasks that are blocked on.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

All tasks that blocked on the `SEMwait()` or `SEMwaitBin()` components are unblocked when the component `SEMabort()` is used on the semaphore. The `SEMwait()` or `SEMwaitBin()` components then return a non-zero value to indicate a timeout occurred, even if the wait time requested was infinite.

`SEMabort()` applied on a semaphore that is attached to a group **<u>will not</u>** unblock a task waiting on that group. The reason is the task is blocked on the group, not blocked on the semaphore. If there are one or more tasks blocked on the same semaphore (non-group blocking is authorized on a semaphore already attached to a group), then these tasks will get unblocked.

If there are no tasks blocked on the semaphore `Sema`, the use of `SEMabort()` on `Sema` has no effects.

See also

`OS_WAIT_ABORT` (Section 4.1.62)
`SEMwait()` (Section 6.4.11)

`SEMwaitBin()` (Section 6.4.12)

6.4.4 SEMnotFCFS

Synopsis

```
#include "Abassi.h"

void SEMnotFCFS(SEM_t *Sema);
```

Description

`SEMnotFCFS()` is a component that configures an existing semaphore to operate in the *Priority* mode instead of the *First Come First Served* mode. When a semaphore operating in *Priority* mode is posted and tasks are blocked on it, the highest priority task blocked is always the next task that will be unblocked first.

Availability

`SEMnotFCFS()` is only available when the build option `OS_FCFS` is non-zero.

Arguments

`Sema`	Descriptor of the semaphore to configure into the *Priority* mode.

Returns

`void`

Component type

Data access

Options

Notes

If the semaphore is already operating in the *Priority* mode, using this component has no effect on the semaphore.
If the semaphore is operating in the *First Come First Served* mode, using this component will not re-order the tasks that are currently blocked on the semaphore. Newly blocked tasks will be inserted in a *Priority* ordering amongst the already *First Come First Served* ordered blocked tasks. This means there may be a transient phase before the semaphore truly operates in a *Priority* mode.

See also

`OS_FCFS` (Section 4.1.6)
`SEMopenFCFS()` (Section 6.4.6)
`SEMsetFCFS()` (Section 6.4.10)

6.4.5 SEMopen

Synopsis

```
#include "Abassi.h"

SEM_t *SEMopen(const char *Name);
```

Description

`SEMopen()` is the component to use to create a semaphore, and is also the component to use to obtain the descriptor of an already existing semaphore (when `OS_NAMES` is non-zero). All semaphores created with `SEMopen()` operate upon creation in the *Priority* mode.

Availability

`SEMopen()` is only available when the build option `OS_RUNTIME` is non-zero.

Arguments

`Name`	Name of the semaphore to create or to obtain the descriptor of.

Returns

Descriptor of the semaphore

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build where `OS_NAMES` is zero, all semaphores are unnamed and every use of `SEMopen()` creates a new semaphore.
If the build option `OS_NAMES` is non-zero, then `SEMopen()` will either return the descriptor of an existing semaphore (previously created with `SEMopen()` or `SEMopenFCFS()`), or when no semaphore with the specified name exists, it will create a new semaphore. This approach makes the creation and opening of semaphores run-time safe. If that feature was not part of the `SEMopen()` component, it would be imperative to either create the semaphore immediately at start-up or to guarantee the first task (using the semaphore) to reach the running state is the one creating the semaphore. With the run-time safe feature, it does not matter which task is the first to open/create the semaphore.
If the build option `OS_STATIC_SEM` is non-zero, the semaphore descriptor uses memory that was allocated/reserved at compile/link time instead of memory dynamically allocated at run-time.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all unnamed semaphores. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.
Be aware, if the build option `OS_FCFS` is non-zero, when the semaphore already exists, there is no guarantee the semaphore is operating in a *Priority* mode as it may have been created with `SEMopenFCFS()` or it may have been set to operate in *First Come First Served* mode with `SEMsetFCFS()`.
At any time a semaphore operating in the *Priority* mode can be modified to operate in the *First Come First Served* mode by using the `SEMsetFCFS()` component when the build option `OS_FCFS` is non-zero.
Semaphores created with the `SEM_STATIC()` component are not part of the search performed by `SEMopen()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`SEMopenFCFS()` (Section 6.4.6)
`SEMsetFCFS()` (Section 6.4.10)

6.4.6 SEMopenFCFS

Synopsis

```
#include "Abassi.h"

SEM_t *SEMopenFCFS(const char *Name);
```

Description

`SEMopenFCFS()` is the component to use to create a semaphore operating in a *First Come First Served* mode, and is also the component to use to obtain the descriptor of an already existing semaphore (when `OS_NAMES` is non-zero).

Availability

`SEMopenFCFS()` is only available when the build options `OS_RUNTIME` and `OS_FCFS` are both non-zero.

Arguments

`Name`	Name of the semaphore to create or to obtain the descriptor of.

Returns

Descriptor of the semaphore

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build where `OS_NAMES` is zero, all semaphores are unnamed and every use of `SEMopenFCFS()` creates a new semaphore.
If the build option `OS_NAMES` is non-zero, then `SEMopenFCFS()` will either return the descriptor of an existing semaphore (previously created with `SEMopen()` or `SEMopenFCFS()`), or when no semaphore with the specified name exists, it will create a new semaphore. This approach makes the creation and opening of semaphores run-time safe. If that feature was not part of the `SEMopenFCFS()` component, it would be imperative to either create the semaphore immediately at start-up or to guarantee the first task (using the semaphore) to reach the running state is the one creating the semaphore. With the run-time safe feature, it does not matter which task is the first to use the semaphore.
If the build option `OS_STATIC_SEM` is non-zero, the semaphore descriptor uses memory that was allocated/reserved at compile/link time instead of memory dynamically allocated at run-time.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all unnamed semaphores. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.
Be aware, when the semaphore already exists, there is no guarantee the semaphore is operating in a *First Come First Served* mode as it may have been created with `SEMopen()`or it may have been set to operate in *Priority* mode with `SEMnotFCFS()`.
At any time a semaphore operating in the *First Come First Served* mode can be modified to operate in the *Priority* mode by using the `SEMnotFCFS()` component.
Semaphores created with the `SEM_STATIC()` component are not part of the search performed by `SEMopenFCFS()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`SEMnotFCFS()` (Section 6.4.4)
`SEMopen()` (Section 6.4.5)
`SEMsetFCFS()` (Section 6.4.10)

6.4.7 SEMpost

Synopsis

```
#include “Abassi.h”

void SEMpost(SEM_t *Sema);
```

Description

`SEMpost()` is the component to use to post a semaphore.

Availability

Always.

Arguments

`Sema`	Descriptor of the semaphore to post.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

Semaphores are internally always operating as counting semaphores, meaning each posting increments a count register, but binary semaphores are also supported. When a semaphore is posted, it is always operating as a counting semaphore. The difference between a counting and binary semaphore occurs only when waiting on the semaphore. A single use of `SEMwaitBin()` on a semaphore will zero the internal counter.
A safety mechanism exists in the Abassi RTOS to saturate the internal counter when it reaches the largest `int` value.

See also

`SEMpostAll()` (Section 6.4.8)
`SEMwait()` (Section 6.4.11)
`SEMwaitBin()` (Section 6.4.12)

6.4.8 SEMpostAll

Synopsis

```
#include "Abassi.h"

void SEMpostAll(SEM_t *Sema);
```

Description

`SEMpostAll()` is sort of a superset of `SEMpost()`. When `SEMpost()` is used and one or more tasks are blocked on `Sema`, only a single task gets unblocked. If `SEMpostAll()` is used instead, then all tasks blocked on `Sema` gets unblocked. When there are no tasks blocked on `Sema`, `SEMpostAll()` behaves exactly the same way as `SEMpost()`.

Availability

Available since 2019

Arguments

`Sema`	Descriptor of the semaphore to post one or multiple times

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

See also

`SEMpost()` (Section 6.4.7)
`SEMwait()` (Section 6.4.11)
`SEMwaitBin()` (Section 6.4.12)

6.4.9 SEMreset

Synopsis

```
#include "Abassi.h"

void SEMreset(SEM_t *Sema);
```

Description

`SEMreset()` is the component to use to remove the excess count of a semaphore.

Availability

Always.

Arguments

`Sema`	Descriptor of the semaphore to remove the excess count.

Returns

`void`

Component type

Atomic macro (unsafe)

Options

Notes

`SEMreset()` is used to remove the excess postings that could have been done on a semaphore. If there are no excess postings or if one or more tasks are blocked on the semaphore, `SEMreset`() does not change the semaphore count.

See also

`SEMpost()` (Section 6.4.7)

6.4.10 SEMsetFCFS

Synopsis

```
#include "Abassi.h"

void SEMsetFCFS(SEM_t *Sema);
```

Description

`SEMsetFCFS()` is the component that configures an existing semaphore to operate in a *First Come First Served* mode. When a semaphore operating in *First Come First Served* mode is posted and tasks are blocked on it, the oldest task that was blocked (time-wise) becomes running/ready to run.

Availability

`SEMsetFCFS()` is only available when the build option `OS_FCFS` is non-zero.

Arguments

`Sema`	Descriptor of the semaphore to set to a *First Come First Served* mode.

Returns

`void`

Component type

Data access

Options

Notes

If the semaphore is already operating in the *First Come First Served* mode, using this component has no effect on such a semaphore.
If the semaphore is operating in the *Priority* mode, using this component will not re-order the tasks that are currently blocked on the semaphore. Newly blocked tasks will be inserted in a *First Come First Served* ordering behind the already *Priority* ordered blocked tasks. This means there may be a transient phase before the semaphore truly operates in a *First Come First Served* mode.

See also

`OS_FCFS` (Section 4.1.6)
`SEMnotFCFS()` (Section 6.4.4)
`SEMopenFCFS()` (Section 6.4.6)

6.4.11 SEMwait

Synopsis

```
#include "Abassi.h"

int SEMwait(SEM_t *Sema, int Timeout);
```

Description

`SEMwait()` is the component to use to wait on / acquire a semaphore. Through the argument `Timeout`, it is possible to request to block until the acquisition succeeds, or to block with timeout, or no blocking at all.

Availability

Always, but see **Options**

Arguments

`Sema`	Descriptor of the semaphore to acquire.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Number of timer ticks before expiry

Returns

0	The semaphore was acquired.
Non-zero	The semaphore was not acquired. This will occur when the argument `Timeout` is non-negative. Either `Timeout` was zero and the semaphore wasn't acquired, or `Timeout` was positive and the semaphore wasn't acquired within `Timeout` number of timer ticks (or the component `TSKtimeoutKill()` was applied to the task blocked on the semaphore.

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

If the build option `OS_TIMEOUT` is set to zero, then when the argument `Timeout` is set to a positive value, `SEMwait()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is set to a negative value, then when the argument `Timeout` is set to a positive value, `SEMwait()` behaves the same as if the `Timeout` argument had been set to a negative value.
If the build option `OS_FCFS` is non-zero and the semaphore was opened with `SEMopenFCFS()`, or if `SEMsetFCFS()`was used on the semaphore, the unblocking operates on a *First Come First Served* basis instead of the highest *Priority* first.

Notes

Unless `Timeout` is negative, always verify the return value: a non-zero value means the semaphore was not acquired and zero means it was acquired, even if the `Timeout` was set to zero.
When `Timeout` is set to a negative value, the component `TSKtimeoutKill()` cannot unblock the task waiting on the semaphore, as an infinite timeout request does not involve the timer service.

Never use `SEMwait()` in an ISR unless `Timeout` is set to zero. In an interrupt, the value returned by `SEMwait()` is always 0, even if the semaphore was not acquired. If the semaphore was truly acquired its internal count will be decremented. If the semaphore was not acquired, the internal count is left unmodified.

See also

`OS_FCFS` (Section 4.1.6)
`OS_TIMEOUT` (Section 4.1.57)
`SEMopenFCFS()` (Section 6.4.6)
`SEMpost()` (Section 6.4.7)
`SEMsetFCFS()` (Section 6.4.10)
`SEMwaitBin()` (Section 6.4.12)
`TSKtimeoutKill()` (Section 6.3.30)

6.4.12 SEMwaitBin

Synopsis

```
#include "Abassi.h"

int SEMwaitBin(SEM_t *Sema, int Timeout);
```

Description

`SEMwaitBin()` is the component to use to wait on / acquire a semaphore, treating the semaphore in a binary fashion. Through the argument `Timeout`, it is possible to request to block until the acquisition succeeds, or to block with timeout, or no blocking at all.

Availability

Always, but see **Options**

Arguments

`Sema`	Descriptor of the semaphore to acquire	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Timer ticks before expiry

Returns

0	The semaphore was acquired.
Non-zero	The semaphore was not acquired. This will occur when the argument `Timeout` is non-negative. Either `Timeout` was zero and the semaphore wasn't acquired, or `Timeout` was positive and the semaphore wasn't acquired within `Timeout` number of timer ticks (or the component `TSKtimeoutKill()` was applied to the task blocked on the semaphore).

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

If the build option `OS_TIMEOUT` is set to zero, then when the argument `Timeout` is set to a positive value, `SEMwaitBin()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is set to a negative value, then when the argument `Timeout` is set to a positive value, `SEMwaitBin()` behaves the same as if the `Timeout` argument had been set to a negative value.
If the build option `OS_FCFS` is non-zero and the semaphore was opened with `SEMopenFCFS()`, or if `SEMsetFCFS()`was used on the semaphore, the unblocking operates on a *First Come First Served* basis instead of the highest *Priority* first.

Notes

Unless `Timeout` is negative, always verify the return value: a non-zero value means the semaphore was not acquired and zero means it was acquired, even if the `Timeout` was set to zero.
When `Timeout` is set to a negative value, the component `TSKtimeoutKill()` cannot unblock the task waiting on the semaphore, as an infinite timeout request does not involve the timer service.
Semaphores are typically counting semaphores, meaning each posting increments a count register. When a semaphore is posted, it is always as a counting semaphore. The difference between a counting and binary semaphore occurs only when waiting on the semaphore. A single use of `SEMwaitBin()` on a semaphore will zero the internal counter.

Never use `SEMwaitBin()` in an ISR unless `Timeout` is set to zero. . In an interrupt, the value returned by `SEMwaitBin()` is always 0, even if the semaphore was not acquired. If the semaphore was truly acquired its internal count will be decremented. If the semaphore was not acquired, the internal count is left unmodified.

See also

`OS_FCFS` (Section 4.1.6)
`OS_TIMEOUT` (Section 4.1.57)
`SEMopenFCFS()` (Section 6.4.6)
`SEMpost()` (Section 6.4.7)
`SEMsetFCFS()` (Section 6.4.10)
`SEMwait()` (Section 6.4.11)
`TSKtimeoutKill()` (Section 6.3.30)

6.4.13 Examples

6.4.13.1 Semaphore Flushing

Semaphore flushing is an expression that means to bring back the semaphore internal counter to zero. This operation translates into dropping all previous accumulated postings. There is no dedicated component supplied to perform this operation because `SEMwaitBin()` (Section 6.4.12) performs a superset of this operation. All there is to do is as follows:

```
      SEMwaitBin(Semaphore, 0);
```

This will reset the internal counter to zero. Calling `SEMwaitBin()` as such does not change the behavior of the semaphore nor the tasks that are blocked on it.

6.5 Mutex Components

This section describes all components related to the mutexes. The mutexes components described in the following sub-sections are:

Table 6-10 Mutex Component List

Section	Name	Description
6.5.2	`MTX_STATIC`	Creation of a mutex at compile / link time
6.5.3	`MTXabort`	Unblock all tasks blocked on a mutex
6.5.4	`MTXcheckOwn`	Allow only the task locking (owner of) a mutex to unlock it
6.5.5	`MTXgetCeilPrio`	Get the current ceiling priority attached to a mutex
6.5.6	`MTXgetPrioInv`	Get the current on/off of the priority inversion protection
6.5.7	`MTXignoreOwn`	Allow any task to unlock a mutex
6.5.8	`MTXisChkOwn`	Report if a mutex is under owner unlocking protection
6.5.9	`MTXlock`	Lock / acquire a mutex
6.5.10	`MTXnotFCFS`	Set a mutex to operate in the *Priority* mode
6.5.11	`MTXopen`	Create a mutex / obtain the descriptor of a mutex
6.5.12	`MTXopenFCFS`	Create a mutex to operate in *First Come First Served* mode Obtain the descriptor of a mutex
6.5.13	`MTXowner`	Report if a mutex is locked Report the task descriptor of the locker of a mutex
6.5.14	`MTXprioInvOff`	Disable the mutex priority inversion protection
6.5.15	`MTXprioInvOn`	Enable the mutex priority inversion protection
6.5.16	`MTXsetCeilPrio`	Set the ceiling priority attached to a mutex
6.5.17	`MTXsetFCFS`	Set a semaphore to operate in the *First Come First Served* mode
6.5.18	`MTXunlock`	Unlock / procure a mutex

6.5.1 Description

As stated in a previous section, mutexes are simply semaphores in disguise. Almost everything explained about the Abassi semaphores in Section 6.4 applies to mutexes. The only differences between mutexes and semaphores are the followings:

- Mutexes always operate in a binary fashion
- Mutexes are created with a single initial "posting"
- All mutexes are reentrant mutexes: the mutex locker can apply multiple locks on a mutex without getting blocked on that mutex
- Mutexes can trigger priority inheritance / priority ceiling on the owning task when the feature is enabled
- A task that locks one or more mutexes will not get suspended as long as it locks the mutex(es), if this feature is enabled by setting the build option `OS_TASK_SUSPEND` to a non-zero value

- NOTE: If for a reason or another non-reentrant mutexes are required in the application all there is to do is to use a semaphore as a replacement. After the creation of a semaphore perform a "dummy" SEMpost() on it. After tha, a `SEMwait()` is equivalent to a mutex lock and a `SEMpost()` the equivalent to a mutex unlock. As it is a semaphore, then it is not possible to protect it against priority inversion protection or to detect a deadlock on it.

6.5.2 MTX_STATIC

Synopsis

```
#include "Abassi.h"

MTX_STATIC(VarName, MtxName);
```

Description

`MTX_STATIC()` is a special component that creates a mutex and initializes its descriptor. It is a macro definition creating a static object, so none of the arguments has a real data type. The mutex is not created/initialized at run time; everything is done at compile/link time.

Availability

Always

Arguments

`VarName`	Name of the variable holding the pointer to the mutex descriptor to create / initialize. This is a variable name therefore do not put double quotes around the name.
`MtxName`	Mutex name. This is not the variable name, it is the name attached to the mutex. As it is a "C" string, the double quotes around the name are required. `G_OSnoName` , and not `NULL`, should be used for an unnamed mutex

Returns

N/A

Component type

Macro (safe)

Options

If the build option `OS_NAMES` is set to a value of zero, the argument `MtxName` is ignored but must still be supplied.

Notes

A mutex created and initialized with `MTX_STATIC()` will not be part of the search done with `MTXopen()`or `MTXopenFCFS()`, unless another mutex (or semaphore) with the exactly the same name was created using `MTXopen()` or `MTXopenFCFS()`.
All mutexes created with `MTX_STATIC()` are set to operate in the *Priority* mode upon creation. To make a mutex created with `MTX_STATIC()` operate in the *First Come First Seerved* mode, apply the component `MTXsetFCFS()` on the mutex.
If mutexes are created using the `MTX_STATIC()` component, it may not be possible to restart the application without reloading the binary image on the processor. This situation happens if the compiler (or compiler configuration) does not reload the initialized data upon start-up.

See also

`OS_NAMES` (Section 4.1.28)
`MTXsetFCFS()` (Section 6.5.17)
`MTXopen()` (Section 6.5.11)
`MTXopenFCFS()` (Section 6.5.12)
`G_OSnoName` (Section 6.14.2)

6.5.3 MTXabort

Synopsis

```
#include "Abassi.h"

void MTXabort(MTX_t *Mutex);
```

Description

`MTXabort()` is a component that unblocks all tasks that are blocked trying to lock the mutex `Mutex`. This is alike applying the component `TSKtimeoutKill()` on all the tasks blocked on the mutex `Mutex` with the extra capability to also unblocked tasks blocked with an infinite timeout.

Availability

`MTXabort()` is only available when the build option `OS_WAIT_ABORT` is defined and non-zero.

Arguments

`Mutex`	Descriptor of the mutex to unblock all the tasks that are blocked on.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

All tasks that blocked on the `MTXlock()` component are unblocked when the component `MTXabort()` is used on the mutex. The `MTXlock()` component then returns a non-zero value to indicate a timeout occurred, even if the wait time requested was infinite. The task that owns the lock on the mutex still owns the lock on that mutex once the component `MTXabort()` has been used; it does not lose the lock as the owner is not blocked on the mutex.

If there are no tasks blocked on the mutex `Mutex`, the use of `MTXabort()` on `Mutex` has no effects.

See also

`OS_WAIT_ABORT` (Section 4.1.62)
`MTXlock()` (Section 6.5.9)

6.5.4 MTXcheckOwn

Synopsis

```
#include "Abassi.h"

void MTXcheckOwn(MTX_t *Mutex);
```

Description

`MTXcheckOwn()` is a component that enables a feature where only the task that locks (the mutex owner) can unlock a mutex. When this feature is enabled, a mutex can always be unlocked in an interrupt handler.

Availability

`MTXcheckOwn()` is only available when the build option `OS_MTX_OWN_UNLOCK` is set to a negative value. By default, all new mutexes have the protection enabled.

Arguments

`Mutex`	Descriptor of the mutex to enable ownership validation on.

Returns

void

Component type

Data access

Options

Notes

See also

`OS_MTX_OWN_UNLOCK` (Section 4.1.27)
`MTXignoreOwn()` (Section 6.5.7)
`MTXisChkOwn()` (Section 6.5.8)

6.5.5 MTXgetCeilPrio

Synopsis

```
#include "Abassi.h"

int MTXgetCeilPrio(MTX_t *Mutex);
```

Description

`MTXgetCeilPrio()` is a component that reports the current ceiling priority value attached to a mutex. The ceiling priority value is the priority a task will be promoted to run at when it is locking a mutex with one or more tasks blocked on it.

Availability

`MTXgetCeilPrio()` is only available when the build option `OS_MTX_INVERSION` is set to a negative value.

Arguments

`Mutex`	Descriptor of the mutex to retrieve the current ceiling priority value.

Returns

Current value of the ceiling priority

Component type

Data access

Options

Notes

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`MTXsetCeilPrio()` (Section 6.5.16)
Priority Inversion (Section 7)

6.5.6 MTXgetPrioInv

Synopsis

```
#include "Abassi.h"

int MTXgetPrioInv(MTX_t *Mutex);
```

Description

`MTXgetPrioInv()` is a component that reports the priority inversion protection setting of a mutex..

Availability

`MTXprioInvOff()` is only available when the build option `OS_MTX_INVERSION` is either set to a value greater than 999 or to a value less than -999.

Arguments

`Mutex`	Descriptor of the mutex to retrieve the current priority inversion protection setting.

Returns

0	The mutex is not under priority inversion protection
Non-zero	The mutex is under priority inversion protection

Component type

Data access

Options

Notes

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`MTXprioInvOff()` (Section 6.5.14)
`MTXprioInvOn()` (Section 6.5.15)
Priority Inversion (Section 7)

6.5.7 MTXignoreOwn

Synopsis

```
#include "Abassi.h"

void MTXignoreOwn(MTX_t *Mutex);
```

Description

`MTXignoreOwn()` is a component that disables a feature where only the task that locks (the mutex owner) can unlock a mutex. Therefore using the component `MTXignoreOwn` on a mutex allows all tasks to unlock the mutex, even if the task is not the owner of the mutex.

Availability

`MTXignoreOwn()` is only available when the build option `OS_MTX_OWN_UNLOCK` is set to a negative value. By default, all new mutexes have the protection enabled.

Arguments

`Mutex`	Descriptor of the mutex to disable ownership validation on.

Returns

void

Component type

Data access

Options

Notes

See also

`OS_MTX_OWN_UNLOCK` (Section 4.1.27)
`MTXcheckOwn()` (Section 6.5.4)
`MTXisChkOwn()` (Section 6.5.8)

6.5.8 MTXisChkOwn

Synopsis

```
#include "Abassi.h"

int MTXisChkOwn(MTX_t *Mutex);
```

Description

`MTXisChkOwn()` is a component that reports if the feature where only the task that locks (the mutex owner) can unlock a mutex is active or not.

Availability

`MTXisChkOwn()` is only available when the build option `OS_MTX_OWN_UNLOCK` is set to a negative value.

Arguments

`Mutex`	Descriptor of the mutex to retrieve the ownership validation status for.

Returns

0	The mutex is not protected against a non-owner task unlocking it
Non-zero	The mutex is protected against a non-owner task unlocking it

Component type

Data access

Options

Notes

See also

`OS_MTX_OWN_UNLOCK` (Section 4.1.27)
`MTXcheckOwn()` (Section 6.5.4)
`MTXignoreOwn()` (Section 6.5.7)

6.5.9 MTXlock

Synopsis

```
#include "Abassi.h"

int MTXlock(MTX_t *Mutex, int Timeout);
```

Description

`MTXlock()` is the component to use to acquire a lock on a mutex. Through the argument `Timeout`, it is possible to request to block until the acquisition succeeds, or to block with timeout, or no blocking at all.

Availability

Always, but see **Options**

Arguments

`Mutex`	Descriptor of the mutex to try to acquire a lock on.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Timer ticks before expiry

Returns

0	The lock was acquired.
Non-zero	The lock was not acquired. Either `Timeout` was zero and the mutex is already locked by another task, or `Timeout` was positive and the mutex lock wasn't acquired within `Timeout` number of timer ticks (or the component `TSKtimeoutKill()` was applied to the task blocked on the mutex). A lock failure can also occur when a mutex deadlock condition is detected (this feature is enabled when the build option `OS_MTX_DEADLOCK` set to non-zero).

Component type

Atomic macro (safe)
- Cannot be used in an interrupt -

Options

If the build option `OS_TIMEOUT` is set to zero, then when the argument `Timeout` is set to a positive value, `MTXlock()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is set to a negative value, then when the argument `Timeout` is set to a positive value, `MTXlock()` behaves the same as if the `Timeout` argument had been set to a negative value.
If the build option `OS_FCFS` is non-zero and the mutex was opened with `MTXopenFCFS()`, or modified with `MTXsetFCFS()`, the time order of the unblocking is on a *First Come First Served* basis instead of the highest *Priority* first.
If the build option `OS_MTX_INVERSION` is positive, then priority inheritance is applied on all mutexes.
If the build option `OS_MTX_INVERSION` is negative, then priority ceiling is applied on all mutexes.

Notes

Even if `Timeout` is negative, always verify the return value: a non-zero value means the mutex was not locked, zero means it was acquired, even if the `Timeout` was set to zero. An infinite timeout (`Timeout` is negative) would report a lock failure if a mutex deadlock is detected (when this feature is enabled).
When `Timeout` is set to a negative value, the component `TSKtimeoutKill()` cannot unblock the task trying to lock the mutex, as an infinite timeout request does not involve the timer service.

Never use `MTXlock()` in an ISR unless `Timeout` is set to zero.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MTX_DEADLOCK` (Section 4.1.25)
`OS_MTX_INVERSION` (Section 4.1.26)
`OS_TIMEOUT` (Section 4.1.57)
`MTXopenFCFS()` (Section 6.5.12)
`MTXsetFCFS()` (Section 6.5.17)
`MTXunlock()` (Section 6.5.18)
`TSKtimeoutKill()` (Section 6.3.30)

6.5.10 MTXnotFCFS

Synopsis

```
#include "Abassi.h"

void MTXnotFCFS(MTX_t *Mutex);
```

Description

`MTXnotFCFS()` is the component to use to configure a mutex to operate in the *Priority* mode. The unblocking order of such a mutex is always the highest priority task first.

Availability

`MTXnotFCFS()` is only available when the build option `OS_FCFS` is non-zero.

Arguments

`Mutex`	Descriptor of the mutex to set to the *Priority* mode.

Returns

`void`

Component type

Data access

Options

Notes

If the mutex was already operating in the *Priority* mode, using this component has no effect on the mutex.
If the mutex was operating in the *First Come First Served* mode, using this component will not re-order tasks that are currently blocked on the mutex. Newly blocked tasks will be inserted in a *Priority* ordering amongst the already *First Come First Served* ordered blocked tasks. This means there may be a transient phase before the mutex truly operates in a *Priority* mode.

See also

`OS_FCFS` (Section 4.1.6)
`MTXopenFCFS()` (Section 6.5.12)
`MTXsetFCFS()` (Section 6.5.17)

6.5.11 MTXopen

Synopsis

```
#include "Abassi.h"

MTX_t *MTXopen(const char *Name);
```

Description

`MTXopen()` is the component to use to create a mutex, and is also the component to use to obtain the descriptor of an already existing mutex.

Availability

`MTXopen()` is only available when the build option `OS_RUNTIME` is non-zero.

Arguments

`Name`	Name of the mutex to create or to obtain the descriptor of.

Returns

Descriptor of the mutex

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build where `OS_NAMES` is zero, all mutexes are unnamed and every use of `MTXopen()` creates a new mutex.
If the build option `OS_NAMES` is non-zero, then `MTXopen()` will either return the descriptor of an existing mutex (previously created with `MTXopen()` or `MTXopenFCFS()`, Section 6.5.12), or when no mutex with the specified name exists, it will create a new mutex. This approach makes the creation and opening of mutexes run-time safe. If that feature was not part of the `MTXopen()` component, it would be imperative to either create the mutex immediately at start-up or to guarantee the first task (using the mutex) to reach the running state is the one creating the mutex. With the run-time safe feature, it does not matter which task is the first to use the mutex.
When the build option `OS_MTX_INVERSION` value is either greater than 999 or less than -999, the mutex is always created with the priority inversion protection enable. If the protection is not required, the component `MTXprioInvOff()` must be applied on the mutex to disable the priority inversion protection.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed mutexes. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.

Be aware, if the build option `OS_FCFS` is non-zero, when the mutex already exists, there is no guarantee the mutex is operating in a *Priority* mode as it may have been created with `MTXopenFCFS()` or it may have been set to operate in *First Come First Served* mode with `MTXsetFCFS()`.

At any time a mutex operating in the *Priority* mode can be modified to operate in the *First Come First Served* mode by using the `MTXsetFCFS()` component when the build option `OS_FCFS` is non-zero.

Mutexes created with the `MTX_STATIC()` component are not part of the search performed by `MTXopen()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MTX_INVERSION` (Section 4.1.26)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`MTXprioInvOff()` (Section 6.5.14)
`MTXprioInvOn()` (Section 6.5.15)
`MTXopenFCFS()` (Section 6.5.12)
`MTXsetFCFS()` (Section 6.5.17)

6.5.12 MTXopenFCFS

Synopsis

```
#include "Abassi.h"

MTX_t *MTXopenFCFS(const char *Name);
```

Description

`MTXopenFCFS()` is the component to use to create a mutex operating in a *First Come First Served* mode, and is also the component to use to obtain the descriptor of an already existing mutex.

Availability

`MTXopenFCFS()` is only available when the build options `OS_RUNTIME` and `OS_FCFS` are non-zero.

Arguments

`Name`	Name of the mutex to create or to obtain the descriptor of.

Returns

Descriptor of the mutex

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build where `OS_NAMES` is zero, all mutexes are unnamed and every use of `MTXopenFCFS()` creates a new mutex.
If the build option `OS_NAMES` is non-zero, then `MTXopenFCFS()` will either return the descriptor of an existing mutex (previously created with `MTXopen()` or `MTXopenFCFS()`), or when no mutex with the specified name exists, it will create a new mutex. This approach makes the creation and opening of mutexes run-time safe. If that feature was not part of the `MTXopenFCFS()` component, it would be imperative to either create the mutex immediately at start-up or to guarantee the first task (using the mutex) to reach the running state is the one creating the mutex. With the run-time safe feature, it does not matter which task is the first to use the mutex.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed mutexes. This is also the reason why the function prototype for this component was kept the same, irrelevant to the setting of the build option `OS_NAMES`.

Be aware, if the build option `OS_FCFS` is non-zero, when the mutex already exists, there is no guarantee the mutex is operating in a *First Come First Served* mode as it may have been created with `MTXopen()` or it may have been set to operate in *Priority* mode with `MTXnotFCFS()`.

At any time, a mutex operating in the *First Come First Served* mode can be modified to operate in the *Priority* mode by using the `MTXnotFCFS()` component when the build option `OS_FCFS` is non-zero.

Mutexes created with the `MTX_STATIC()` component are not part of the search performed by `MTXopenFCFS()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`MTXnotFCFS()` (Section 6.5.10)
`MTXopen()` (Section 6.5.11)

6.5.13 MTXowner

Synopsis

```
#include "Abassi.h"

TSK_t *MTXowner(MTX_t *Mutex);
```

Description

`MTXowner()` is the component that reports if a mutex is currently locked and, if so, what is the task locking the mutex.

Availability

Always

Arguments

`Mutex`	Descriptor of the mutex to retrieve the lock status and locker information.

Returns

`NULL`	The mutex is not locked
`Non-NULL`	Descriptor of the task that currently has a lock on the mutex

Component type

Data access

Options

Notes

See also

6.5.14 MTXprioInvOff

Synopsis

```
#include "Abassi.h"

void MTXprioInvOff(MTX_t *Mutex);
```

Description

`MTXprioInvOff()` is a component that disable the priority inversion, be either priority ceiling or priority inheritance, on the mutex specified by the argument `Mutex`

Availability

`MTXprioInvOff()` is only available when the build option `OS_MTX_INVERSION` are is either set to a value greater than 999 or to a value less than -999.-zero.

Arguments

`Mutex`	Descriptor of the mutex to disable the priority inversion protection.

Returns

`void`

Component type

Macro (safe)

Options

Notes

If the mutex is already locked and the locking task has its priority increased due to priority inversion protection, using the component `MTXprioInvOff()` on the mutex will not bring back the priority of the locking task to its original value. Only when the task unlocks the mutex will it go back to its original priority. Undr the same condition, if more task at higher priority get blocked trying to lock the mutex, the current owner will not gets its priority changed.
If the priority inversion protection of the mutex is already disabled, the use of the component `MTXprioInvOff()` on the mutex has no impact.

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`MTXgetPrioInv()` (Section 6.5.6)
`MTXprioInvOn()` (Section 6.5.15)
Priority Inversion (Section 7)

6.5.15 MTXprioInvOn

Synopsis

```
#include "Abassi.h"

void MTXprioInvOn(MTX_t *Mutex);
```

Description

`MTXprioInvOff()` is a component that enable the priority inversion, be either priority ceiling or priority inheritance, on the mutex specified by the argument `Mutex`

Availability

`MTXprioInvOn()` is only available when the build option `OS_MTX_INVERSION` are is either set to a value greater than 999 or to a value less than -999.-zero.

Arguments

`Mutex`	Descriptor of the mutex to disable the priority inversion protection.

Returns

`void`

Component type

Macro (safe)

Options

Notes

When the mutex priority inversion protection is disabled and if the mutex is already locked and other higher priority task are blocked trying to lock the mutex, then using the component MTXprioInvOn() on the mutex will not change the priority of the locking task. The priority change will only occur if a new task of higher priority gets blocked trying to lock the mutex. When the locking task unlock the mutex, the new locking task will be handled according to the selected priority inversion protection scheme.
If the priority inversion protection of the mutex is already enable, the use of the component `MTXprioInvOn()` on the mutex has no impact.

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`MTXgetPrioInv()` (Section 6.5.6)
`MTXprioInvOff()` (Section 6.5.14)
Priority Inversion (Section 7)

6.5.16 MTXsetCeilPrio

Synopsis

```
#include "Abassi.h"

void MTXsetCeilPrio(MTX_t *Mutex, int Prio);
```

Description

`MTXsetCeilPrio()` is a component that sets the current ceiling priority value attached to a mutex. The ceiling priority value is the promoted priority a task will run at when it is locking a mutex with one or more tasks blocked on.

Availability

`MTXsetCeilPrio()` is only available when the build option `OS_MTX_INVERSION` is set to a negative value.

Arguments

`Mutex`	Descriptor of the mutex to set the current priority ceiling value.
`Prio`	New ceiling priority to attach to the mutex

Returns

`void`

Component type

Data access

Options

Notes

Modifying the ceiling priority of a mutex does not stop the automatic priority increase performed by the priority ceiling mechanism; it simply updates the ceiling prioriy of the mutex. There could be two reasons why one would set a ceiling priority on a mutex. The first would be to deal with the situation where a high priority level task, which was one that could lock the mutex, gets suspended. It does not make sense to keep raising the priority of tasks locking the mutex to such a high priority level when no tasks at that priority would lock the mutex anymore. The second case would be to set the mutex to the final ceiling priority; the one matching the priority of the highest priority task that locks this mutex. The latter case is a bit irrelevant as there is no CPU saving doing so.
When the ceiling priority is modified on a mutex that is currently operating in the priority ceiling mechanism, nothing changes until the current locker of the mutex unlocks it. Then the new setting for the ceiling priority is taken into account.

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`MTXgetCeilPrio()` (Section 6.5.5)

Priority Inversion (Section 7)

6.5.17 MTXsetFCFS

Synopsis

```
#include "Abassi.h"

void MTXsetFCFS(MTX_t *Mutex);
```

Description

`MTXsetFCFS()` is the component to use to configure a mutex to operate in a *First Come First Served* mode.

Availability

`MTXsetFCFS()` is only available when the build option `OS_FCFS` is non-zero.

Arguments

`Mutex`	Descriptor of the mutex to set into a *First Come First Served* mode.

Returns

`void`

Component type

Definition

Options

Notes

If the mutex was already operating in the *First Come First Served* mode, using this component has no effect on the mutex.
If the mutex was operating in the *Priority* mode, using this component will not re-order tasks that are currently blocked on the mutex. Newly blocked tasks will be inserted in a *First Come First Served* ordering amongst the already *Priority* ordered blocked tasks. This means there may be a transient phase before the mutex truly operates in a *First Come First Served* mode.

See also

`OS_FCFS` (Section 4.1.6)
`MTXnotFCFS()` (Section 6.5.10)
`MTXopenFCFS()` (Section 6.5.12)

6.5.18 MTXunlock

Synopsis

```
#include "Abassi.h"

int MTXunlock(MTX_t *Mutex);
```

Description

`MTXunlock()` is the component that relinquishes the lock on a mutex.

Availability

Always

Arguments

`Mutex`	Descriptor of the mutex to unlock.

Returns

`0`	When the build option `OS_MTX_OWN_UNLOCK` is disabled: always
`0`	When the build option `OS_MTX_OWN_UNLOCK` is enabled: the mutex has been successfully unlocked.
`!= 0`	When the build option `OS_MTX_OWN_UNLOCK` is enabled: the mutex has been not been unlocked (the task performing the unlock operaton does not own the mutex.

Component type

Atomic macro (safe)

Options

If the build option `OS_MTX_INVERSION` is positive, then priority inheritance is applied on all mutexes.
If the build option `OS_MTX_INVERSION` is negative, then priority ceiling is applied on all mutexes.

Notes

Mutexes operate the same way as binary semaphores. This means if `MTXunlock()` is repeatedly used without a matching count of `MTXlock()`, a single lock will be available; not the difference of counts.
When the locker of a mutex has applied more than one `MTXlock()` on the mutex, exactly the same number of `MTXunlock()` must be applied before the mutex gets unlocked.

When the build option `OS_MTX_OWN_UNLOCK` is enabled `MTXunlock()` should not be used inside interrupts. The build option `OS_MTX_OWN_UNLOCK` restricts the unlocking operations of a mutex to the task that locks the mutex, as such, interrupts have to be considered as not owning the lock on any mutexes.

See also

`OS_MTX_INVERSION` (Section 4.1.26)
`OS_MTX_OWN_UNLOCK` (Section 4.1.27)
`MTXlock()` (Section 6.5.9)

6.5.19 Examples

6.6 Event Components

This section describes all components related to the event flags. The event flags components described in the following sub-sections are:

Table 6-11 Event Component List

Section	Name	Description
6.6.2	`EVTabort`	Unblock a task blocked on its events
6.6.3	`EVTget`	Retrieve the event flags that validated the last mask conditions
6.6.4	`EVTgetAcc`	Retrieve the current event flags
6.6.5	`EVTreset`	Clear the event flags that validated the last mask conditions
6.6.6	`EVTresetAcc`	Clear the current event flags
6.6.7	`EVTset`	Set event flags
6.6.8	`EVTwait`	Wait for event flags to validate mask conditions

6.6.1 Description

What are named Events in the Abassi RTOS are a group of flags (bits). These bits are used as a synchronization mechanism in which each task owns a unique event register (register holding the flags). A task may check or get blocked until a selected combination of flags has been set in its event register. Any task can set the flags in any task's register, but individual flags cannot be reset. A task can be blocked until one out of two conditions on the flags are met: one condition is an AND mask where the bits set in the mask specifies that all the corresponding flags must be set, and the other condition is an OR mask, where the bits set in the mask specifies that any of the flags must be set. When one or both conditions are true, the task can retrieve the flags received.

To better understand how to use the Event component in the Abassi RTOS, here is a description on the internals. Each task possesses 4 registers dedicated for the events handling:

- AND Mask register
- OR Mask register
- Event Accumulation register
- Event Received register

When a task needs to get synchronized on events, `EVTwait()` is used. Using this component sets the values in the OR and the AND mask registers, and a timeout. Neither the Event Accumulation register nor the Event Received register are cleared when `EVTwait()` is used; past flags that were set are kept as is. Any task can set one or more events flags, which are always set in the Event Accumulation register, but only the event owner can reset the flags in this register. Every time a task sets flags and the event owner is waiting on the events, the contents of the Event Accumulation register is verified against the two Mask registers. When the flags fulfill one of the two masks conditions, and the flag owner use the component `EVTwait()`, the following occurs:

- The Event Accumulation register is copied into the Event Received register
- The Event Accumulation register is zeroed
- The AND mask register is zeroed
- The OR mask register is zeroed
- The task gets unblocked (if was blocked on the events)

If the component `EVTwait()` is used with a timeout and the waiting expires before the flag conditions are met, only 2 operations are performed: the AND mask and the OR mask registers are zeroed. The Event Accumulation and the Event Received registers remain untouched.

To retrieve the contents of the Event Received register, the component `EVTget()` is used. And at any time it is possible to peek at the contents of the Event Accumulation register by using the component `EVTgetAcc()`.

There is no special component for events (a component that would be named `EVT_STATIC()`) like there are for the other services. As the event registers are part of the task descriptors, events are statically created/initialized when `TSK_STATIC()` (Section 6.3.2) is used.

6.6.2 EVTabort

Synopsis

```
#include "Abassi.h"

void EVTabort(TSK_t *Task);
```

Description

`EVTabort()` is a component that unblocks a task that is blocked waiting on its event flags. The task to unblock is specified with the argument `Task`. This is alike applying the component `TSKtimeoutKill()` on the task blocked on its events with the extra capability to also unblocked a task blocked with an infinite timeout

Availability

`EVTabort()` is only available when the build option `OS_WAIT_ABORT` is defined and non-zero and the build option `OS_EVENTS` is non-zero.

Arguments

`Task`	Descriptor of the task to unblock when it's waiting on its events.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

A task that blocked on the `EVTget()` or `MBXput()` component is unblocked when the component `EVTabort()` is used on the task. The `EVTwait()` component then returns a non-zero value to indicate a timeout occurred, even if the wait time requested was infinite.

If the task `Task` is not blocked on its events, the use of `EVTabort()` on it has no effects.

See also

`OS_WAIT_ABORT` (Section 4.1.62)
`EVTwait()` (Section 6.6.8)

6.6.3 EVTget

Synopsis

```
#include "Abassi.h"

int EVTget(void);
```

Description

`EVTget()` is the component to use to retrieve the most recent event flags that have met the last AND / OR conditions when `EVTwait()`was successful. No task descriptor needs to be supplied to `EVTget()` because only the task owning the event registers can read the flags.

Availability

`EVTget()`is only available if the build option `OS_EVENTS` is non-zero

Arguments

`void`

Returns

The contents of the Event Received register.

Component type

Data access
- Meaningless in an interrupt -

Options

Notes

This component does not wait, nor set the conditional masks. Its only use is to read the last flags set that met the AND / OR conditions.
If `EVTwait()` times out before having received the flags that meet the AND / OR flags conditions, the event received register is neither zeroed, nor does it inherit the contents of the event accumulation register: the event register always holds the last successful valid flags combination.

See also

`OS_EVENTS` (Section 4.1.5)
`EVTset()` (Section 6.6.7)
`EVTwait()` (Section 6.6.8)

6.6.4 EVTgetAcc

Synopsis

```
#include "Abassi.h"

int EVTgetAcc(void);
```

Description

`EVTgetAcc()` is the component to use to peek at the flags that have been set since the last successful call to `EVTwait()`. No task descriptor needs to be supplied to `EVTgetAcc()` because only the task owning the events can read the flags.

Availability

`EVTgetAcc()` is only available if the build option `OS_EVENTS` is non-zero.

Arguments

`void`

Returns

The contents of the Event Accumulation register.

Component type

Data access
- Meaningless in an interrupt -

Options

Notes

This component does not wait, nor set the conditional masks. Its only use is to read the current flags that have been set since the last use of `EVTreset()` or the last successful use of `EVTwait()`. If `EVTwait()` times out before having received the flags that meet the AND / OR flags conditions, the contents of event accumulation register is not forced to zero but remain at its current setting.

See also

`OS_EVENTS` (Section 4.1.5)
`EVTget()` (Section 6.6.3)
`EVTreset()` (Section 6.6.5)
`EVTset()` (Section 6.6.7)
`EVTwait()` (Section 6.6.8)

6.6.5 EVTreset

Synopsis

```
#include "Abassi.h"

void EVTreset(void);
```

Description

`EVTreset()` is used to clear the contents of the Event Receive register. No task descriptor needs to be supplied to `EVTreset()` because only the task owning the events register can clear the flags.

Availability

`EVTreset()` is only available if the build option `OS_EVENTS` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Data access
- Cannot be used in an interrupt -

Options

Notes

When `EVTwait()` is used and the flags meet the AND / OR mask conditions, the Event Receive register holds the flags that has validated the mask conditions. `EVTreset()` usefulness is to remove the flags in the received register since the last successful use of `EVTwait()`.
Only the owner of the event flags can reset the flags. If other tasks were able to reset flags, it could provoke a synchronization issue. For example, if a task is waiting on a single flag, and then gets it, it would go into the ready to run state. If higher priority tasks are using the CPU and one of these tasks resets the flag, then from the event owner this would have to be declared a false synchronization trigger.

See also

`OS_EVENTS` (Section 4.1.5)
`EVTget()` (Section 6.6.3)
`EVTset()` (Section 6.6.7)
`EVTwait()` (Section 6.6.8)

6.6.6 EVTresetAcc

Synopsis

```
#include "Abassi.h"

void EVTresetAcc(void);
```

Description

`EVTresetAcc()` is used to clear the contents of the Event Accumulation register. No task descriptor needs to be supplied to `EVTresetAcc()` because only the task owning the events register can clear the flags.

Availability

`EVTresetAcc()` is only available if the build option `OS_EVENTS` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Data access
- Cannot be used in an interrupt -

Options

Notes

When `EVTwait()` is used and the flags meet the AND / OR mask conditions, the Event Receive register holds the flags that has validated the mask condition. On the counterpart, the event accumulation register holds the flags that were set that still have not matched a mask condition. `EVTresetAcc()` usefulness is to remove the flags in the accumulation register that have been set since the last successful use of `EVTwait()`.
Only the owner of the event flags can reset the flags in the accumulation register

See also

`OS_EVENTS` (Section 4.1.5)
`EVTget()` (Section 6.6.3)
`EVTset()` (Section 6.6.7)
`EVTwait()` (Section 6.6.8)

6.6.7 EVTset

Synopsis

```
#include "Abassi.h"

void EVTset(TSK_t *Task, int Bits);
```

Description

`EVTset()` is used to set the event flags in the Event Accumulation register of the task specified with the argument `Task`.

Availability

`EVTset()`is only available if the build option `OS_EVENTS` is non-zero.

Arguments

`Task`	Descriptor of the task to set the event flags.
`Bits`	Data of type `int` that specifies which flags to set. Bits set to 1 in `Bits` activate the respective flag in the event accumulation register of the task indicated by the argument `Task`.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

See also

`OS_EVENTS` (Section 4.1.5)
`EVTget()` (Section 6.6.3)
`EVTset()` (Section 6.6.7)
`EVTwait()` (Section 6.6.8)

6.6.8 EVTwait

Synopsis

```
#include “Abassi.h”

int EVTwait(int ANDmask, int ORmask, int Timeout);
```

Description

`EVTwait()` is the component used to synchronize a task with the validation of event flags. An event is validated when all the bits in the `ANDmask` are set, or any of the bits in the `ORmask` are set. Through the argument `Timeout`, it is possible to request to block until the acquisition succeeds, or to block with timeout, or no blocking at all.

Availability

`EVTwait()` is only available if the build option `OS_EVENTS` is non-zero; also see **Options**.

Arguments

`ANDmask`	Bit field mask where the bit(s) set to 1 must cumulatively be set with `EVTset()` to validate the event.	
`ORmask`	Bit field mask where any of the bit(s) set to 1 must be set with `EVTset()` to validate the event.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Timer ticks before expiry

Returns

0	The event flags conditions have matched the AND / OR masks.
Non-zero	The event flags did not meet the AND / OR conditions. This will occur if the argument `Timeout` is non-negative. Either `Timeout` was zero and the event flags didn’t match the masks, or `Timeout` was positive and the event flags have not yet matched the masks within `Timeout` number of timer ticks (or when the component `TSKtimeoutKill()` is applied to the task that owns the events).

Component type

Atomic macro (unsafe)
- Cannot be used in an interrupt -

Options

If the build option `OS_TIMEOUT` is set to zero, then when the argument `Timeout` is set to a positive value, `EVTwait()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is set to a negative value, then when the argument `Timeout` is set to a positive value, `EVTwait()` behaves the same as if the `Timeout` argument had been set to a negative value.

Notes

Unless `Timeout` is negative, always verify the return value: a non-zero value means the flags have not met the condition, zero means the flag matched to conditions, even if the `Timeout` was set to zero.
When timeout is negative, the component `TSKtimeoutKill()` cannot unblock the task blocked on the events, as an infinite timeout request does not involve the timer service.
When the event flags match the mask conditions, both masks are reset to zero, the contents of the Event Accumulation register is copied into the Event Received register, and the Event Accumulation register is reset to zero.
If the blocking expires before the flags have met the conditions, the masks are also reset to zero, but the contents of the Event Accumulation register and the Event Received register are left untouched.
To obtain the flags that have matched the two masks conditions, the component `EVTget()` is used.

`EVTwait()` should never be used in an ISR since the component always applies to the currently running task.

See also

`OS_EVENTS` (Section 4.1.5)
`OS_TIMEOUT` (Section 4.1.57)
`EVTget()` (Section 6.6.3)
`EVTset()` (Section 6.6.7)
`TSKtimeoutKill()` (Section 6.3.30)

6.6.9 Examples

6.7 Mailboxes components

This section describes all components related to the mailboxes. The mailbox components described in the following sub-sections are:

Table 6-12 Mailbox Component List

Section	Name	Description
6.7.2	`MBX_STATIC`	Creation of a mailbox at compile / link time
6.7.3	`MBXabort`	Unblock all tasks blocked on a mailbox
6.7.4	`MBXavail`	Report how many element are free in a mailbox
6.7.5	`MBXget`	Retrieve one message from a mailbox
6.7.6	`MBXnotFCFS`	Set a mailbox to operate in the *Priority* mode
6.7.7	`MBXopen`	Create a mailbox / obtain the descriptor of a mailbox
6.7.8	`MBXopenFCFS`	Create a mailbox to operate in *First Come First Served* mode Obtain the descriptor of a mailbox
6.7.9	`MBXput`	Insert one message in a mailbox
6.7.10	`MBXputInISR`	Enable the validation of the return value when `MBXput()` is called from within an interrupt handler.
6.7.11	`MBXsetFCFS`	Set a mailbox to operate in the *First Come First Served* mode
6.7.12	`MBXused`	Report how many elements are in used in a mailbox

6.7.1 Description

A mailbox is a mechanism to communicate information between tasks, where the reader retrieves the information in a first in, first out manner. Compared to a queue, a mailbox conveys fixed sized messages between tasks. In the case of the Abassi RTOS, the message size is always of type `intptr_t`, which means it can be either an `int` or a pointer. The individual messages have a pre-defined data type, but a mailbox can be created to hold as many messages as needed.

One important point to remember about mailboxes it that they are a one-reader / multiple-writer system, and that is how the Abassi RTOS has been designed to handle mailboxes. If multiple readers are accessing the same mailbox, the mailbox system operates correctly, but the behavior will quite likely not be as desired as two or more tasks can empty the mailbox at will; much like your neighbor stealing your morning newspaper.

A mailbox reader can block when the mailbox is empty. The same applies when the mailbox is full; writers can block. Mailboxes can operate in two different modes: the writers (and readers, if more than one) get either blocked in a *First Come First Served* fashion or they get blocked in a *Priority* based fashion. The former mode is normally used with time sensitive information, when the latter is typically used for critical information.

The exact size of the buffer holding the message of a mailbox is always one more than the size requested at mailbox creation. This was chosen for 2 reasons. First, it simplifies the determination if a mailbox is empty or full. Second, and more importantly, because mailboxes are expected to be the core resource when queues need to be implemented. With queues, it becomes necessary to make sure that when the reader gets a queue buffer, that this buffer does not becomes re-used by a writer during the time the reader accesses the buffer. That extra buffer element in the mailbox is the safeguard against the possible buffer clash between reader and writer.

Queues are not supported by the Abassi RTOS, but it is quite straightforward to create queues using the mailbox service. Section 6.7.13.3 gives example on how to create queues based on the Abassi RTOS mailboxes.

6.7.2 MBX_STATIC

Synopsis

```
#include "Abassi.h"

MBX_STATIC(VarName, MbxName, Size);
```

Description

`MBX_STATIC()` is a special component that creates a mailbox and the associated circular buffer, and that initializes its descriptor. It is a macro definition creating a static object, so none of the arguments has a real data type. The mailbox is not created/initialized at run time; everything is done at compile/link time.

Availability

Only available when the build option `OS_MAILBOX` is non-zero.

Arguments

`VarName`	Name of the variable holding the pointer to the mailbox descriptor to create / initialize. This is a variable name therefore do not put double quotes around the name.
`MbxName`	Mailbox name. This is not the variable name, it is the name attached to the mailbox. As it is a "C" string, the double quotes around the name are required. `G_OSnoName` , and not `NULL`, should be used for an unnamed mailbox.
`Size`	Maximum number of messages in the mailbox.

Returns

N/A

Component type

Macro (Unsafe)

Options

If the build option `OS_NAMES` is set to a value of zero, the argument `MbxName` is ignored but must still be supplied.

Notes

A mailbox created and initialized with `MBX_STATIC()` will not be part of the search done with `MBXopen()` or `MBXopenFCFS()`, unless another mailbox with the exactly the same name was created using `MBXopen()` or `MBXopenFCFS()`.
If mailboxes are created using the `MBX_STATIC()` component, it may not be possible to restart the application without reloading the binary image on the processor. This situation happens if the compiler (or compiler configuration) does not reload the initialized data upon start-up.
A mailbox created with this component is always created to operate in the *Priority* mode. If the build option `OS_FCFS` is non-zero and a mailbox created with `MBX_STATIC()` is targeted to operate in the *First Come First Served* mode, use the component `MBXsetFCFS()` on the mailbox in `main()`, once the component `OSstart()` has been used.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_NAMES` (Section 4.1.28)
`MBXopen()` (Section 6.7.7)
`MBXopenFCFS()` (Section 6.7.8)
`MBXsetFCFS()` (Section 6.7.11)
`G_OSnoName` (Section 6.14.2)

6.7.3 MBXabort

Synopsis

```
#include "Abassi.h"

void MBXabort(MBX_t *Mbox);
```

Description

`MBXabort()` is a component that unblocks all tasks that are blocked trying to either write in a mailbox that is full or trying to read a mailbox that is empty. The mailbox is specified with the argument `Mbox`. This is alike applying the component `TSKtimeoutKill()` on all the tasks blocked on the mailbox `Mbox` with the extra capability to also unblocked tasks blocked with an infinite timeout. `MBXabort()` has no effect on a task blocked on a group the mailbox is attached to.

Availability

`MBXabort()` is only available when the build option `OS_WAIT_ABORT` is defined and non-zero and the build option `OS_MAILBOX` is non-zero.

Arguments

`Mbox`	Descriptor of the mailbox to unblock all the tasks that are blocked on.

Returns

`void`

Component type

Atomic macro (safe)

Options

Notes

All tasks that blocked on the `MBXget()` or `MBXput()` components are unblocked when the component `MBXabort()` is used on the mailbox `Mbox`. The `MBXget()` or `MBXput()` components then return a non-zero value to indicate a timeout occurred, even if the wait time requested was infinite. The contents of the mailbox is left untouched once the component `MBXabort()` has been used.

`MBXabort()` applied on a mailbox that is attached to a group **will not** unblock a task waiting on that group. The reason is the task is blocked on the group, not blocked on the mailbox. If there are one or more tasks blocked on the same mailbox (non-group blocking is authorized on a mailbox already attached to a group), then these tasks will get unblocked.

If there are no tasks blocked on the mailbox `Mbox`, the use of `MBXabort()` on `Mbox` has no effects.

See also

`OS_WAIT_ABORT` (Section 4.1.62)
`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)

6.7.4 MBXavail

Synopsis

```
#include "Abassi.h"

int MBXavail(MBX_t *Mbox);
```

Description

`MBXavail()` is a component that reports how many elements in the mailbox that are not used. It's the number of free elements in the mailbox `Mbox`.

Availability

`MBXavail()` is only available when the build option `OS_MAILBOX` is non-zero in. It is not available in any releases before Abassi version 1.273.262 and mAbassi version 1.94.97.

Arguments

`Mbox`	Descriptor of the mailbox to report the number of free elements.

Returns

Number of free elements in `Mbox`.

Component type

Atomic macro (unsafe)

Options

Notes

See also

`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)
`MBXused()` (Section 6.7.12)

6.7.5 MBXget

Synopsis

```
#include "Abassi.h"

int MBXget(MBX_t *Mbox, intptr_t *Msg, int Timeout);
```

Description

`MBXget()` is used to retrieve a message from a mailbox. The oldest message (time-wise) held in the mailbox is retrieved and written to the location indicated by the argument `Msg`. When the mailbox is empty, it is possible to request to block until a message is deposited in the mailbox, or to block with timeout, or no blocking at all.

Availability

`MBXget()` is available only if the build option `OS_MAILBOX` is non-zero. Depending on the setting of the build option `OS_TIMEOUT`, the meaning of the argument `Timeout` slightly changes. See **Options** below.

Arguments

`Mbox`	Descriptor of the mailbox to read the message from.	
`Msg`	Pointer to the location the message is written into.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Number of timer ticks before expiry

Returns

0	The message in the location pointed by `Msg` is valid.
Non-zero	A new message was not available. This will occur if the argument `Timeout` is non-negative. Either `Timeout` was zero and there was no message, or `Timeout` was positive and no new message was deposited in the mailbox within `Timeout` number of timer ticks (or when the component `TSKtimeoutKill()` is applied to the task that is waiting on the mailbox). If an application ever has multiple readers for a mailbox, blocked readers will unblock in a priority ordering (or request order if FCFS) depending on the mode of operation of the mailbox.

Component type

Function.

Options

If the build option `OS_TIMEOUT` is zero, then when the argument `Timeout` is set to a positive value, `MBXget()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is a negative value, then when the argument `Timeout` is set to a positive value, `MBXget()` behaves the same as if the `Timeout` argument had been set to a negative value.

Notes

Unless `Timeout` is negative, always verify the return value: a non-zero value means the mailbox is empty, zero means a message was retrieved, even if the timeout was set to zero. When timeout is negative, the component `TSKtimeoutKill()` cannot unblock the task waiting to obtain a message, as an infinite timeout request does not involve the timer service.

`MBXget()` can be used in an ISR as this component has special hooks added to it for this purpose. When used in an ISR, the argument `Timeout` is ignored and is always considered being equal to zero. The return value for this component is valid in an ISR. Only one ISR handler can use `MBXget()` on a mailbox. If two or more ISR handlers apply the `MBXget()` component on the same mailbox, message losses or message duplication may occur.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_TIMEOUT` (Section 4.1.57)
`MBXopen()` (Section 6.7.7)
`MBXput()` (Section 6.7.9)
`TSKtimeoutKill()` (Section 6.3.30)

6.7.6 MBXnotFCFS

Synopsis

```
#include "Abassi.h"

void MBXnotFCFS(MBX_t *Mbox);
```

Description

`MBXnotFCFS()` is the component to use to configure a mailbox to operate in the *Priority* mode. The unblocking order of such a mailbox is always the highest priority task that is blocked. This mode of operation only applies to the tasks that write messages into the mailbox through `MBXput()` (Section 6.7.9), or in the not-obvious case when multiple readers are blocked through `MBXget()` (Section 6.7.5).

Availability

`MBXnotFCFS()` is only available when the build options `OS_MAILBOX` and `OS_FCFS` are both non-zero.

Arguments

`Mbox`	Descriptor of the mailbox to set the operation to the *Priority* mode.

Returns

`void`

Component type

Data access

Options

Notes

If the mailbox was already operating in the *Priority* mode, using this component has no effect on such a mailbox.
If the mailbox was operating in the *First Come First Served* mode, using this component will not re-order tasks that are currently blocked on the mailbox. Newly blocked tasks will be inserted in a *Priority* ordering amongst the already *First Come First Served* ordered blocked tasks. This means there may be a transient phase before the mailbox cold truly operates in a *Priority* mode.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MAILBOX` (Section 4.1.18)
`MBXopen()` (Section 6.7.7)
`MBXsetFCFS()` (Section 6.7.11)

6.7.7 MBXopen

Synopsis

```
#include "Abassi.h"

MBX_t *MBXopen(const char *Name, int Size);
```

Description

`MBXopen()` is the component to use to create a mailbox, and is also the component to use to obtain the descriptor of an already existing mailbox. When a mailbox is created with `MBXopen()`, it operates in the *Priority* mode (for the writers and reader).

Availability

`MBXopen()` is only available if the build option `OS_MAILBOX` and `OS_RUNTIME` are both non-zero.

Arguments

`Name`	Name of the mailbox to create or to obtain the descriptor of.
`Size`	Maximum number of messages the mailbox can hold.

Returns

Descriptor of the mailbox

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build when `OS_NAMES` is zero, all mailboxes are unnamed and every use of `MBXopen()` creates a new mailbox.
If the build option `OS_NAMES` is non-zero, then `MBXopen()` will either return the descriptor of an existing mailbox (previously created with `MBXopen()`or `MBXopenFCFS()`), or when no mailbox with the specified name exists, it will create a new mailbox. This approach makes the creation and opening of mailboxes run-time safe. If that feature were not part of the `MBXopen()` component, it would be imperative to either create the mailbox immediately at start-up, or to guarantee the first task (using the mailbox) to reach the running state is the one creating the mailbox. With the run-time safe feature, it does not matter which task is the first to use the mailbox.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed mailboxes. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.
Be aware, if the build option `OS_FCFS` is non-zero, when the mailbox already exists, there is no guarantee the mailbox is operating in a *Priority* mode, as it may have been created with `MBXopenFCFS()`or it may have been set to operate in *First Come First Served* mode with `MBXsetFCFS()`.
Also, if a mailbox already exists, the requested size may not be the size of the existing mailbox.
At any time, a mailbox operating in the *Priority* mode can be modified to operate in the *First Come First Served* mode by using the `MBXsetFCFS()` component when the build option `OS_FCFS` is non-zero.
Mailboxes created with the `MBX_STATIC()` component are not part of the search performed by `MBXopen()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MAILBOX` (Section 4.1.18)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`MBXget()` (Section 6.7.5)
`MBXopenFCFS()` (Section 6.7.8)
`MBXput()` (Section 6.7.9)
`MBXsetFCFS()` (Section 6.7.11)

6.7.8 MBXopenFCFS

Synopsis

```
#include "Abassi.h"

MBX_t *MBXopenFCFS(const char *Name, int Size);
```

Description

`MBXopenFCFS()` is the component to use to create a mailbox, and also the one to use to obtain the descriptor of an already existing mailbox. When a mailbox is created with `MBXopenFCFS()`, it operates in the *First Come First Served* mode (for the writers and reader).

Availability

`MBXopenFCFS()` is only available if the build options `OS_MAILBOX`, `OS_RUNTIME`, and `OS_FCFS` are all non-zero.

Arguments

`Name`	Name of the mailbox to create or to obtain the descriptor of.
`Size`	Maximum number of messages the mailbox can hold.

Returns

Descriptor of the mailbox

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In this kind of build of the Abassi RTOS, all mailboxes are unnamed and every use of `MBXopenFCFS()` creates a new mailbox.
If the build option `OS_NAMES` is non-zero, then `MBXopenFCFS()` will either return the descriptor of an existing mailbox (previously created with `MBXopen()` or `MBXopenFCFS()`) otherwise it will create a new mailbox. This approach makes the creation and opening of mailboxes run-time safe. If that feature were not part of the `MBXopenFCFS()` component, it would be imperative to either create the mailbox immediately at start-up, or to guarantee the first task (using the mailbox) to reach the running state is the one creating the mailbox. With the run-time safe feature, it does not matter which task is the first to use the mailbox.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed mailboxes. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.

When the mailbox already exists, there is no guarantee the mailbox is operating in a *First Come First Served* mode, as it may have been created with `MBXopen()`or it may have been set to operate in *Priority* mode with `MBXnotFCFS()`.

Also, if a mailbox already exists, the requested size may not be the size of existing mailbox.

At any time, a mailbox operating in the *First Come First Served* mode can be modified to operate in the *Priority* mode by using the `MBXnotFCFS()` component.

Mailboxes created with the `MBX_STATIC()` component are not part of the search performed by `MBXopenFCFS()`.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MAILBOX` (Section 4.1.18)
`OS_NAMES` (Section 4.1.28)
`OS_RUNTIME` (Section 4.1.37)
`MBXget()` (Section 6.7.5)
`MBXnotFCFS()` (Section 6.7.6)
`MBXopen()` (Section 6.7.7)
`MBXput()` (Section 6.7.9)

6.7.9 MBXput

Synopsis

```
#include "Abassi.h"

int MBXput(MBX_t *Mbox, intptr_t Msg, int Timeout);
```

Description

`MBXput()` is used to deposit a message into a mailbox. When the mailbox is full, it is possible to request to block until there is room in the mailbox, or to block with timeout, or no blocking at all.

Availability

`MBXput()` is available if the build option `OS_MAILBOX` is non-zero.

Arguments

`Mbox`	Descriptor of the mailbox to put the message in.	
`Msg`	Message to insert in the mailbox.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Timer ticks before expiry

Returns

0	The message was added in the mailbox.
Non-zero	The message was not added in the mailbox. This will occur if the argument `Timeout` is non-negative. Either `Timeout` was zero and there was no room in the mailbox to add the message, or `Timeout` was positive and no room became available in the mailbox within `Timeout` number of timer ticks (or when the component `TSKtimeoutKill()` is applied to the task that waits on the mailbox). The room availability is dependent on the priority ordering (or request order if FCFS) of the writing task when one or more tasks are blocked

Component type

Inline function

Options

If the build option `OS_TIMEOUT` is zero, then when the argument `Timeout` is set to a positive value, `MBXput()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is a negative value, then when the argument `Timeout` is set to a positive value, `MBXput()` behaves the same as if the `Timeout` argument had been set to a negative value.

Notes

The message `Msg` is copied into the mailbox. But if `Msg` is a pointer, the pointer is copied but the memory pointed by `Msg` is NOT copied.

Unless `Timeout` is negative, always verify the return value: a non-zero value means the mailbox is full and the message hasn't been written, zero means a message was written, even if the timeout was set to zero.
When timeout is negative, the component `TSKtimeoutKill()` cannot unblock the task waiting to write a message, as an infinite timeout request does not involve the timer service.

`MBXput()` can safely be used in an ISR as this component was specially modified to ignore the value of the argument `Timeout` when applied in an ISR. The return value when this component is used in an ISR is always zero, no matter if the mailbox is full or not.

It is possible to make `MBXput()`, when called in an ISR, return the information if the mailbox is full or not. This is achieved by setting the build option `OS_MBXPUT_ISR` to a non-zero value and applying the component `MBXputInISR()` on the mailbox right after creation/opening.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_MBXPUT_ISR` (Section 4.1.22)
`OS_TIMEOUT` (Section 4.1.57)
`MBXget()` (Section 6.7.5)
`MBXopen()` (Section 6.7.7)
`MBXopenFCFS()` (Section 6.7.8)
`MBXputInISR()` (Section 6.7.10)
`TSKtimeoutKill()` (Section 6.3.30)

6.7.10 MBXputInISR

Synopsis

```
#include "Abassi.h"

void MBXputInISR(MBX_t *Mbox);
```

Description

`MBXputInISR()` is used to configure a mailbox to return a valid result when called from within an interrupt handler. It will report if the mailbox is not full, and the message will land in the mailbox, or if the mailbox is full, and the message will not land in the mailbox.

Availability

`MBXputInISR()` is available if the build option `OS_MBX_ISR` is defined and non-zero.

Arguments

`Mbox`	Descriptor of the mailbox to enable the feature.

Returns

`void`

Component type

Macro (safe)

Options

Notes

This component is only available when the build option `OS_MAILBOX` is non-zero and the build option `OS_MBXPUT_ISR` is defined and set to a non-zero value. When `OS_MBXPUT_ISR` is non-zero, none of the mailboxes upon creation (or start-up for statically defined mailboxes) have the `MBXput()` in an ISR return full/not full feature enabled. Each mailbox must have this feature individually enabled, through `MBXputInISR()`, to have `MBXput()` used in an ISR return if the mailbox is full or not.

The component `MBXputInISR()` **MUST** be used immediately after the creation/opening of the mailbox, before the mailbox is ever used. If this condition is not fulfilled, the operation of the mailbox will most likely be faulty, There is no way to recover from a faulty set-up.

The reason to make the application enable the validation of the return value of `MBXput()` in an ISR is to optimize the real-time performance and minimize the duration interrupts are disabled. When `MBXputInISR()` is applied to a mailbox, it drastically changes the behavior of the components `MBXget()` and `MBXput()` used on that mailbox: interrupts are constantly getting disable/restored (plus a spinlock lock/unlock in multi-core) when any of these two components are used on the mailbox. Because the disabling/restoring of the interrupts directly impacts the interrupt response time of Abassi, by only enabling the feature on the mailboxes that require it reduces the number of times the interrupts are disabled/restored, minimizing the impact on the interrupt response time.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_MBXPUT_ISR` (Section 4.1.22)
`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)
`MBXopen()` (Section 6.7.7)
`MBXopenFCFS()` (Section 6.7.8)

6.7.11 MBXsetFCFS

Synopsis

```
#include “Abassi.h”

void MBXsetFCFS(MBX_t *Mbox);
```

Description

`MBXsetFCFS()` is the component to use to configure a mailbox to operate in the *First Come First Served* mode. The unblocking order of such a mailbox is always the oldest task that was blocked is unblocked first. This mode of operation only applies to the tasks that write messages into the mailbox through `MBXput()` (Section 6.7.9), or in the not-obvious case when multiple readers are blocked through `MBXget()` (Section 6.7.5).

Availability

`MBXsetFCFS()` is only available when the build options `OS_MAILBOX` and `OS_FCFS` are both non-zero.

Arguments

`Mbox`	Descriptor of the mailbox to set into a *First Come First Served* mode.

Returns

`void`

Component type

Definition

Options

Notes

If the mailbox was already operating in the *First Come First Served* mode, using this component has no effect on such a mailbox.
If the mailbox was operating in the *Priority* mode, using this component will not re-order tasks that are currently blocked on the mailbox. Newly blocked tasks will be inserted in a *First Come First Served* ordering amongst the already *Priority* ordered blocked tasks. This means there may be a transient phase before the mailbox truly operates in a *First Come First Served* mode.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MAILBOX` (Section 4.1.18)
`MBXget()` (Section 6.7.5)
`MBXnotFCFS()` (Section 6.7.6)
`MBXopen()` (Section 6.7.7)
`MBXput()` (Section 6.7.9)

6.7.12 MBXused

Synopsis

```
#include "Abassi.h"

int MBXused(MBX_t *Mbox);
```

Description

`MBXavail()` is a component that reports how many elements in held in the mailbox `Mbox`.

Availability

`MBXused()` is only available when the build option `OS_MAILBOX` is non-zero in. It is not available in any releases before Abassi version 1.273.262 and mAbassi version 1.94.97.

Arguments

`Mbox`	Descriptor of the mailbox to report the number of elements in use.

Returns

Number of elements held in `Mbox`.

Component type

Atomic macro (unsafe)

Options

Notes

See also

`MBXavail()` (Section 6.7.4)
`MBXget()` (Section 6.7.5)
`MBXput()` (Section 6.7.9)

6.7.13 Examples

The mailbox component queue is capable of holding a maximum number of elementary data types, which can be either an integer or a pointer. If pointers are passed through a mailbox, it is quite straightforward to create a data queue to hold a number of buffers. The following example explains how to handle a data queue and mainly explains how to allocate enough buffers to not have buffer trampling by the queue reader(s) and writer(s).

6.7.13.1 Buffers

The mailboxes are designed to hold `intptr_t` type of data. This means the elements can be either an integer or a pointer. To create a data queue, pointers to buffers are used. One must remember only the pointer, not the buffer contents, is copied and held inside the mailbox during the exchanges between the writer(s) and reader(s). When working with a pre-allocated set of buffers, e.g. using Abassi's memory block component, there is a minimum number of buffers required to not run out of buffers before the mailbox becomes full.

6.7.13.2 Buffer count

When a mailbox is dimensioned to size `S`, it means the mailbox can hold a maximum of `S` elements. So at first there is a minimum of `S` buffers required to not starve the filling of the mailbox. In addition to the mailbox holding capability, it is desirable to allow the writer(s) to prepare new buffers to be sent through the mailbox even when the mailbox is full. This pipelines the mailbox filling, which helps reduce the processing latency. If extra buffer(s) were not available for the writer(s) when the mailbox is full, the writer task(s) would then have to block waiting for the availability of a new buffer. Once it gets the buffer it would fill it and send it through the mailbox. The latency added in this case is the processing time involved between the availability of the buffer and the sending to the mailbox. Extra buffer(s) should also be allocated to allow the reader to process the received buffer. In this case, the purpose of extra buffer(s) for the reader(s) is to maximize the holding capability of the mailbox. If there are no extra buffer(s) allocated for the reader(s), then when a reader is using a buffer, before it returns it to the pool of buffers, it would possibly starve the writer(s) from being capable of processing buffers in advance.

So a mailbox sized to `S` elements should be associated with the following number of buffers:

```
Nblk = S + Nrd + Nwrt
```

Where:

`Nblk` :	Number of buffers required to not starve the mailbox operations
`S` :	Size of the mailbox (as set when using the `MBXopen()` component)
`Nrd` :	Total number of tasks that can read from the mailbox
	This includes an interrupt if the mailbox is read in an interrupt
`Nwrt` :	Total number of tasks that can write to the mailbox
	This includes the number of interrupts if the mailbox is written in interrupts

6.7.13.3 Example code

The following example shows how to use a mailbox to create a data queue. This example relies on the run-time safe creation feature of Abassi, meaning the build option `OS_NAMES` must be set to a non-zero value.

The example code assumes only one task writes to the mailbox and only one task reads the mailbox. As previously explained, the optimal number of buffer to be made available is the mailbox size + 2.

The writer task simply performs the following:

- Create / open (run-time creation safe) the mailbox to write to
- Create / open (run-time creation safe) the memory block management to use
- Infinite loop
    - Get a new buffer
    - Process and fill the buffer
    - Send the buffer through the mailbox

As the memory block management service is created with 2 more buffers than the mailbox can hold, the call to the service `MBLKalloc()` can never block the task. The task will only block if the mailbox is full.

Table 6-13 Queue writer code

```
#include "MyApp.h"

void WriterTask(void)
{
int    Buffer;                                  /* Buffer to process / queued     */
MBX_t  MyMbx;                                   /* Mailbox written to             */
MBLK_t MyPool;                                  /* Memory block pool used here    */

                                                /* Create / open the mailbox      */
  MyMbx = MBXopen(MBX_DATA1_NAME, MBX_DATA1_NBUF);
  if (MyMbx == (MBX_t *)NULL) {
     ….                                         /* ERROR                          */
  }
                                                /* Create / open the memory block */
  MyPool = MBLKopen(MBLK_DATA1_NAME, MBX_DATA1_NBUF+2, MBX_DATA1_BSIZE*sizeof(int));
  if (MyPool == (MBLK_t *)NULL) {
     …                                          /* ERROR                          */
  }

  for (;;) {
     Buffer = MBLKalloc(MyPool, -1);            /* Get a new buffer to fill       */

     …                                          /* PROCESSING                     */

     MBXput(MyMbx, (intptr_t )Buffer, -1);      / Send it to the mailbox          */
  }
}
```

The reader task simply performs the following:

- Create / open (run-time creation safe) the mailbox to read from
- Create / open (run-time creation safe) the memory block management to use
- Infinite loop
    - Get a new buffer from the mailbox
    - Process and fill the buffer
    - Return the buffer to the pool of memory

Table 6-14 Queue reader code

```
#include “MyApp.h”

void ReaderTask(void)
{
int    Buffer;                                  /* Buffer to process / queued     */
MBX_t  MyMbx;                                   /* Mailbox read from              */
MBLK_t MyPool;                                  /* Memory block pool used here    */

                                                /* Create / open the mailbox      */
  MyMbx = MBXopen(MBX_DATA1_NAME, MBX_DATA1_NBUF);
  if (MyMbx == (MBX_t *)NULL) {
      ….                                        /* ERROR                          */
  }
                                                /* Create / open the memory block */
  MyPool = MBLKopen(MBLK_DATA1_NAME, MBX_DATA1_NBUF+2, MBX_DATA1_BSIZE*sizeof(int));
  if (MyPool == (MBLK_t *)NULL) {
      …                                         /* ERROR                          */
  }

  for (;;) {
                                                / Read fromt the mailbox          */
      Buffer = (int *)MBXget(MyMbx, (intptr_t )Buffer, -1);

      …                                         /* PROCESSING                     */

      MBLKfree(MyPool, Buffer);                 /* Return the buffer to the pool  */
  }
}
```

NOTE: Although the example uses an integer buffer, nothing prevents the application from exchanging a data structure alike:

Table 6-15 Data structure buffers

```
typedef struct {
  int Size;                                     /* # of valid entries in Buffer   */
  int Buffer[10];                               /* Data buffer                    */
} Xchg_t;


  …


Xchg_t Data;                                    / Local data structure pointer    */


  MyPool = MBLKopen(MBLK_DATA1_NAME, MBX_DATA1_NBUF+2, MBX_DATA1_BSIZE*sizeof(Xchg_t));

  …

  MBXput(MyMbx, (intptr_t )Data, -1);

  …

  Data = (Xchg_t *)MBXget(MyMbx, (intptr_t )Buffer, -1);

  …
```

Timer Components

6.7.14 OS_TIM_TICK_ACK

Synopsis

```
#ifdef OS_TIM_TICK_ACK
   OS_TIM_TICK_ACK;
#endif
```

Description

`OS_TIM_TICK_ACK` is an optional pre-processor statement that can be used when the RTOS timer tick interrupt source needs an acknowledgement operation. The code described in the **Synopsis** is what is implemented in the timer tick interrupt handler.

Availability

Optional

Arguments

`N/A`

Returns

`N/A`

Component type

Pre-processor definition

Options

Notes

The definition for `OS_TIM_TICK_ACK` must be inserted in the file `Abassi.h` or defined on the compiler command line.

See also

`OS_TIMER_US` (Section 4.1.59)

6.7.15 G_OStimCnt

Synopsis

```
#include "Abassi.h"

int G_OStimCnt;
```

Description

`G_OStimCnt` is a counter incremented once every timer tick.

Availability

Only available when the build option `OS_TIMER_US` is non-zero.

Arguments

`N/A`

Returns

`N/A`

Component type

Variable

Options

Notes

This variable must always be accessed in a read-only fashion. If `G_OStimCnt` is modified by the application, components that rely on the timer service may not operate correctly.
The variable is not an `unsigned int`, but an `int`. The `int` data type was selected because of the way the internal timeout service operates.

See also

`OS_TIMER_US` (Section 4.1.59)

6.7.16 TIMcallBack

Synopsis

```
#include "Abassi.h"

void TIMcallBack(void);
```

Description

`TIMcallBack()` is a function the application supplies to the Abassi RTOS. It is the function the timer calls at a pre-programmed rate, defined by the build option `OS_TIMER_CB`.

Availability

`TIMcallBack()` is only needed if the build options `OS_TIMER_CB` and `OS_TIMER_US` are non-zero.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

It is important to remember this function is called within the timer interrupt handler, so it is operating in an interrupt context.

See also

`OS_TIMER_CB` (Section 4.1.58)
`OS_TIMER_US` (Section 4.1.59)

6.7.17 Time Converters

The time converters are components that convert back and forth hour - minute - second time units and time ticks units. If the build option `OS_TIMER_US` (Section 4.1.59) is zero, none of the time converters are available.

6.7.17.1 OS_HAS_TIMEDOUT

Synopsis

```
#include "Abassi.h"

int OS_HAS_TIMEDOUT(int TickCount);
```

Description

`OS_HAS_TIMEDOUT()` returns the information if the RTOS timer tick counter has reached or has exceeded the count specified by the argument `TickCount`. This component is normally used with the component `OS_TICK_EXPIRY`.

Availability

Only available when the build option `OS_TIMER_US` is non-zero.

Arguments

`TickCount`	Timer tick counter value for which a time out is declared.

Returns

0	The timer tick counter value has not exceeded the value of `TickCount`.
Non-zero	The timer tick counter value is equal to or exceeded the value of `TickCount`.

Component type

Macro (safe)

Options

Notes

The component `OS_HAS_TIMEDOUT` does not handle infinite timeout.

See also

`OS_TICK_EXPIRY` (Section 6.7.17.6)

6.7.17.2 OS_HMS_TO_TICK

Synopsis

```
#include "Abassi.h"

int OS_HMS_TO_TICK(int nHours, int nMins, int nSecs);
```

Description

`OS_HMS_TO_TICK()` converts a time value indicated in hours, minutes and seconds into the equivalent number of timer tick units.

Availability

Only available when the build option `OS_TIMER_US` is non-zero.

Arguments

N/A

Returns

N/A

Component type

Macro (safe)

Options

Notes

In Abassi version 1.266.247, mAbassi version 1.85.85 and above, if the resulting number of seconds is negative, the component `OS_HMS_TO_TICK()` returns a negative value because a negative value is considered an infinite time in Abassi.

See also

`OS_TIMER_US` (Section 4.1.59)

6.7.17.3 OS_MS_TO_TICK

Synopsis

```
#include "Abassi.h"

int OS_MS_TO_TICK(int nMs);
```

Description

`OS_MS_TO_TICK()` is the component that converts a time value expressed in milliseconds into its equivalent number of timer ticks. The conversion is rounded to the nearest integer.

Availability

Only available when the build option `OS_TIMER_US` is non-zero.

Arguments

`nMs`	Millisecond value to convert in timer tick count.

Returns

The number of timer ticks during `nMs` milliseconds.

Component type

Macro (safe)

Options

Notes

The conversion from milliseconds to the number of timer ticks is rounded to the nearest integer, not ceiled toward the higher integer. Therefore, if the argument `nMs` is smaller than half the value of the build option `OS_TIMER_US`, which is expressed in microseconds, then a value of zero will be returned.
When the Abassi RTOS is operating on a processor/compiler port that uses 16 bit for the data type `int`, the dynamic range is definitely restricted.
In Abassi version 1.266.247, mAbassi version 1.85.85 and above, if the argument `nMs` is negative, the component `OS_MS_TO_TICK()` returns a negative value because a negative value is considered an infinite time in Abassi.

See also

`OS_TIMER_US` (Section 4.1.59)
`OS_HMS_TO_TICK` (Section 6.7.17.2)
`OS_SEC_TO_TICK` (Section 6.7.17.5)
`OS_TICK_PER_SEC` (Section 6.7.17.7)

6.7.17.4 OS_MS_TO_MIN_TICK

Synopsis

```
#include "Abassi.h"

int OS_MS_TO_MIN_TICK(int nMs, int MinTick);
```

Description

`OS_MS_TO_MIN_TICK()` is performing the same conversion as `OS_MS_TO_TICK` except it floors the returned value (number of ticks) to `MinTick`. This component is useful for example when a short minimum delay is required. For example, if the `nMs` value translates into 0 ticks when at least two timer ticks are required, one should set `MinTick` to 2.

Availability

Only available when the build option `OS_TIMER_US` is non-zero.

Arguments

`nMs`	Millisecond value to convert in timer tick count.
`MinTick`	Minimum number of timer tick required.

Returns

The number of timer ticks during `nMs` milliseconds if greater than `MinTick`, otherwise `MinTick` is returned.

Component type

Macro (unsafe)

Options

Notes

The conversion from milliseconds to the number of timer ticks is rounded to the nearest integer, not ceiled toward the higher integer. Therefore, if the argument `nMs` is smaller than half the value of the build option `OS_TIMER_US`, which is expressed in microseconds, then a value of zero will be returned.
When the Abassi RTOS is operating on a processor/compiler port that uses 16-bit for the data type `int`, the dynamic range is definitely restricted.
In Abassi version 1.266.247, mAbassi version 1.85.85 and above, if any of the 2 arguments is negative, the component `OS_MS_TO_MIN_TICK()` returns a negative value because a negative value is considered an infinite time in Abassi.

See also

`OS_TIMER_US` (Section 4.1.59)
`OS_HMS_TO_TICK` (Section 6.7.17.2)
`OS_SEC_TO_TICK` (Section 6.7.17.5)
`OS_TICK_PER_SEC` (Section 6.7.17.7)

6.7.17.5 OS_SEC_TO_TICK

Synopsis

```
#include "Abassi.h"

int OS_SEC_TO_TICK(int nSec);
```

Description

`OS_SEC_TO_TICK()` is the component that converts a time value expressed in second into its equivalent of number of timer ticks. The conversion is rounded to the nearest integer

Availability

Only available when the build option `OS_TIMER_US` is non-zero

Arguments

`nSec`	Second value to convert in timer tick count

Returns

The number of timer ticks during `nSec` seconds.

Component type

Macro (safe)

Options

Notes

The conversion from seconds to the number of timer ticks is rounded to the nearest integer, not ceiled toward the higher integer. It is quite unlikely (because most embedded applications typically rely on timer with periods smaller than 1 second), but if the argument `nSec` is smaller than half the value of the build option `OS_TIMER_US`, which is expressed in microseconds, then a value of zero will be returned.
When using the Abassi RTOS is operating on a processor / compiler port that uses 16 bit for the data type `int`, the range is definitely restricted.
In Abassi version 1.266.247, mAbassi version 1.85.85 and above, if the argument `nMs` is negative, the component `OS_SEC_TO_TICK()` returns a negative value because a negative value is considered an infinite time in Abassi.

See also

`OS_TIMER_US` (Section 4.1.59)
`OS_HMS_TO_TICK` (Section 6.7.17.2)
`OS_MS_TO_TICK` (Section 6.7.17.3)
`OS_TICK_PER_SEC` (Section 6.7.17.7)

6.7.17.6 OS_TICK_EXPIRY

Synopsis

```
#include "Abassi.h"

int OS_TICK_EXPIRY(nMs, MinTick);
```

Description

`OS_TICK_EXPIRY` is the component returning what will be the timer tick value in `nMs` milli-seconds, with a minimum of `MinTicks`. This component is normally used with the component `OS_HAS_TIMEDOUT`.

Availability

Only available when the build option `OS_TIMER_US` is non-zero

Arguments

`nMs`	Expiry time in milli-seconds.
`MinTick`	Minimum number of timer tick required before expiry.

Returns

RTOS timer tick counter value where an expiry is declared.

Component type

Macro (unsafe)

Options

Notes

This macro performs the following calculations:

```
G_OStimCnt + MIN(MinTick, OS_MS_TO_MIN_TICK(nMs))
```

The component `OS_TICK_EXPIRY` does not handle infinite timeout. If any of the argument is negative, the result will be an immediate timeout validation when used with the component `OS_HAS_TIMEDOUT().`

See also

`OS_TIMER_US` (Section 4.1.59)
`OS_HAS_TIMEDOUT` (Section 6.7.17.1)
`OS_HMS_TO_TICK` (Section 6.7.17.2)
`OS_MS_TO_TICK` (Section 6.7.17.3)
`OS_SEC_TO_TICK` (Section 6.7.17.5)

6.7.17.7 OS_TICK_PER_SEC

Synopsis

```
#include "Abassi.h"

int OS_TICK_PER_SEC;
```

Description

`OS_TICK_PER_SECOND` is a definition that gives access to the application on the number of timer tick there is per second. This is not a function, but a constant value.

Availability

Only available when the build option `OS_TIMER_US` is non-zero

Arguments

`N/A`

Returns

`N/A`

Component type

Definition

Options

Notes

The conversion from the number of timer ticks to a time in seconds is rounded to the nearest integer, not ceiled toward the higher integer. If the build option `OS_TIMER_US` is set to value larger than 499,999, then a timer count of zero will be returned.
The build option `OS_TIMER_US` may not be an exact fraction of 1 second, therefore it is highly advisable to use the `OS_SEC_TO_TICK()` component to obtain the number of timer tick for a value different than 1 second.
The same applies for fractions of a second, `OS_MS_TO_TICK()` is the preferred component to use.

See also

`OS_TIMER_US` (Section 4.1.59)
`OS_HMS_TO_TICK` (Section 6.7.17.2)
`OS_MS_TO_TICK` (Section 6.7.17.3)
`OS_SEC_TO_TICK` (Section 6.7.17.5)

6.7.18 Examples

6.7.18.1 Periodic Timer

The Abassi RTOS does not provide any components for periodic operations. However, the timer callback facility delivers more powerful access to periodic operations. Instead of providing dedicated components for periodic semaphore posting, mutex posting, event setting, or mailbox writing, the timer callback facility is fully open to the application to be used for that. Any component can be used in the timer callback function (excluding those that can’t be used in an interrupt context, refer to Section 3.3.2).

6.8 Interrupt Components

The Abassi RTOS provides an easy to use component to attach a function to a source of interrupts on a processor. All there is to do is to use the `OSisrInstall()` component for each of the interrupts the application needs to handle.

Another required feature in a multi-tasking application is the capability to disable and re-enable interrupts to protect critical regions in the application; the Abassi RTOS supports this with the `OSintOFF()`, `OSintOn()` and `OSintBack()` components. The legacy components `OSdint()` and `OSeint()` can also be used for the interrupt control but are not as real time efficient as the three others.

6.8.1 OSdint

Synopsis

```
#include "Abassi.h"

int OSdint(void);
```

Description

`OSdint()` is the legacy component to use to disable the interrupts. If the interrupts are already disabled, `OSdint()` has no effect.

Availability

Always

Arguments

`void`

Returns

State of the interrupts before the use of the component `OSdint()`.

Component type

Definition involving a function

Options

Notes

Do NOT use the return value of `OSdint()` with `OSintBack()` as they are not compatible.
Enabling and disabling the interrupts must always be performed through the `OSdint()` and `OSeint()` components. Not doing so can create issues on some processor/compiler ports.
As `OSdint()` returns the previous state of the interrupt enable, the proper way to use the pair `OSdint()` and `OSeint()` is:

Table 6-16 Proper way to use OSdint() and OSeint()

```
ISRstate = OSdint();

...

OSeint(ISRstate);
```

This allows multiple level calls with multiple disabling/enabling at the different levels to properly put back the interrupt enable state without having to keep track of the state of the interrupt enable at the highest level.

See also

`OSeint()` (Section 6.8.2)

6.8.2 OSeint

Synopsis

```
#include "Abassi.h"

void OSeint(int ISRstate);
```

Description

`OSeint()` is the legacy component to use to enable/disable the interrupts. The argument `ISRstate` is a Boolean specifying if the interrupts must be disabled (value of zero) or enabled (non-zero value).

Availability

Always

Arguments

`ISRstate`	Boolean	
	0	Disable all interrupts
	non-zero	Enable all interrupts

Returns

`void`

Component type

Function

Options

Notes

Do NOT use the return value of `OSintOff()` with `OSeint()` as they are not compatible.

Enabling and disabling the interrupts must always be performed through the `OSdint()` and `OSeint()` components. Not doing so can create issues on some processor/compiler ports. As `OSdint()` returns the previous state of the interrupt enable, the proper way to use the pair `OSdint()` and `OSeint()` is:

Table 6-17 Proper way to use OSdint() and OSeint()

```
ISRstate = OSdint();

...

OSeint(ISRstate);
```

This allows multiple level calls with multiple disabling/enabling at the different levels to properly put back the interrupt enable state without having to keep track of the state of the interrupt enable at the highest level.
The only time when a pre-defined value should be used is at start-up to enable the interrupts, since, upon boot-up, interrupts are always disabled.

See also

`OSdint()` (Section 6.8.1)

6.8.3 OSintBack

Synopsis

```
#include "Abassi.h"

void OSintBack(int ISRstate);
```

Description

`OSintBack()` is the component to use to restore the interrupt state (re-enable them or keep them disabled). The argument `ISRstate` is processor specific and it must be a previous return value from the component `OSintOff()`.

Availability

Always

Arguments

`ISRstate`	processor specific value returned by `OSintOff()`.

Returns

`void`

Component type

Function

Options

Notes

Do NOT use the return value of `OSdint()` with `OSintBack()` as they are not compatible.

Enabling and disabling the interrupts must always be performed through the `OSintOff()` and `OSintBack()` components. Not doing so can create issues on some processor/compiler ports.
As `OSintOff()` returns the previous state of the interrupt enable/disable (processor specific), the proper way to use the pair `OSintOff()` and `OSintBack()` is:

Table 6-18 Proper way to use OSintOff() and OSintBack()

```
ISRstate = OSintOff();

...

OSintBack(ISRstate);
```

This allows multiple level calls with multiple disabling/enabling at the different levels to properly put back the interrupt enable state without having to keep track of the state of the interrupt enable at the highest level.

See also

`OSintOff()` (Section 6.8.4)
`OSintOn()` (Section 6.8.5)

6.8.4 OSintOff

Synopsis

```
#include "Abassi.h"

int OSintOff(void);
```

Description

`OSintOff()` is the component to use to disable the interrupts. If the interrupts are already disabled, `OSintOff()` has no effect.

Availability

Always

Arguments

`void`

Returns

Processor specific information of the state of the interrupt enable/disable before the use of the component `OSintOff()`.

Component type

Function

Options

Notes

Do NOT use the return value of `OSintOff()` with `OSeint()` as they are not compatible.
Enabling and disabling the interrupts must always be performed through the `OSintOff()` and `OSintBack()` components. Not doing so can create issues on some processor/compiler ports.
As `OSintOff()` returns the previous state of the interrupt enable, the proper way to use the pair `OSintOff()` and `OSintBack()` is:

Table 6-19 Proper way to use OSintOff() and OSintBack()

```
ISRstate = OSintOff();

...

OSintBack(ISRstate);
```

This allows multiple level calls with multiple disabling/enabling at the different levels to properly put back the interrupt enable state without having to keep track of the state of the interrupt enable at the highest level.

See also

`OSintBack()` (Section 6.8.3)
`OSintOn()` (Section 6.8.5)

6.8.5 OSintOn

Synopsis

```
#include “Abassi.h”

void OSintOn(void);
```

Description

`OSintOn()` is the component to use to force-enable the interrupts

Availability

Always

Arguments

`void`

Returns

`void`

Component type

Function

Options

See also

`OSintBack()` (Section 6.8.3)
`OSintOff()` (Section 6.8.4)

6.8.6 OSisrInstall

Synopsis

```
#include "Abassi.h"

void OSisrInstall(int ISRnmb, void(*Fct)(void));
```

Description

`OSisrInstall()` is the component to use to attach an interrupt handler to source of interrupt.

Availability

Always

Arguments

`ISRnmb`	Vector number of the interrupt to attach to the function `Fct`.
`Fct`	Pointer to the function to use on the interrupt number `ISRnmb`.

Returns

`void`

Component type

Macro (safe)

Options

Notes

The function `Fct` is always a regular "C" function; there should not be non-standard syntax such as `_interrupt` or `#pragma interrupt`, etc.
Refer to the port document for the target compiler/processor to get more information on the meaning of the arguments of this component.

If Abassi is used in a C++ environment, the function attached to the task (the argument `Fct`) must be declared with "C" linkage (see section 3.5), not as a regular C++ function.

See also

6.9 Timer Services

The timer services are part of an optional module, included in the build when the build option `OS_TIMER_SRV` is set to a non-zero value. The timer services offer a simple way to get an operation performed after a selected delay or to have an operation periodically performed. Some of the above operations, when periodic, have or require an argument. It is possible to modify the argument each time the operation is performed by post-adding a programmable offset. The six operations supported in the timer services are:

- Writing data to a memory location
- Setting event flags of a task
- Calling a function with an argument
- Writing a value to a mailbox
- Unlocking a mutex
- Posting a semaphore

The following table lists the timer services components:

Table 6-20 Timer Services list

Section	Name	Description
6.9.1	`TIM_STATIC`	Create a timer at compile / link time
6.9.2	`TIMarg`	Modify the argument used by the timed operation
6.9.3	`TIMdata`	Delayed and periodic writing at a memory location
6.9.4	`TIMevt`	Delayed and periodic event flag setting
6.9.5	`TIMfct`	Delayed and periodic function call
6.9.6	`TIMfreeze`	Stop/hold an active timer service
6.9.7	`TIMkill`	Terminate an active timer service
6.9.8	`TIMleft`	Report the time left of an active timer service
6.9.9	`TIMmbx`	Delayed and periodic mailbox writing
6.9.10	`TIMmtx`	Delayed and periodic mutex unlocking
6.9.11	`TIMopen`	Create a timer service / obtain the descriptor of a timer service
6.9.12	`TIMpause`	Pause the timer operation performed upon expiry
6.9.13	`TIMperiod`	Modify the period of an active timer
6.9.14	`TIMrestart`	Restart a timer expiries as they were originally set-up
6.9.15	`TIMresume`	Resume the timer operation performed upon expiry if paused
0	`TIMsem`	Delayed and periodic semaphore posting
6.9.17	`TIMtoAdd`	Modify the value to add to the argument of a timed operation

To be usable, a timer must be first be created/opened with the component `TIMopen()`, or it must have been created a compile time with the `TIM_STATIC()` component. All there is to do afterward to activate the timer is to use one the following components, to either indicate the delay before the operation is performed and the periodicity, if desired:

- `TIMdata()`
- `TIMevt()`
- `TIMfct()`
- `TIMmbx()`
- `TIMmtx()`
- `TIMsem()`

All six timer operations are programmed the same way. One argument indicates the expiry time of the timer in number of timer ticks. When the desired time has elapsed, the operation is performed. A second argument indicates the periodicity of the timer operation. If the desired period is set to zero or a negative value, the timer becomes inactive after the first expiry. If the period is positive, then the selected operation will be performed every *period* number of timer tick, after the first expiry. Some of the operations use an argument; when the timer is periodic, the argument is modified, by adding a "*To Add*" value. Therefore, to keep the same argument at every period timer ticks, the "*To Add*" value must be set to zero; otherwise, the argument is updated adding the "*To Add*" value after each time the timer triggers the selected operation. At any time, an active timer can be deactivated with the component `TIMkill()`, or its operating parameters modified with the components `TIMarg()`, `TIMperiod()` and `TIMtoAdd()`.

If the target application has limited data or code memory, it is highly recommended to not enable/use the timer services. Instead, enable the timer callback facility and add in the timer callback function the desired functionality. The reason why the timer services are not as memory efficient as the timer callback solution is simply due to the fact the timer services offer a generic interface/functionality. As with any generic code, the code and memory is always less efficiently used than in a custom solution.

6.9.1 TIM_STATIC

Synopsis

```
#include "Abassi.h"

TIM_STATIC(VarName, TimName);
```

Description

`TIM_STATIC()` is a special component that creates a timer and initializes its descriptor. It is a macro definition creating a static object, so none of the arguments has a real data type. The timer is not created/initialized at run time; everything is done at compile/link time.

Availability

Only when the build option `OS_TIMER_SRV` is non-zero.

Arguments

`VarName`	Name of the variable holding the pointer to the timer descriptor to create / initialize. This is a variable name, therefore do not put double quotes around the name.
`TimName`	Timer name. This is not the variable name, it is the name attached to the timer. As it is a "C" string, the double quotes around the name are required. `G_OSnoName` , and not `NULL`, should be used for an unnamed timer.

Returns

N/A

Component type

Macro (safe)

Options

If the build option `OS_NAMES` is set to a value of zero, the argument `TimName` is ignored but must still be supplied.

Notes

A timer created and initialized with `TIM_STATIC()` will not be part of the search done with `TIMopen()`, unless another timer with the exactly the same name was created using `TIMopen()`. Then the descriptor of that other timer, the one that was not created statically, will be returned.

See also

`OS_NAMES` (Section 4.1.28)
`OS_TIMER_SRV` (Section 4.1.59)
`TIMopen()` (Section 6.9.11)
`G_OSnoName` (Section 6.14.2)

6.9.2 TIMarg

Synopsis

```
#include "Abassi.h"

void TIMarg(TIM_t *Timer, intptr NewArg);
```

Description

`TIMarg()` is a timer service component that modifies the argument of some but not all of the timer services operations. It allows the application to replace the argument once the timer has been activated. The timer services components that use arguments and can have `TIMarg()` applied to are: `TIMdata()`, `TIMevt()`, `TIMfct()`, `TIMmbx()`.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero.

Arguments

`Timer`	Descriptor of the timer service to replace the argument of the operation
`NewArg`	New value of operation argument

Returns

void

Component type

Data access

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the argument value set when using `TIMarg()` will be overloaded when the timer is activated with `TIMdata()`, or `TIMevt()`, or `TIMfct()`, or `TIMmbx()`.
There is no side effect if the component `TIMarg()` is applied on an inactive timer or a timer that was activated to perform an operation which does not requires an argument, such as `TIMmtx()` and `TIMsem()`. The new value of the timer operation argument is the argument that will be used the next time the timer expires; there is no other latency than the expiry time itself.
The `NewArg` argument is of `intptr_t` type, allowing integer operation on integer and pointers. The choice of using a `intptr_t` type was selected because on a some processors (or data memory models), the data size of an integer is different from the data size of a pointer.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMsem()` (Section 6.9.16)

6.9.3 TIMdata

Synopsis

```
#include "Abassi.h"

void TIMdata(TIM_t *Timer, intptr_t *Addr, intptr_t Data, int ToAdd,
             int Expiry, int Period);
```

Description

`TIMdata()` is the timer service component to use to activate a timer service that writes data to a memory location. The timer descriptor to program is specified with the argument `Timer`, and the location to write to is indicated with the argument `Addr`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks. The value written at the address `Addr` is specified with the argument `Data`, which is post-write updated with the value specified in the argument `ToAdd`.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Addr`	Address where the write operation is performed
`Data`	Value to write into `Addr`. If the operation is periodic, with the argument `Period` set to a positive value, the value specified by the argument `ToAdd` is added to `Data` after every write.
`ToAdd`	Post-write update to apply
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks.
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

The data type written is of `intptr_t` type. As such, the location where the write operation occurs (`Addr`) must also be declared as an `intptr_t`. This is important, as on some processors (or depending on the memory model) the size of an `int` is different from the size of a pointer. A data type of `intptr_t` was retained as it allows writing an `int` or a pointer to the desired location. In the case a pointer is specified with the argument `Data`, the value to specify with the argument `ToAdd` is the number of address bytes to add to the pointer. For example, if the pointer is a 4 byte integer pointer, to increment the pointer by 1 integer, the value to specify with `ToAdd` is 4, not 1.
The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries. The argument (current `Data` value), the increment (`ToAdd`) and the period can be modified on the fly with the components `TIMarg()`, `TIMtoAdd()` and `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMdata()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMarg()` (Section 6.9.2)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMsem()` (Section 6.9.16)
`TIMtoAdd()` (Section 6.9.17)

6.9.4 TIMevt

Synopsis

```
#include "Abassi.h"

void TIMevt(TIM_t *Timer, TSK_t *Task, int Flags, int Expiry,
            int Period);
```

Description

`TIMevt()` is the timer service component to use to activate a timer service that sets the event flags of a task. The timer descriptor to program is specified with the argument `Timer`, and the task to set the event flags of is indicated with the argument `Task`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks. The event flags are specified with the argument `Flags`.

Availability

Only available when the build options `OS_TIMER_SRV` and `OS_EVENTS` are non-zero

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Task`	Descriptor of the task to set the event flags of
`Flags`	Event flags to set in the task `Task`
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks.
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries. The argument (current `Data` value), the increment (`ToAdd`) and the period can be modified on the fly with the components `TIMarg()`, `TIMtoAdd()` and `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMevt()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

See also

`OS_EVENTS` (Section 4.1.5)
`OS_TIMER_SRV` (Section 4.1.59)
`EVTset()` (Section 6.6.7)
`TIMarg()` (Section 6.9.3)
`TIMdata()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMsem()` (Section 6.9.16)
`TIMtoAdd()` (Section 6.9.17)

6.9.5 TIMfct

Synopsis

```
#include "Abassi.h"

void TIMfct(TIM_t *Timer, void (*Fct)(intptr), intptr_t Data,
            int ToAdd, int Expiry, int Period);
```

Description

`TIMfct()` is the timer service component to use to activate a timer service that calls a void function with a single argument. The timer descriptor to program is specified with the argument `Timer`, and the function call indicated with the argument `Fct`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks. The argument passed to the function `Fct` is specified with the argument `Data`, which is post-write updated with the value specified in the argument `ToAdd`.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Fct`	Pointer to the function to call when the call operation is performed
`Data`	Argument to the function `Fct`. If the operation is periodic, with the argument `Period` set to a positive value, the value specified by the argument `ToAdd` is added to `Data` after every function call.
`ToAdd`	Post-write update to apply
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

The data type of the function argument is of `intptr_t` type. As such, the function prototype of `Fct()` must declare a single argument of type `intptr_t`. This is important, as on some processors (or depending on the memory model) the size of an `int` is different from the size of a pointer. A data type of `intptr_t` was retained as it allows passing an `int` or a pointer to the function. In the case where a pointer is specified with the argument `Data`, the value to specify with the argument `ToAdd` is the number of address bytes to add to the pointer. For example, if the pointer is a 4 byte integer pointer, to increment the pointer by 1 integer, the value to specify with `ToAdd` is 4, not 1.
The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries.
The argument (current `Data` value), the increment (`ToAdd`) and the period can be modified on the fly with the components `TIMarg()`, `TIMtoAdd()` and `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMfct()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

If Abassi is used in a C++ environment, the function attached to the task (the argument `Fct`) must be declared with "C" linkage (see section 3.5), not as a regular C++ function.

IMPORTANT

The function called cannot use any components that enter the kernel. The reason is the function is called from within the kernel and because the kernel is non-reentrant using a kernel-based component would re-enter the kernel. If a kernel service is called in the function then the processor will enter and remain in the function `OStrap()` if either build option `OS_OUT_OF_MEM` or `OS_STACK_CHECK` is non-zero. If both build options are zero, then the kernel service called in the function returns a non-zero value reporting a non-execution/failure. Refer to `OStrap()` (Section 6.14.6) for a list of the errors that are trapped.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMarg()` (Section 6.9.2)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMsem()` (Section 6.9.16)
`TIMtoAdd()` (Section 6.9.17)

6.9.6 TIMfreeze

Synopsis

```
#include "Abassi.h"

void TIMfreeze(TIM_t *Timer);
```

Description

`TIMfreeze()` is a timer service component that stops the countdown of an active.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero. Was added in Abassi version 1.282.273 and mAbassi version 1.118.120.

Arguments

`Timer`	Descriptor of the timer service to freeze the countdown

Returns

`void`

Component type

Macro (unsafe)

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the use of the `TIMfreeze()` component on it does nothing. The component `TIMresume()` (Section 6.9.15) reactivate the countdown. If a timer has been paused with the component `TIMpause()` and the component `TIMfreeze()` is then applied on that timer, the pause state of the time gets replaced by the frozen state. If the component `TIMfreeze()` is applied multiple times on a timer already frozen, nothing happens.

See also

`OS_TIMER_SRV`(Section 4.1.59)
`TIMpause` (Section 6.9.12)
`TIMresume` (Section 6.9.15)

6.9.7 TIMkill

Synopsis

```
#include "Abassi.h"

void TIMkill(TIM_t *Timer);
```

Description

`TIMkill()` is a timer service component that deactivates an active timer service. The descriptor of the timer service to deactivate is specified with the argument `Timer`.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero

Arguments

`Timer`	Descriptor of the timer service to deactivate

Returns

void

Component type

Atomic macro (safe)

Options

N/A

Notes

If the argument `Timer` is the descriptor of a timer service which is not active, then applying the component `TIMkill()` has no effect.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMsem()` (Section 6.9.16)

6.9.8 TIMleft

Synopsis

```
#include "Abassi.h"

int TIMleft(TIM_t *Timer);
```

Description

`TIMleft()` reports the time left (in RTOS timer tick count) until the timer Timer expires (single shot) or until the next expiry (periodic).

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero. Was added in Abassi version 1.282.273 and mAbassi version 1.118.120.

Arguments

`Timer`	Descriptor of the timer service to deactivate

Returns

`>= 0` : number of RTOS timer ticks left before the expiry
`<  0` : The timer has expired or is not active

Component type

Macro (unsafe)

Options

N/A

Notes

See also

`OS_TIMER_SRV` (Section 4.1.59)

6.9.9 TIMmbx

Synopsis

```
#include "Abassi.h"

void TIMmbx(TIM_t *Timer, MBX_t *Mbox, intptr_t Data, int ToAdd,
            int Expiry, int Period;
```

Description

`TIMmbx()` is the timer service component to use to activate a timer service that writes a value into a mailbox. The timer descriptor to program is specified with the argument `Timer`, and the mailbox to write to is indicated with the argument `Mbox`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks. The value to write in the mailbox `Mbox` is specified with the argument `Data`, which is post-write updated with the value specified in the argument `ToAdd`.

Availability

Only available when the build option `OS_TIMER_SRV` and `OS_MAILBOX` are non-zero.

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Mbox`	Descriptor of the mailbox to write to
`Data`	Value to write to the mailbox `Mbox`. If the operation is periodic, with the argument `Period` set to a positive value, the value specified by the argument `ToAdd` is added to `Data` after every function call.
`ToAdd`	Post-write update to apply
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks.
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

As for any mailbox, the data type of the value to write is of `intptr_t` type. This is important, as on some processors (or depending on the memory model) the size of an `int` is different from the size of a pointer. A data type of `intptr_t` was retained as it allows the write of an int or a pointer. In the case a pointer is specified with the argument `Data`, the value to specify with the argument `ToAdd` is the number of address bytes to add to the pointer. For example, if the pointer is a 4 byte integer pointer, to increment the pointer by 1 integer, the value to specify with `ToAdd` is 4, not 1.
The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries.
The argument (current `Data` value), the increment (`ToAdd`) and the period can be modified on the fly with the components `TIMarg()`, `TIMtoAdd()` and `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMmbx()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_TIMER_SRV` (Section 4.1.59)
`TIMarg()` (Section 6.9.2)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMsem()` (Section 6.9.16)
`TIMtoAdd()` (Section 6.9.17)

6.9.10 TIMmtx

Synopsis

```
#include “Abassi.h”

void TIMmtx(TIM_t *Timer, MTX_t *Mutex, int Expiry, int Period);
```

Description

`TIMmtx()` is the timer service component to use to activate a timer service that unlocks a mutex. The timer descriptor to program is specified with the argument `Timer`, and the mutex to unlock is indicated with the argument `Mutex`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero.

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Mutex`	Descriptor of the mutex to unlock
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks.
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries. The period can be modified on the fly with the component `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMmtx()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_TIMER_SRV` (Section 4.1.59)
`TIMarg()` (Section 6.9.2)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMsem()` (Section 6.9.16)
`TIMtoAdd()` (Section 6.9.17)

6.9.11 TIMopen

Synopsis

```
#include "Abassi.h"

TIM_t *TIMopen(const char *Name);
```

Description

`TIMopen()` is the component to use to create a timer service, and is also the component to use to obtain the descriptor of an already existing timer service (when `OS_NAMES` is non-zero).

Availability

`TIMopen()` is only available when the build option `OS_RUNTIME` and `OS_TIMER_SRV` are non-zero.

Arguments

`Name`	Name of the timer service to create or to obtain the descriptor of

Returns

Descriptor of the timer service

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build where `OS_NAMES` is zero, all semaphores are unnamed, and every use of `TIMopen()` creates a new timer service.
If the build option `OS_NAMES` is non-zero, then `TIMopen()` will either return the descriptor of an existing timer service (previously created with `TIMopen()`), or when no timer services with the specified name exists, it will create a new timer service. This approach makes the creation and opening of timer services run-time safe. If that feature was not part of the `TIMopen()` component, it would be imperative to either create the timer service immediately at start-up, or to guarantee the first task (using the timer service) to reach the running state is the one creating the timer service. With the run-time safe feature, it does not matter which task is the first to open/create the timer service.
If the build option `OS_STATIC_TIM_SRV` is non-zero, the timer service descriptor uses memory that was allocated/reserved at compile/link time instead of memory dynamically allocated at run-time.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all unnamed timer services. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.
Timer services created with the `TIM_STATIC()` component are not part of the search performed by `TIMopen()`.

See also

`OS_ALLOC_SIZE` (Section 4.1.1)
`OS_STATIC_TIM_SRV` (Section 4.1.52)
`OS_TIMER_SRV` (Section 4.1.59)

6.9.12 TIMpause

Synopsis

```
#include "Abassi.h"

void TIMpause(TIM_t *Timer);
```

Description

`TIMpause()` is a timer service component that allow to temporary pause the operation performed upon expiry of a timer. It does not pause the timer countdown, it only disables the "callback" done upon expiry. To pause the timer countdown, refer to the `TIMfreeze()` component (Section 6.9.6).

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero. Was added in Abassi version 1.282.273 and mAbassi version 1.118.120.

Arguments

`Timer`	Descriptor of the timer service to pause

Returns

void

Component type

Macro (Safe)

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the use of the `TIMpause()` component on it does nothing. The component `TIMresume()` (Section 6.9.15) removes the pause. If a timer has been frozen with the component `TIMfreeze()` and the component `TIMpause()` is applied on that timer, the timer remains in the frozen condition. If the component `TIMpause()` is applied multiple times on a timer already paused, nothing happens.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMfreeze` (Section 6.9.6)
`TIMresume` (Section 6.9.15)

6.9.13 TIMperiod

Synopsis

```
#include "Abassi.h"

void TIMperiod(TIM_t *Timer, int Perod);
```

Description

`TIMperiod()` is a timer service component that modifies the period of an active timer services. It allows the application to modify the periodicity once the timer has been activated.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero

Arguments

`Timer`	Descriptor of the timer service to modify the period of
`Period`	New period of the timer. A negative or zero value makes the time operate once for the last time.

Returns

void

Component type

Data access

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the period set when using `TIMperiod()` will be overloaded when the timer is activated with `TIMdata()`, `TIMevt()`, or `TIMfct()`, `TIMmbx()`, `TIMmtx()`, or `TIMsem()`.
If the argument `Period` is negative or equal to 0. it will deactivate the timer once the expiry time is reached.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)

`TIMmtx()` (Section 6.9.10)
`TIMsem()` (Section 6.9.16)

6.9.14 TIMrestart

Synopsis

```
#include "Abassi.h"

void TIMrestart(TIM_t *Timer);
```

Description

`TIMrestart()` is a timer service component that allow the restart of a timer.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero. Was added in Abassi version 1.282.273 and mAbassi version 1.118.120.

Arguments

`Timer`	Descriptor of the timer service to restart

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the use of the `TIMrestart()` component on it does nothing. When an active timer is restarted with the component `TIMrestart()`, it is almost the same as when the timer was activated with activated with `TIMdata()`, `TIMevt()`, or `TIMfct()`, `TIMmbx()`, `TIMmtx()`, or `TIMsem()`. If a timer has been paused with the component `TIMpause()` or frozen, with the component `TIMfreeze()`, `TIMrestart()` used on such a timer will activate the timer, effectively canceling the pause or frozen state.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMfreeze` (Section 6.9.6)
`TIMpause` (Section 6.9.12)
`TIMperiod` (Section 6.9.13)

6.9.15 TIMresume

Synopsis

```
#include "Abassi.h"

int TIMresume(TIM_t *Timer);
```

Description

`TIMresuem()` is the timer service component to use to "re-activate" a timer that has been paused with `TIMpause()` (Section 6.9.12), or frozen with `TIMfreeze()` (Section 6.9.6).

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero. Was added in Abassi version 1.282.273 and mAbassi version 1.118.120.

Arguments

`Timer`	Descriptor of the timer service to reactivate.

Returns

`void`

Component type

Macro (unsafe)

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the use of the `TIMresume()` component on it does nothing. Same if the timer is active and neither `TIMpause()` nor `TIMfreeze()` has been applied to it, it does nothing.

See also

`OS_TIMER_SRV`(Section 4.1.59)
`TIMpause` (Section 6.9.12)
`TIMresume` (Section 6.9.15)

6.9.16 TIMsem

Synopsis

```
#include "Abassi.h"

void TIMsem(TIM_t *Timer, SEM_t *Sema, int Expiry, int Period);
```

Description

`TIMsem()` is the timer service component to use to activate a timer service that post a semaphore. The timer descriptor to program is specified with the argument `Timer`, and the semaphore to post is indicated with the argument `Sem`. The programmed operation can be a single time delayed operation when the argument `Period` is less or equal to 0, or if positive, it can be periodic, once every `Period` timer ticks. The first timer expiry always occurs after `Expiry` timer ticks.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero.

Arguments

`Timer`	Descriptor of the timer service to use to perform the timed operation
`Sema`	Descriptor of the semaphore to post
`Expiry`	Number of timer ticks the timer is requested to expire after the first time. If the timer is periodic, when the argument `Period` is positive, then further expiries will occur every `Period` timer ticks.
`Period`	If set to a zero or negative value, the timer performs a one shot operation. If set to a positive value, periodicity in number of timer ticks.

Returns

void

Component type

Macro (unsafe)

Options

N/A

Notes

The argument `Expiry` must be positive. If a zero or negative value is specified, the application will quite likely lock-up as it will continuously have to handle timer expiries. The period can be modified on the fly with the component `TIMperiod()`.
The timer can be deactivated at any time with the component `TIMkill()`. If `TIMsem()` is applied on an timer service that is already active, the timer service will first be deactivated, reprogrammed and then reactivated.

See also

`OS_MAILBOX` (Section 4.1.18)
`OS_TIMER_SRV` (Section 4.1.59)
`TIMarg()` (Section 6.9.2)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)
`TIMfct()` (Section 6.9.5)
`TIMkill()` (Section 6.9.7)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMperiod()` (Section 6.9.13)
`TIMtoAdd()` (Section 6.9.17)

6.9.17 TIMtoAdd

Synopsis

```
#include "Abassi.h"

void TIMtoAdd(TIM_t *Timer, int ToAdd);
```

Description

`TIMtoAdd()` is a timer service component that modifies the value to add to the argument of some but not all of the timer services operations. It allows the application to replace the argument update value once the timer has been activated. The timer services components that use arguments and can have `TIMarg()` applied to are: `TIMdata()`, `TIMevt()`, `TIMfct()`, and `TIMmbx()`.

Availability

Only available when the build option `OS_TIMER_SRV` is non-zero.

Arguments

`Timer`	Descriptor of the timer service to replace the argument to the operation
`ToAdd`	New value of operation argument modifier

Returns

void

Component type

Data access

Options

N/A

Notes

If the timer service descriptor specified with the argument `Timer` is not active, the argument value set when using `TIMtoAdd()` will be overloaded when the timer is activated with `TIMdata()`, `TIMevt()`, `TIMfct()` or `TIMmbx()`.
There is no side effect if the component `TIMtoAdd()` is applied on an inactive timer or a timer that was activated to perform an operation which does not requires an argument, such as `TIMmtx()` and `TIMsem()`. The new value of the timer operation argument is the argument that will be used the next time the timer expires; there is no other latency than the expiry time itself.

See also

`OS_TIMER_SRV` (Section 4.1.59)
`TIMdata()` (Section 6.9.3)
`TIMevt()` (Section 6.9.4)

`TIMfct()` (Section 6.9.5)
`TIMmbx()` (Section 6.9.9)
`TIMmtx()` (Section 6.9.10)
`TIMopen()` (Section 6.9.11)
`TIMsem()` (Section 6.9.16)

6.10 Memory Block Management Services

The optional memory block management services gives access to one or multiple data memory pools, each holding one or more blocks of data, always with the same size within a pool; the number of blocks and their size is selected upon creation of the memory pool. All the data blocks supplied by the memory block management services are aligned to the largest data alignment required by the processor. When used with the mailbox services, the memory block management services simplify the management of queues with buffers.

Table 6-21 Memory Block Management Service Component list

Section	Name	Description
6.10.2	`MBLKalloc`	Retrieve a memory block from a pool
6.10.3	`MBLKfree`	Return a memory block to its pool
6.10.4	`MBLKnotFCFS`	Set a memory block pool to operate in the *Priority* mode.
6.10.5	`MBLKopen`	Create a memory block pool / obtain the descriptor of a memory block pool
6.10.6	`MBLKopenFCFS`	Create a memory block pool to operate in *First Come First Served* mode / obtain the descriptor of a memory block pool
6.10.7	`MBLKsetFCFS`	Set a memory block pool to operate in the *First Come First Served* mode.

The usage of the memory block management service is quite straightforward, alike `malloc()` and `free()`. First, the pool of memory must be created with the component `MBLKopen()` or `MBLKopenFCFS()`. Then a buffer is retrieved from the memory pool with the `MBLKalloc()` component and returned to the same memory pool with the `MBLKfree()` component. A task can block when retrieving a buffer from an empty memory pool; it will unblock when a buffer is returned to the empty pool or a timeout occurs.

The memory block management service is the only type of dynamic memory allocation service supplied as an integral part of Abassi. Custom byte size memory pool management isn't provided for the simple reason that the `malloc()` family of functions are a standard requirement in ANSI C-99.

6.10.1 Memory Requirement Rules

The memory required to hold all the memory blocks in a single pool is not simply the size of the buffers times the number of blocks. There are a few rules due to the way the memory block management is implemented, and also some constrains due to the fact that each buffer in the memory pool is aligned according to the largest data alignment requirement by the processor. The explanation given here is to inform the designer of the exact amount of memory a memory pool requires. One must remember the information given here is the amount of memory used by the memory pool; when a memory block management pool is created, the size of the block can be specified as small as a single byte (although the real buffer size is quite likely bigger than a single byte).

The amount of memory used to hold the buffers in a memory block management pool depends on the real block size, as used internally by the service. Each memory block must be sized to at least the size of a pointer. This requirement exists as the memory block, when held inside the pool, is part of a linked list of "free buffers"; therefore, the buffer must be able to hold a pointer. Using a pointer to hold the linked list pointer eliminates the extra memory that would be required to implement a linked list. The second requirement is due to the fact that each buffer from a memory block is guaranteed to be aligned to the largest data alignment requirement of the processor. Therefore, each memory block must be sized to an exact multiple of the size of the data type that requires the largest alignment by the processor[2].

[2] Internally, this data type is defined as `OSalign_t`. In case of uncertainty on the maximum alignment requirement by the processor, verify the definition of `OSalign_t` in `Abassi.h` for the target processor.

The memory requirement to hold all the buffers in a memory block pool is as follows:

Aligned Block Size:	`sizeof(OSalign_t)`
	`* ceil((Block Size) / sizeof(OSalign_t))`
Internal Block Size:	`max((Aligned Block Size), sizeof(void *))`
Total Memory:	`(Number of Blocks) * (Internal Block Size)`

For all processors supported by Abassi, `sizeof(OSalign_t)` is 8 bytes or less. If data memory is not an issue in the target application, always use memory blocks sized with exact multiple of 8 bytes.

6.10.2 MBLKalloc

Synopsis

```
#include "Abassi.h"

void *MBLKalloc(MBLK_t *MemBlock, int Timeout);
```

Description

`MBLKalloc()` is the component to use to retrieve a memory block from a memory pool. The memory block management descriptor is specified by the argument `MemBlock`. When the memory pool is empty, it is possible to request the task to block until a buffer is returned to the memory pool, or to block with timeout, or to not block at all.

Availability

`MBLKalloc()` is only available when the build option `OS_MEM_BLOCK` is non-zero. Depending on the setting of the build option `OS_TIMEOUT`, the meaning of the argument `Timeout` slightly changes. See **Options** below.

Arguments

`MemBlock`	Descriptor of the memory block management pool to return the buffer to	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Number of timer ticks before expiry

Returns

`void *`	Base address of the buffer retrieved from the memory pool
`NULL`	The memory pool is empty. This will occur if the argument `Timeout` is non-negative. Either `Timeout` was zero and there was no buffer, or `Timeout` was positive and no new buffer was returned to the memory pool within `Timeout` number of timer ticks (or when the component `TSKtimeoutKill()` is applied to the task waiting on the memory pool). If an application ever has multiple extractors for a memory pool, blocked readers will unblock in a priority ordering, or request order if *First Come First Served*, depending on the mode of operation of the memory block pool.

Component type

Function

Options

If the build option `OS_TIMEOUT` is zero, then when the argument `Timeout` is set to a positive value, `MBLKalloc()` behaves the same as if the `Timeout` argument had been set to zero.
If the build option `OS_TIMEOUT` is a negative value, then when the argument `Timeout` is set to a positive value, `MBLKalloc()` behaves the same as if the `Timeout` argument had been set to a negative value.

Notes

Unless `Timeout` is negative, always verify the return value: a `NULL` value means the memory pool is empty; non-`NULL` means a buffer was retrieved and the pointer is the base address of the buffer, even if the timeout was set to zero.

When timeout is negative, the component `TSKtimeoutKill()` cannot unblock the task waiting to obtain a memory buffer, as an infinite timeout request does not involve the timer service.

`MBLKalloc()` can be used in an ISR as this component has special hooks added to it for this purpose. When used in an ISR, the argument `Timeout` is ignored and is always considered being equal to zero. The return value for this component is valid in an ISR.

Make sure to always return a block of memory to the memory block management pool it was obtained from. There are no checks performed to enforce this requirement.

See also

`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKfree()` (Section 6.10.3)
`MBLKopen()` (Section 6.10.5)
`MBLKopenFCFS()` (Section 6.10.6)

6.10.3 MBLKfree

Synopsis

```
#include “Abassi.h”

void MBLKfree(MBLK_t *MemBlock, void *Buffer);
```

Description

`MBLKfree()` is the component to use to return a memory block to its original pool. The memory block management pool descriptor is specified by the argument `MemBlock`, and the pointer to the base of the buffer to return is specified with the argument `Buffer`.

Availability

`MBLKfree()` is only available when the build option `OS_MEM_BLOCK` is non-zero.

Arguments

`MemBlock`	Descriptor of the memory block management pool to return the buffer to
`Buffer`	Pointer to the base address of the buffer to return to the pool

Returns

`void`

Component type

Macro (Safe)

Options

Notes

Make sure to always return a block of memory to the memory block management pool it was obtained from. There are no checks performed to enforce this requirement. Also, the pointer indicated by `Buffer` must be the same as the one that was obtained through `MBLKalloc()`.

If the pointer to the buffer to return is `NULL`, the use of this component has no effect; the kernel traps the occurrence of a `NULL` pointer and skips further processing.

See also

`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKalloc()` (Section 6.10.2)
`MBLKopen()` (Section 6.10.5)
`MBLKopenFCFS()` (Section 6.10.6)

6.10.4 MBLKnotFCFS

Synopsis

```
#include "Abassi.h"

void MBLKnotFCFS(MBLK_t *MemBlock);
```

Description

`MBLKnotFCFS()` is the component to use to configure a memory block management pool to operate in the *Priority* mode. The unblocking order of such a memory block management pool is always the highest priority task that is blocked.

Availability

`MBLKnotFCFS()` is only available when the build options `OS_MEM_BLOCK` and `OS_FCFS` are both non-zero.

Arguments

`MemBlock`	Descriptor of the memory block management pool to set into a *Priority* mode

Returns

`void`

Component type

Definition

Options

Notes

If the memory block management pool was already operating in the *Priority* mode, using this component has no effect on such a memory pool.
If the memory block management pool was operating in the *First Come First Served* mode, using this component will not re-order tasks that are currently blocked on the memory pool. Newly blocked tasks will be inserted in a *Priority* ordering amongst the already *First Come First Served* ordered blocked tasks. This means there may be a transient phase before the memory block management truly operates in a *Priority* mode.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKnotFCFS ()` (Section 6.7.6)
`MBLKopen()` (Section 6.10.5)
`MBLKopenFCFS()` (Section 6.10.6)
`MBLKsetFCFS()` (Section 6.10.7)

6.10.5 MBLKopen

Synopsis

```
#include “Abassi.h”

MBX_t *MBLKopen(const char *Name, int NmbBlock, int BlkSize);
```

Description

`MBLKopen()` is the component to use to create a memory block pool, and is also the component to use to obtain the descriptor of an already existing memory block pool. When a memory block pool is created with `MBLKopen()`, it operates in the *Priority* mode (for the extractor).

Availability

`MBLKopen()` is only available if the build option `OS_MEM_BLOCK` and `OS_RUNTIME` are both non-zero.

Arguments

`Name`	Name of the memory block pool to create or to obtain the descriptor of
`NmbBlock`	Maximum number of memory blocks the pool can hold
`BlkSize`	Size in bytes of each memory block

Returns

Descriptor of the memory block pool

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build when `OS_NAMES` is zero, all memory block managements are unnamed and every use of `MBLKopen()` creates a new memory block pool.
If the build option `OS_NAMES` is non-zero, then `MBLKopen()` will either return the descriptor of an existing memory block pool (previously created with `MBLKopen()`or `MBLKopenFCFS()`), or, when no memory block pool with the specified name exists, it will create a new memory pool. This approach makes the creation and opening of memory block pools run-time safe. If that feature were not part of the `MBLKopen()` component, it would be imperative to either create the memory pool immediately at start-up, or to guarantee the first task (using the memory pool) to reach the running state is the one creating the memory pool. With the run-time safe feature, it does not matter which task is the first to use the memory pool.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed memory pools. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.

Be aware, if the build option `OS_FCFS` is non-zero, when the memory pool already exists, there is no guarantee the memory pool is operating in a *Priority* mode, as it may have been created with `MBLKopenFCFS()`or it may have been set to operate in *First Come First Served* mode with `MBLKsetFCFS()`.

Also, if a memory pool already exists, the requested number of blocks and the block size may not be the ones of the existing memory pool.

At any time, a memory pool operating in the *Priority* mode can be modified to operate in the *First Come First Served* mode by using the `MBLKsetFCFS()` component when the build option `OS_FCFS` is non-zero.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKnotFCFS ()` (Section 6.10.4)
`MBLKopenFCFS()` (Section 6.10.6)
`MBLKsetFCFS()` (Section 6.10.7)

6.10.6 MBLKopenFCFS

Synopsis

```
#include "Abassi.h"

MBX_t *MBLKopenFCFS(const char *Name, int NmbBlock, int BlkSize);
```

Description

`MBLKopenFCFS()` is the component to use to create a memory block pool, and is also the component to use to obtain the descriptor of an already existing memory block pool. When a memory block pool is created with `MBLKopenFCFS()`, it operates in the *First Come First Served* mode (for the extractor).

Availability

`MBLKopenFCFS()` is only available if the build options `OS_MEM_BLOCK`, `OS_RUNTIME` and `OS_FCFS` are all non-zero.

Arguments

`Name`	Name of the memory block pool to create or to obtain the descriptor of
`NmbBlock`	Maximum number of memory blocks the pool can hold
`BlkSize`	Size in bytes of each memory block

Returns

Descriptor of the memory block pool

Component type

Function
- Cannot be used in an interrupt -

Options

When the build option `OS_NAMES` is zero, the argument `Name` is ignored but must still be supplied. In a build when `OS_NAMES` is zero, all memory block managements are unnamed and every use of `MBLKopenFCFS()` creates a new memory block pool.
If the build option `OS_NAMES` is non-zero, then `MBLKopenFCFS()` will either return the descriptor of an existing memory pool (previously created with `MBLKopen()`or `MBLKopenFCFS()`), or when no memory block pool with the specified name exists, it will create a new memory pool. This approach makes the creation and opening of memory block pools run-time safe. If that feature were not part of the `MBLKopenFCFS()` component, it would be imperative to either create the memory pool immediately at start-up, or to guarantee the first task (using the memory pool) to reach the running state is the one creating the memory pool. With the run-time safe feature, it does not matter which task is the first to use the memory pool.

Notes

One should avoid setting the argument `Name` to `NULL` when the build option `OS_NAMES` is zero. The reasoning behind this is: if logging ever needs to be turned on, it becomes impossible to differentiate amongst all the unnamed memory pools. This is also the reason why the function prototype for this component was kept the same, irrespective of the setting of the build option `OS_NAMES`.
Be aware, if the build option `OS_FCFS` is non-zero, when the memory pool already exists, there is no guarantee the memory pool is operating in a *First Come First Served* mode, as it may have been created with `MBLKopen()`or it may have been set to operate in *Priority* mode with `MBLKset()`.
Also, if a memory pool already exists, the requested number of blocks and the block size may not be the ones of the existing memory pool.
At any time, a memory pool operating in the *First Come First Served* mode can be modified to operate in the *Priority* mode by using the `MBLKnotFCFS()` component when the build option `OS_FCFS` is non-zero.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKnotFCFS ()` (Section 6.10.4)
`MBLKopen()` (Section 6.10.5)
`MBLKsetFCFS()` (Section 6.10.7)

6.10.7 MBLKsetFCFS

Synopsis

```
#include "Abassi.h"

void MBLKsetFCFS(MBLK_t *MemBlock);
```

Description

`MBLKsetFCFS()` is the component to use to configure a memory block management pool to operate in the *First Come First Served* mode. The unblocking order of such a memory pool is always the oldest task that was blocked is unblocked first.

Availability

`MBLKsetFCFS()` is only available when the build options `OS_MEM_BLOCK` and `OS_FCFS` are both non-zero.

Arguments

`MemBlock`	Descriptor of the memory block management pool to set into a *First Come First Served* mode.

Returns

`void`

Component type

Definition

Options

Notes

If the memory block management pool was already operating in the *First Come First Served* mode, using this component has no effect on such a memory pool.
If the memory block management pool was operating in the *Priority* mode, using this component will not re-order tasks that are currently blocked on the memory pool. Newly blocked tasks will be inserted in a *First Come First Served* ordering amongst the already *Priority* ordered blocked tasks. This means there may be a transient phase before the memory block management truly operates in a *First Come First Served* mode.

See also

`OS_FCFS` (Section 4.1.6)
`OS_MEM_BLOCK` (Section 4.1.18)
`OS_STATIC_BUF_MBLK` (Section 4.1.44)
`OS_STATIC_MBLK` (Section 4.1.46)
`MBLKnotFCFS ()` (Section 6.10.4)
`MBLKopen()` (Section 6.10.5)
`MBLKopenFCFS()` (Section 6.10.6)

6.10.8 Examples

Refer to the mailbox example in section 6.7.13

6.11 GROUPING

The optional grouping of services is used to encapsulate multiple services and make them operate as a single blocking service. An example where service grouping is useful is when a task needs to concurrently process the contents of 2 or more mailboxes. In the absence of grouping, an extra blocking service, e.g. semaphore or event flags, must be used and managed by both the writer and the reader of the mailboxes. With grouping, when said mailboxes are attached to a group, nothing special needs to be done on the writer side and all the reader needs to do is to wait on the group.

Table 6-22 Memory Block Management Service Component list

Section	Name	Description
6.11.4	`GRPaddMBX`	Add a mailbox and its callback function to a group
6.11.5	`GRPaddSEM`	Add a counting semaphore and its callback function to a group
6.11.6	`GRPaddSEMbin`	Add a binary semaphore and its callback function to a group
6.11.7	`GRPdscMBX`	Obtain the descriptor of the trigger a mailbox is attached to
6.11.8	`GRPdscSEM`	Obtain the descriptor of the trigger a semaphore is attached to
6.11.9	`GRPrmAll`	Remove all triggers from a group
6.11.10	`GRPrmMBX`	Remove a mailbox from a group
6.11.11	`GRPrmSEM`	Remove a semaphore from a group
6.11.12	`GRPwait`	Wait for

The usage of the grouping service is fairly intuitive. A group must first be created, which is done by attaching services to that group. The services are attached using the components `GRPaddMBX()`, `GRPaddSEM()` and `GRPaddSEMbin()`. When a service is attached to a group, a callback function is specified for each attached service, as that callback function is used when its associated service is available. A callback function eliminates the need for the application to determine which of the triggers has been validated. All there is to do then in a task is to use the component `GRPwait()` to wait / block on the group of services; when a trigger is validated, the callback function is automatically operated. At any time, a selected service or all services in a group can be deleted using the components `GRPrmMBX()`, `GRPrmSEM()` and `GRPrmAll()`.

6.11.1 Nomenclature

The grouping facility is described using three different names (*group*, *trigger* and *service*) and two qualifiers (*attached* and *part of*). A group is internally a linked list of `GRT_t` type of descriptors. Each `GRP_t` type of descriptors in a group's linked list are called a trigger descriptor. Each trigger descriptor has a blocking service attached to it: either a mailbox to be read, or a counting semaphore to acquire, or a binary semaphore to acquire. So when a service is "attached" to a group, in truth it is attached to a trigger descriptor, which in turn is added in the group's linked list.

6.11.2 Restrictions

There are a few restrictions when groups of triggers are used. The restrictions are the followings:

- A service can only be used as a trigger in a single group only. When a request to attach a service to a second group is performed, the attachment operation to the second group is aborted and an error reported.
- Only one task can wait on a group. If a second task attempts to wait on a group that is already being waited on by another task, the waiting operation of the second task is aborted and an error reported.

- If a group does not have any services attached to it, a request to wait on that group will be aborted and an error reported.
- None of the group components can be used inside an interrupt. If any of the components is used in an interrupt, the application will most likely lock-up.

NOTE: the callback function is executed within the context of the task that has used the `GRPwait()` component.

6.11.3 Coexistence

When a service is one of the triggers attached to a group, it does not change the usability of that service outside of the group; this means the group does not require nor have exclusive blocking access to that service. In other words, although a service may be part of a group, e.g. a semaphore, that semaphore can still be acquired in an independent manner from using the group. For example, a task can wait on a group a semaphore is part of, and at the same time another task can try to acquire that same semaphore (as is, not as part of the group). The mixed group / standalone usage changes a bit the order tasks get unblocked (see below).

When coexisting with usage outside of a group, services in a group do not exactly block tasks in the same order as if they were not part of a group. When a service is configured to operate in the *First Come First Served* (FCFS) mode, then if a task is blocked on a group (using `GRPwait`) in which that service is attached, that task will always get unblocked before any other tasks that are blocked on that service (using `MBXget` or `SEMwaitBin`). This is alike a group jumping the line when waiting for a service configured in the FCFS mode. When the service is configured in *Priority* mode and it is part of a group, then the unblocking of task is still done according to the priorities of the tasks blocked on it. But a task blocked on a group that has this service attached to it will always get unblocked first amongst all tasks at the same priorities (blocked through `MBXget` or `SEMwaitBin`). This is almost the same as when the service is configured in FCFS mode, except the jumping of the line is only done amongst tasks with the same priority level as the task that is blocked on the group.

6.11.4 GRPaddMBX

Synopsis

```
#include "Abassi.h"

GRP_t *GRPaddMBX(GRP_t *Group, MBX_t *Mbox, void (* CB)(intptr_t));
```

Description

`GRPaddMBX()` is the component that attaches a mailbox service to a group of triggers. The argument `Group` is the pointer to the group descriptor to add the `Mbox` service to. When the mailbox `Mbox` contains something, using `GRPwait()`, then the function `CB` is called within `GRPwait()`, passing the element retrieved from the mailbox as its argument.
Upon the first attachment to a group, the group is always considered to not exist. So the argument `Group` **must always** be set to `NULL` when performing the first attachment. Further attachments to that new group must then use the returned value when `Group` was set to `NULL`.

Availability

`GRPaddMBX()` is only available when both build options `OS_GROUP and OS_MAILBOX` are non-zero.

Arguments

`Group`	Pointer to the group descriptor to attach the mailbox service `Mbox`. Upon the first attachment, `Group` must be set to `NULL`. Further attachment must set `Group` to the value returned when the first attachment was performed.
`Mbox`	Pointer to the descriptor of the mailbox service to attach to the group `Group`
`CB`	Pointer to the callback function used when the `Mbox` service is not empty. The argument passed when the callback function is called is the element read from the mailbox.

Returns

`GRP_t *`	Pointer to the descriptor of the newly created group (Only valid when the first service is attached with the argument `Group` set to NULL). Non-NULL in other cases simply returns the argument `Group`.
`NULL`	Indicates an error. The two possible errors are: the `Mbox` service is already attached to another group; when using pre-allocated trigger descriptors (build option `OS_GROUP > 0`), out of trigger descriptors.

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

When the first service is attached to a group, the argument `Group` **must always** be set to `NULL`. Re-using what may be considered an empty group (e.g. after having used the component `GRPrmALL()` on that group) does not guaranteed the group is empty as it may have been reused if another group was created.
For proper operation, the component `MBXget()` should not be used in the callback function as the element has already been retrieved from the mailbox and is passed as the argument to the callback function.
When a non-empty mailbox is attached to a group on which a task is blocked, the attachment of the mailbox will immediately unblock the task.

See also

`OS_GROUP` (Section 4.1.7)
`OS_MAILBOX` (Section 4.1.18)
`GRPaddSEM()` (Section 6.11.5)
`GRPaddSEMbin()` (Section 6.11.6)
`GRPrmAll()` (Section 6.11.9)
`GRPrmMBX()` (Section 6.11.10)
`GRPrmSEM()` (Section 6.11.11)
`GRPwait()` (Section 6.11.12)
`MBXget()` (Section 6.7.5)

6.11.5 GRPaddSEM

Synopsis

```
#include "Abassi.h"

GRP_t *GRPaddSEM(GRP_t *Group, SEM_t *Sema, void (* CB)(SEM_t *));
```

Description

`GRPaddSEM()` is the component that attaches a counting semaphore service to a group of triggers. The argument `Group` is the pointer to the group descriptor to add the `Sema` service to. When the semaphore `Sema` has a positive count, using `GRPwait()`, then the function `CB` is called within `GRPwait()`, passing the pointer to the service descriptor `Sema` as its argument.
Upon the first attachment to a group, the group is always considered to not exist. So the argument `Group` **must always** be set to `NULL` when performing the first attachment. Further attachments to that new group must then use the returned value when `Group` was set to `NULL`.

Availability

`Only` available when the build option `OS_GROUP` is non-zero.

Arguments

`Group`	Pointer to the group descriptor to attach the counting semaphore service `Sema`. Upon the first attachment, `Group` must be set to `NULL`. Further attachment must set `Group` to the value returned when the first attachment was performed.
`Sema`	Pointer to the descriptor of the counting semaphore service to attach to the group `Group`.
`CB`	Pointer to the callback function used when the `Sema` service has a positive count. The argument passed when the callback function is called is pointer to the descriptor of the semaphore `Sema`.

Returns

`GRP_t *`	Pointer to the descriptor of the newly created group (only valid when the first service is attached with the argument `Group` set to NULL). Non-NULL in other cases indicates success.
`NULL`	Indicates an error. The two possible errors are: the `Sema` service is already attached to another group; when using pre-allocated trigger descriptors (build option `OS_GROUP` >0), out of trigger descriptors.

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

When the first service is attached to a group, the argument `Group` **<u>must always</u>** be set to `NULL`. Re-using what may be considered an empty group (e.g. after having used the component `GRPrmALL()` on that group) does not guaranteed the group is empty as it may have been reused if another group was created.
One must not use the component `SEMwait()` on the trigger semaphore in the callback function (the trigger semaphore descriptor is the lone argument of the callback function). If `SEMwait()` is used on the trigger semaphore it becomes a double waiting on that semaphore.
When semaphore with a positive count is attached to a group on which a task is blocked, the attachment of the semaphore will immediately unblock the task.

See also

`OS_GROUP` (Section 4.1.7)
`GRPaddMBX()` (Section 6.11.4)
`GRPaddSEMbin()` (Section 6.11.6)
`GRPrmAll()` (Section 6.11.9)
`GRPrmMBX()` (Section 6.11.10)
`GRPrmSEM()` (Section 6.11.11)
`GRPWait()` (Section 6.11.12)
`SEMwait()` (Section 6.4.11)

6.11.6 GRPaddSEMbin

Synopsis

```
#include "Abassi.h"

GRP_t *GRPaddSEMbin(GRP_t *Group, SEM_t *Sema, void (* CB)(SEM_t *));
```

Description

`GRPaddSEM()` is the component that attaches a binary semaphore service to a group of triggers. The argument `Group` is the pointer to the group descriptor to add the `Sema` service to. When the semaphore `Sema` has a positive count, using `GRPwait()`, then the function `CB` is called within `GRPwait()`, passing the pointer to the service descriptor `Sema` as its argument.
Upon the first attachment to a group, the group is always considered to not exist. So the argument `Group` **must always** be set to `NULL` when performing the first attachment. Further attachments to that new group must then use the returned value when `Group` was set to `NULL`.

Availability

`Only` available when the build option `OS_GROUP` is non-zero.

Arguments

`Group`	Pointer to the group descriptor to attach the counting semaphore service `Sema`. Upon the first attachment, `Group` must be set to `NULL`. Further attachment must set `Group` to the value returned when the first attachment was performed.
`Sema`	Pointer to the descriptor of the binary semaphore service to attach to the group `Group`
`CB`	Pointer to the callback function used when the `Sema` service has a positive count. The argument passed when the callback function is called is pointer to the descriptor of the semaphore `Sema`.

Returns

`GRP_t *`	Pointer to the descriptor of the newly created group (only valid when the first service is attached with the argument `Group` set to NULL). Non-NULL in other cases indicates success.
`NULL`	Indicates an error. The two possible errors are: the `Sema` service is already attached to another group; when using pre-allocated trigger descriptors (build option `OS_GROUP` >0), out of trigger descriptors.

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

When the first service is attached to a group, the argument `Group` **<u>must always</u>** be set to `NULL`. Re-using what may be considered an empty group (e.g. after having used the component `GRPrmALL()` on that group) does not guaranteed the group is empty as it may have been reused if another group was created.

One must not use the component `SEMwaitBin()` on the trigger semaphore in the callback function (the trigger semaphore descriptor is the lone argument of the callback function). If `SEMwaitBin()` is used on the trigger semaphore it becomes a double waiting on that semaphore.

When semaphore with a positive count is attached to a group on which a task is blocked, the attachment of the semaphore will immediately unblock the task.

See also

`OS_GROUP` (Section 4.1.7)
`GRPaddMBX()` (Section 6.11.4)
`GRPaddSEM()` (Section 6.11.5)
`GRPrmAll()` (Section 6.11.9)
`GRPrmMBX()` (Section 6.11.10)
`GRPrmSEM()` (Section 6.11.11)
`GRPwait()` (Section 6.11.12)
`SEMwaitBin()` (Section 6.4.12)

6.11.7 GRPdscMBX

Synopsis

```
#include "Abassi.h"

GRP_t *GRPdscMBX(MBX_t *MBX_t);
```

Description

`GRPdscMBX()` is the component to use to obtain the pointer to the group descriptor the mailbox `MBox` service is attached to.

Availability

`GRPdscMBX()` is only available when both build options `OS_GROUP and OS_MAILBOX` are non-zero.

Arguments

`Mbox`	Pointer to the descriptor of the mailbox service to obtain the group it is attached to

Returns

`GRP_t *`	Pointer to the descriptor of the group `Mbox` is attached to.
`NULL`	The mailbox service `Mbox` is not part of any group.

Component type

Macro (safe)

Options

N/A

Notes

See also

`OS_GROUP` (Section 4.1.7)
`OS_MAILBOX` (Section 4.1.18)
`GRPdscSEM()` (Section 6.11.8)
`GRPrmAll()` (Section 6.11.9)

6.11.8 GRPdscSEM

Synopsis

```
#include "Abassi.h"

GRP_t *GRPdscSEM(SEM_t *Sema);
```

Description

`GRPdscSEM()` is the component to use to obtain the pointer to the group descriptor the semaphore `Sema` service is attached to.

Availability

`GRPdscSEM()` is only available when the build options `OS_GROUP` is non-zero.

Arguments

`Sema`	Pointer to the descriptor of the semaphore service to obtain the group it is attached to

Returns

`GRP_t *`	Pointer to the descriptor of the semaphore `Sema` is attached to.
`NULL`	The semaphore service `Sema` is not part of any group.

Component type

Macro (safe)

Options

N/A

Notes

See also

`OS_GROUP` (Section 4.1.7)
`GRPdscMBX()` (Section 6.11.7)

6.11.9 GRPrmAll

Synopsis

```
#include "Abassi.h"

int GRPrmAll(GRP_t *Trigger);
```

Description

`GRPrmAll()` is a group component that removes all triggers from the group. If a task is blocked on the group, when the group is emptied, it will force the unblocking of the task and the task is informed of this abnormal condition through the return value of `GRPwait()`.

Availability

Only available when the build option `OS_GROUP` is non-zero.

Arguments

`Trigger`	The trigger descriptor of the service to remove. The descriptor can be obtain using either `GRPdscMBX()` or `GRPdscSEM()`.

Returns

`== 0`	The group has been successfully emptied
`!= 0`	The group is already empty

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

See also

`OS_GROUP` (Section 4.1.7)
`GRPdscMBX()` (Section 6.11.7)
`GRPdscSEM()` (Section 6.11.8)
`GRPrmAll()` (Section 6.11.9)

6.11.10 GRPrmMBX

Synopsis

```
#include "Abassi.h"

int GRPrmMBX(MBX_t *Mbox);
```

Description

`GRPrmMBX()` is a group component that detaches the mailbox `Mbox` from the group it is part of. If the mailbox to detach is the only trigger in the group and a task is blocked on that group, when the mailbox is detached from the group, it will force the unblocking of the task and the task is informed of this abnormal condition through the return value of `GRPwait()`.

Availability

Only available when the build options `OS_GROUP` is non-zero and `OS_MAILBOX` are non-zero.

Arguments

`Mbox`	The descriptor of the Mailbox to remove

Returns

`== 0`	The mailbox has been successfully removed from the group it was part of
`!= 0`	The mailbox is not attached to any group

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

See also

`OS_GROUP` (Section 4.1.7)
`GRPdscMBX()` (Section 6.11.7)
`GRPrmAll()` (Section 6.11.9)
`GRPrmSEM()` (Section 6.11.8)

6.11.11 GRPrmSEM

Synopsis

```
#include "Abassi.h"

int GRPrmSEM(SEM_t *Sema);
```

Description

`GRPrmSEM()` is a group component that detaches the semaphore `Sema` from the group it is part of. If the semaphore to detach is the only trigger in the group, and a task is blocked on that group, when the semaphore is detached from the group, it will force the unblocking of the task and the task is informed of this abnormal condition through the return value of `GRPwait()`.

Availability

Only available when the build option `OS_GROUP` is non-zero.

Arguments

`Sema`	The descriptor of the semaphore to remove

Returns

`== 0`	The semaphore has been successfully removed from the group it was part of
`!= 0`	The semaphore is not attached to any group

Component type

Macro (safe)
- Cannot be used in an interrupt -

Options

N/A

Notes

See also

`OS_GROUP` (Section 4.1.7)
`GRPdscSEM()` (Section 6.11.8)
`GRPrmAll()` (Section 6.11.9)
`GRPrmMBX()` (Section 6.11.7)

6.11.12 GRPwait

Synopsis

```
#include "Abassi.h"

int GRPwait(GRP_t *Group, int TimeOut, int All);
```

Description

`GRPwait()` is the group component used by a task to wait / block on a group of triggers. The wait / block on is specified with the argument `Group`. Through the argument `Timeout`, it is possible to request to block until one or more triggers are validated or to block with timeout, or no blocking at all. The argument `All` is used to set the operation of `GRPwait()` into a bulk waiting mode: `GRPwait()` can then continuously wait for a trigger as long as the timeout, which is restarted after each valid triggers, does not expire.
When a trigger in the Group `Group` is validated, the callback function that was associated with the trigger (when the components `GRPaddMBX()`, `GRPaddSEM()` or `GRPaddSEMbin()` were applied on that trigger) will be called inside `GRPwait()`.

Availability

Only available when the build option `OS_GROUP` is non-zero.

Arguments

`Group`	Pointer to the group to wait / block.	
`Timeout`	Negative	Infinite blocking
	0	Never blocks
	Positive	Number of timer ticks before expiry
`All`	`== 0` Wait for a single trigger	
	`!= 0` Continuous wait for multiple triggers	

Returns

`== 0`	The wait was successful, a trigger was validated and processed.
`== 1`	None of the triggers in the group have been validated during the timeout duration. This will occur when the argument `Timeout` is non-negative. Either `Timeout` was zero and no triggers were validated, or `Timeout` was positive and none of the triggers were validated within `Timeout` number of timer ticks (or the component `TSKtimeoutKill()` was applied to the task blocked on the group).
`== 2`	All triggers in the group were deleted by another task using `GRPrmMBX`, `GRPrmMBX`, or `GRPrmAll` while waiting.
`== 3`	The group is already in use by another task
`== 4`	The group does not have any services attached to it

Component type

Function
- Cannot be used in an interrupt -

Options

N/A

Notes

If the argument `Timeout` specifies an infinite time and the argument `All` is non-zero (to continuously wait / block on the group) then unless another task deletes all triggers from the group `GRPwait()` is effectively performing an infinite loop.

See also

`OS_GROUP` (Section 4.1.7)
`GRPaddMBX()` (Section 6.11.4)
`GRPaddSEM()` (Section 6.11.5)
`GRPaddSEMbin()` (Section 6.11.6)
`GRPrm()` (Section 6.11.10)
`GRPrmAll()` (Section 6.11.9)
`TSKtimeoutKill()` (Section 6.3.30)

6.11.13 Grouping Examples

The following example implement a simple case of grouping, were two mailboxes and one semaphore are grouped together. Let's say the two mailboxes and the semaphore are controlled (through interrupts or by another task) by stimuli generated by an interface. The whole code example is listed and a section by section descriptions follows.

Table 6-23 Grouping Example

```
MBX_t Mbox1;                              /* Mailbox to attach to the group        */
MBX_t Mbox2;                              /* Mailbox to attach to the group        */
SEM_t MySema;                             /* Semaphore to attach to the group      */

void SemCB(SEM_t Arg);                    /* Callback funtion for MySema           */
void Mbx1CB(inptr_t *Msg);                /* Callback function for Mbox1           */
void Mbx2CB(intptr_t *Msg);               /* Callback function for Mbox2           */

GRP_t *MyGrp;

/* ------------------------------------------------------------------------------ */

void Task1(void)
{
int Err;
GRP_t *GrpErr;

   Mbox1  = MBXopen("Malbox #1, 32);      /* Create/open mailbox Mbox1             */
   Mbox2  = MBXopen("Malbox #2, 32);      /* Create/open mailbox Mbox2             */
   MySema = SEMopen("My Sema");           /* Create/open semaphore MySema          */

   MyGrp = GRPaddSEM(NULL, MySema, SemCB);  /* Create & attach MySema to the group   */
   if (MyGrp == (GRP_t *)NULL) {
       puts("Group allocation error");
    }
   GrpErr = GRPaddMBX(MyGrp, Mbox1, Mbx1CB);/* Attach Mbox1 to the group           */
   if (GrpErr == (GRP_t *)NULL) {
       puts("Group allocation error");
    }
   GrpErr = GRPaddMBX(MyGrp, Mbox2, Mbx2CB);/* Attach Mbox2 to the group           */
   if (GrpErr == (GRP_t *)NULL) {
       puts("Group allocation error");
    }

   do {
       "Config & enable the interface"
       Err = GrpWait(MyGrp, OS_TICK_PER_SEC, 0)
   } while (Err != 0);

   GRPwait(MyGrp, -1, 1);

   puts("The group was deleted");

   TSKselfSusp();
}

/* ------------------------------------------------------------------------------ */

void SemCB(SEM_t *Arg)
{
   Arg = Arg;
   puts("My semaphore was posted");
   return;
}

/* ------------------------------------------------------------------------------ */

void Mbx1CB(intptr_t Msg)
{
int Err;

   printf("Mailbox #1 got message 0x%08X\n", (int)Msg);
   return;
}

/* ------------------------------------------------------------------------------- */

void Mbx2CB(intptr_t Msg)
{
int Err;

   printf("Mailbox #2 got message 0x%08X\n", (int)Msg);
   return;
}
```

The two mailboxes and the semaphores are created / opened using the `MBXopen()` and `SEMopen()` components; that's Abassi standard way to create / open mailboxes and semaphores.

Table 6-24 Grouping Example (Section #1)

```
   Mbox1  = MBXopen("Malbox #1, 32);        /* Create/open mailbox Mbox1            */
   Mbox2  = MBXopen("Malbox #2, 32);        /* Create/open mailbox Mbox2            */
   MySema = SEMopen("My Sema");             /* Create/open semaphore MySema         */
```

Once the mailboxes and semaphore are created / opened, their descriptors are attached to the group named `MyGrp`. The first attachment has the first argument of `GRPadd`*`NNN`*`()` set to `NULL` as this is the beginning of the *"group construction"*. The other services attached after the first attachment use the returned value when the first attachment was performed. The semaphore service is assigned the `SemCB()` callback function and the two mailbox services are assigned the `Mbx1CB()` and `MBX2()` callback functions. The following code extract has been purged from the error messages to ease the understanding.

Table 6-25 Grouping Example (Section #2)

```
   MyGrp  = GRPaddSEM(NULL, MySema, SemCB); /* Create & attach MySema to the group   */
   GrpErr = GRPaddMBX(MyGrp, Mbox1, Mbx1CB);/* Attach Mbox1 to the group            */
   GrpErr = GRPaddMBX(MyGrp, Mbox2, Mbx2CB);/* Attach Mbox2 to the group            */
```

Once all three triggers have been attached to the group, the interface is configured and a timeout of 2 seconds is used (the second argument to `GRPwait()`). `GRPwait()` is informed to wait for a single trigger until expiry (third argument is set to 0). If no activity has been received from the interface (any of the three triggers were validated (return value is non-0), then the interface is configured again and the group waited on.

Table 6-26 Grouping Example (Section #3)

```
   do {
       "Config & enable the interface"
       Err = GrpWait(MyGrp, 2*OS_TICK_PER_SEC, 0)
   } while (Err != 0);
```

Once the interface has validated one of the three triggers, `GRPwait()` is called but this time informed to perform an infinite loop (second argument that specifies the timeout is negative and the third argument that specifies single vs. all trigger is non-zero). In case another task deletes all the triggers in the Group `MyGrp`, a task self-suspension (`TSKselfSusp()`) is inserted after `GRPwait()`.

Table 6-27 Grouping Example (Section #4)

```
GRPwait(MyGrp, -1, 1);

puts(“The group was deleted”);

TSKselfSusp();
```

6.12 Logging Services

The optional logging services allow the designer to get insight on what operations are performed in the kernel. It is a useful debugging tool, but be aware that when the logging services are enabled, the overall timing of the application could be affected as the logging adds extra code and CPU usage in the kernel.

There are two type of logging available:

- Direct writing to an ASCII output device;
- Recording in a circular buffer for later writing on an ASCII output device

6.12.1 Direct writing

The logging services are configured to perform a direct writing operation when the build option `OS_LOGGING_TYPE` (Section 4.1.17) is set to 1. This type of logging sends an ASCII string to the output device as soon as the operation that triggers the message occurs. This obviously translates into a big CPU impact on the operation of the kernel because every time a logging message is generated, all the operations required to generate the message occur in the middle of kernel operations.

There are two mechanisms to control the output of logging messages. One mechanism is an on/off switch, allowing or disallowing the logging output. The other mechanism controls the behavior of the individual messages: each message can be enabled or disabled.

Logging message output is turned off with the use of the component `LOGoff()` and it can be turned on with the component `LOGon()`. When the RTOS is started, the message output is always off. This means `LOGon()` must be used to start the output of the logging messages. The same applies with the individual messages: they are all disabled, therefore they must be turned on using either `LOGenb()` or `LOGallOn()`.

Individual messages are enabled with the component `LOGenb()` and disabled with the component `LOGdis()`. To simplify the use of `LOGenb()` and `LOGdis()`, the components `LOGallOn()` and `LOGallOff()` respectively enable all messages and disable all messages; this is the same as wrapping a loop around `LOGdis()` or `LOGenb()`.

Note: Refer to the description of the component `OSputchar()` (Section 6.14.5) as there is a major restriction about using direct logging when multithread safe libraries are used.

6.12.2 Buffer recording

Instead of directly writing to the output device at each key step operation performed in the kernel, the operation performed and its parameters can be memorized in a circular buffer. The recording in the circular buffer can be controlled to be continuous, overwriting older event occurrences, or to stop when the buffer is full. The circular buffer is enabled when the build option `OS_LOGGING_TYPE` (Section 4.1.17) is set to a value greater than 1. The value of `OS_LOGGING_TYPE` then specifies the number of event occurrence the circular buffer can hold. Continuous recording is enabled with the component `LOGcont()`; recording with stopping when the buffer is full is enabled with the component `LOGonce()`. The contents of the circular buffer can later be sent directly to the output device with the components `LOGdumpAll()` or `LOGdumpNext()`, or the buffer contents can be extracted and formatted, ready to be sent on any output devices, with the component `LOGgetNext()`.

6.12.3 Description

There are two control mechanisms when the logging service is part of the RTOS build. One is a global enable/disable and the other one enables/disables individual logging messages. The two mechanisms are completely independent. To globally turn on the printing or recording, the component `LOGon()` is used. To globally turn off the printing or recording, the component `LOGoff()` is used. The individual messages can be enabled with `LOGenb()` or all enable with `LOGallOn()`. They are individually disabled with `LOGdis()` or all disabled with `LOGallOff()`. When the application is started, with the logging feature part of the build, the logging is off by default, and all individual messages also disabled. If the logging uses the circular buffer, the recording mode is continuous, overwriting older event occurrences.

Table 6-28 Logging Service Component list

Section	Name	Description
6.12.4	`LOGallOff`	Disable all logging messages
6.12.5	`LOGallOn`	Enable all logging messages
6.12.6	`LOGcont`	With circular buffer, set up and start continuous recording
6.12.7	`LOGdis`	Disable a specific logging message
6.12.8	`LOGdumpAll`	Print the all the logging messages that were recorded
6.12.9	`LOGdumpNext`	Print the next logging message that was recorded
6.12.10	`LOGenb`	Enable the recording or printing of the logging messages
6.12.11	`LOGgetNext`	Get the next logging message that was recorded
6.12.12	`LOGoff`	Stop the recording or printing of the logging messages
6.12.13	`LOGon`	Turn on the recording or printing of the logging messages
6.12.14	`LOGonce`	Flush and start the recording of logging messages until explicitly stopped or, until the buffer is full

Note: The original RTOS design requirements were to ultimately have the capability to transfer “tokens” to an external monitoring device, and on that device, decode the tokens and print the logging in an ASCII form. As the names of the services are an integral part of the descriptor, it becomes fairly complex to try to transfer the information of the names to an external monitoring device. As such, the formatting in ASCII strings has to be performed directly in the application.

6.12.4 LOGallOff

Synopsis

```
#include "Abassi.h"

void LOGallOff(void);
```

Description

`LOGallOff()` is a logging component that configures the filter to disable the printing or recording of all messages. Individual messages can be disabled with the component `LOGdis()`. All messages can be re-enabled with the component `LOGallOn()`, or individually with the component `LOGenb()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

`LOGallOff()` activates the filter that selects which messages are allowed to be printed / recorded, and which are not. This is not the same operation the component `LOGoff()` performs. The latter disallows printing/recording, irrelevant of the filter configuration.

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.5 LOGallOn

Synopsis

```
#include "Abassi.h"

void LOGallOn(void);
```

Description

`LOGallOn()` is a logging component that configures the filter to enable the printing or recording of all messages. Individual messages can be enabled with the component `LOGenb()`. All messages can be disabled with the component `LOGallOff()`, or individually messages with the component `LOGdis()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

`LOGallOn()` activates the filter that selects which messages are allowed to be printed / recorded, and which are not. This is not the same operation the component `LOGon()` performs. The latter allows printing/recording, irrelevant of the filter configuration.

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.6 LOGcont

Synopsis

```
#include "Abassi.h"

void LOGcont(void);
```

Description

`LOGcont()` empties the circular buffer, sets the recording for continuous, and starts the recording. When the recording is configured in continuous mode, using `LOGcont()`, it means that oldest recordings are overwritten when the buffer is full.

Availability

Only available when the build option `OS_LOGGING_TYPE` is greater than one.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.7 LOGdis

Synopsis

```
#include “Abassi.h”

void LOGdis(int MsgNmb);
```

Description

`LOGdis()` configures the logging message filter to disable the message number specified with the argument `MsgNmb`. All messages can be disabled with the component `LOGallOff()`

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`MsgNmb`	Message number to disable

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.8 LOGdumpAll

Synopsis

```
#include "Abassi.h"

void LOGdumpAll(void);
```

Description

`LOGdumpAll()` stops the recording in the circular buffer, and then formats and sends all recorded messages to the output device, using the `OSputchar()` component. Once `LOGdumpAll()` has completed these operations, the recording remains disabled, but in the same recording mode (either continuous or one shot) and the recording buffer is declared empty. To restart the recording, one must use `LOGon()`, `LOGcont()` or `LOGonce()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is greater than one.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.9 LOGdumpNext

Synopsis

```
#include "Abassi.h"

void LOGdumpNext(void);
```

Description

`LOGdumpNext()` stops the recording in the circular buffer, and then formats and sends the oldest recorded message to the output device, using the `OSputchar()` component. The oldest message is then discarded from the recording buffer. The next use of the `LOGdumpNext()` will perform the same operation, but this time on the next oldest message. To restart the recording, one must use `LOGon()`, `LOGcont()` or `LOGonce()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is greater than one.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

N/A

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.10 LOGenb

Synopsis

```
#include "Abassi.h"

void LOGenb(int MsgNmb);
```

Description

`LOGenb()` configures the logging message filter to enable the message number specified with the argument `MsgNmb`. All messages can be enabled with the component `LOGallOn()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`MsgNmb`	Message number to enable

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.11 LOGgetNext

Synopsis

```
#include "Abassi.h"

const char *LOGgetNext(void);
```

Description

`LOGgetNext()` stops the recording in the circular buffer, and then formats and return an ASCII string for the oldest recorded message. The oldest message is then discarded from the recording buffer. The next use of the `LOGgetNext()` will perform the same operation, but this time on the next oldest message. To restart the recording, one must use `LOGon()`, `LOGcont()` or `LOGonce()`.

Availability

Only available when the build option `OS_LOGGING_TYPE` is greater than one.

Arguments

`void`

Returns

Pointer to a formatted string

Component type

Function

Options

N/A

Notes

N/A

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.12 LOGoff

Synopsis

```
#include "Abassi.h"

void LOGoff(void);
```

Description

`LOGoff()` sets the logging facilities to stop the printing/recording of messages.

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.13 LOGon

Synopsis

```
#include "Abassi.h"

void LOGon(void);
```

Description

`LOGon()` sets the logging facilities to restart the printing/recording of messages.

Availability

Only available when the build option `OS_LOGGING_TYPE` is non-zero.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.14 LOGonce

Synopsis

```
#include "Abassi.h"

void LOGonce(void);
```

Description

`LOGonce()` empties the circular recording buffer, sets the recording for one shot, meaning the recording stops when the circular buffer is full, and then starts the recording.

Availability

Only available when the build option `OS_LOGGING_TYPE` is greater than one.

Arguments

`void`

Returns

`void`

Component type

Function

Options

N/A

Notes

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.12.15 Logging Messages Numbers

This section lists and described all the logging messages supported by the logging facilities. When formatted, each one of the message is preceded by `“[NNN] “`, where `NNN` indicates the current timer tick counter value.

Numbers are given instead of tokens. The decision to not use tokens is that, one way or another, using tokens forces the designer to look-up the token names. By indicating the message number, it gives the freedom to the designer to define their own tokens, which will have more meaningful names, as the way to name tokens is always a personal preference.

When logging is enabled, naming of descriptors is enabled, even if the build option `OS_NAMES` is set to a value of zero. If a descriptor has been given no name, the descriptor name will show up as `“…”`. Also, as each task possesses a private semaphore, the private semaphore shows up as “Priv”

6.12.15.1 Semaphores

Message 0:	`ISR posting semaphore “SSS"` This message indicates the semaphore `SSS` is posted in an interrupt handler.
Message 1:	`“SSS” semaphore posted by “TTT”` This message indicates the semaphore `SSS` is posted by task `TTT`.
Message 2:	`“TTT” is unblocked from “SSS”` This message indicates the task `TTT`, that was blocked on the semaphore `SSS`, gets unblocked due to the semaphore posting.
Message 3:	`“TTT” to wait on semaphore “SSS”` This message indicates the task `TTT` is trying to wait on semaphore `SSS`.
Message 4:	`“TTT” not blocked on semaphore “SSS” (timeout==0)` This message indicates the task `TTT` is not blocking on semaphore `SSS`, even though the semaphore was not posted, as the requested timeout specified in `SEMwait()` was zero.
Message 5:	`“TTT” blocks on semaphore “SSS”` This message indicates the task `TTT` is getting blocked on semaphore `SSS`.
Message 6:	`“TTT” not blocked on semaphore “SSS” (was posted)` This message indicates the task `TTT` is not blocking on semaphore `SSS` as the semaphore was previously posted.

6.12.15.2 Mailboxes

Message 7:	`ISR reading mailbox “MMM”` This message indicates the mailbox `MMM` is read in an ISR handler.
Message 8:	`”TTT” reading mailbox “MMM”` This message indicates the mailbox `MMM` is being read by task `TTT`.
Message 9:	`“MMM” contained value 0xNNNN` This message indicates what value was read from the mailbox.
Message 10:	`“TTT” blocks on mailbox MMM (empty)`

This message indicates the task `TTT` is reading mailbox `MMM`, but as the mailbox is empty, the task gets blocked.

Message 11: `"TTT" not blocked on mailbox MMM (timeout==0)`
This message indicates the task `TTT` is reading mailbox `MMM`, and even though the mailbox is empty, the task does not get blocked as the requested timeout specified in `MBXget()` was zero.

Message 12: `ISR writing mailbox "MMM"`
This message indicates the mailbox `MMM` is written in an ISR handler.

Message 13: `Mailbox "MMM" write by "TTT"`
This message indicates mailbox `MMM` is written by task `TTT`.

Message 14: `0xNNNN written into mailbox "MMM"`
This message indicates the value written in the mailbox.

Message 15: `Mailbox "MMM" full, "TTT" blocks`
This message indicates task `TTT` is getting blocked, as the mailbox `MMM` is full.

Message 16: `Mailbox "MMM" full, 0xNNNN not written`
This message indicates the mailbox `MMM` did not get written. This could be due to the requested timeout specified in `MBXget()` being zero, or the mailbox write occurred in an ISR handler.

6.12.15.3 Timer

Message 17: `"TTT" added to timeout list, expiry tick is NNNN`
This message indicates task `TTT` is blocked and inserted in the time-out linked list. If the task does not get normally unblocked, the timeout will unblock the task when the timer tick counter reaches or exceeds `NNNN`.

Message 18: `"TTT" removed from timer list`
This message indicates the task `TTT` is removed from the time-out linked list. This can be due to the task getting unblocked on the mechanism it was blocked on, or because the timeout time was exceeded.

Message 80: `New timeout on task "TTT" : NNNN`
This message indicates the timeout task `TTT` is blocked until expiry is set to the new value NNN. This is a result of the use of `TSKtout()` or `TSKtimeoutKill()` components.

6.12.15.4 Priority / Running

Message 19: `"TTT" set to priority NNN`
This message indicates the task `TTT` has its priority changed. This can be due to priority inversion protection, starvation protection, or a request through the component `TSKprio()`.

Message 20: `"TTT" is running at priority NNN`
This message indicates the task `TTT` is now the running task; a context switch has occurred.

6.12.15.5 Timer Services

Message 21: `Timer service "XXX" has expired`
This message indicates the timer `XXX` has expired, therefore the operation attached to the timer service is performed.

Message 22: `Timer service "XXX" removed from the list`
This message indicates the timer service `XXX` was deactivated. This could be due to a single shot operation that was performed, or the timer service, which was already active, being re-programmed, or the component `TIMkill()` being applied on the timer service.

Message 23: `Timer service "XXX" added to list, expiry tick NNN`
This message indicates the timer service `XXX` is deactivated. This could be due to a periodic operation that was performed, or the timer service, which as already active, being re-programmed.

6.12.15.6 Event Flags

Message 24: `0xNNNN set in events flags of "TTT"`
This message indicates the running task is setting the events flags of task `TTT`.

Message 25: `ISR set value 0xNNNN in events flags of "TTT"`
This message indicates an ISR handler is setting the events flags of task `TTT`.

Message 26: `"TTT" unblocked by event flags`
This message indicates the task `TTT` has its priority changed. This can be due to priority inversion protection, starvation protection, or a request through `TSKprio()`.

Message 27: `"TTT" not blocked by event flags NNN`
This message indicates the task `TTT` getting its event flags is not getting blocked as the event flags `NNN` fulfill the condition masks specified in `EVTwait()`.

Message 28: `"TTT" blocked on event flags NNN`
This message indicates the task `TTT` getting its event flags is getting blocked as the current event flags `NNN` do not fulfill the condition masks specified in `EVTwait()`.

6.12.15.7 State changes

Message 29: `"TTT" running (round-robin), time-slice NNNN ticks`
This message indicates the task `TTT` is now running due to round robin. The maximum run-time duration is indicated by `NNNN`.

Message 30: `"TTT" is requested to be suspended`
This message indicates the task `TTT` is going to be suspended. The suspension could be delayed as long as the task locks one or more mutexes.

Message 31: `"TTT" is now suspended`
This message indicates the task `TTT` is now suspended

Message 32: `"TTT" is requested to be resumed`
This message indicates the task `TTT` is requested to be resumed. If the task is in the suspended state, then it resumes. If the task is not in the suspended state, the pending suspension request is dropped.

Message 33: `"TTT" is yielding CPU`
This message indicates the task `TTT` is yielding the CPU. This message only applies when the RTOS is built to operate in cooperative mode.

Message 34: `"TTT" is not yielding CPU, no other ready to run`
This message indicates the task `TTT` is not yielding the CPU. This occurs when no tasks of equal or higher priority are in the ready to run state. This message only applies when the RTOS is built to operate in cooperative mode.

6.12.15.8 Starvation Protection

Message 35: `"TTT" added to starvation list`
This message indicates the task `TTT` is now under starvation protection.

Message 36: `"TTT" back to pre-starvation priority NNN`
This message indicates the task `TTT` ran long enough under starvation protection and is going back to its original priority

Message 37: `"TTT" removed from starvation list`
This message indicates the task `TTT` is removed from the starvation list. This can be due to it running normally, that it has run long enough under the starvation protection, or the task state changed.

6.12.15.9 Priority Inversion

Message 38: `"TTT" new priority NNN (Prio invert protection)`
This message indicates the task `TTT` has its priority changed as it locks a mutex and is under priority inversion protection.

6.12.15.10 Stack monitoring

Message 39: `"TTT" stack overflow TOS address at NNN`
This message indicates the task `TTT` has used more stack than was allocated to it. The application get frozen

6.12.15.11 Memory Block management

Message 40: `ISR requesting buffer from memory management "MMM"`
This message indicates a buffer from the memory management `MMM` is being requested in an ISR handler.

Message 41: `"TTT" requesting buffer form memory management "MMM"`

This message indicates a buffer from the memory management `MMM` is being requested by task `TTT`.

Message 42: `"MMM" returned buffer 0xNNNN`
This message indicates the pointer of the buffer value was obtained from the memory management.

Message 43: `"TTT" blocks on memory management MMM (empty)`
This message indicates the task `TTT` is requesting a buffer from the memory management `MMM`, but as the memory pool is empty, the task gets blocked.

Message 44: `"TTT" not blocked on mailbox MMM (timeout==0)`
This message indicates the task `TTT` is requesting a buffer from the memory management `MMM`, and even though the memory pool is empty, the task does not get blocked as the requested timeout specified in `MBXalloc()` was zero.

Message 45: `Buffer return to memory pool "MMM" by "TTT"`
This message indicates a buffer is returned to the memory pool `MMM` is by task `TTT`.

Message 46: `Buffer 0xNNNN returned to memory pool "MMM"`
This message indicates the pointer of the buffer returned to the memory pool `MMM`.
This message will not be sent out if the pointer to the buffer to return in `NULL`.

6.12.15.12 SMP multi-core

Message 47: `Sending ISR to core #N for load balancing`
This message indicates a core is sending an interrupt to another core because a task switch must occur on the target core (core #`N`).

Message 48: `Performing load balancing check`
This message indicates kernel is current performing load balancing as a task switch is about to happen on one or multiple cores.

Message 49: `Got an ISR requesting to check load balancing`
This message indicates a core has received an interrupt from another core because a task switch should occur on it.

Message 50: `Task "TTT" remains running`
This message indicates there is no task switching after the load balancing was performed

6.12.15.13 Out of Memory Checks

Message 51: `Out of alloc memory (increase OS_ALLOC_SIZE)`
This message indicates the application has run out of memory used by `OSalloc()`.

Message 52: `Out of heap memory`
This message indicates the application has run out of memory used by `malloc()`. This implies the heap area size should be increased.

Message 53: `Out of mailboxes (increase OS_STATIC_MBX)`
This message indicates the application has run out of mailbox descriptors allocated by `OS_STATIC_MBX`.

Message 54: `Out of mailboxes buffer(increase OS_STATIC_BUF_MBX)`

This message indicates the application has run out of mailbox memory buffer that was reserved by `OS_STATIC_BUF_MBX`.

Message 55: `Out of memory blocks (increase OS_STATIC_MBLK)`
This message indicates the application has run out of memory block descriptors allocated by `OS_STATIC_MBLK`.

Message 56: `Out of memory block buffer memory (increase OS_STATIC_BUF_MBLK)`
This message indicates the application has run out of memory block buffer that was reserved by `OS_STATIC_BUF_MBLK`.

Message 57: `Out of name memory (increase OS_STATIC_NAME)`
This message indicates the application has run out of memory needed to hold the names of all services and tasks, which was reserved by `OS_STATIC_NAME.`

Message 58: `Out of semaphores (increase OS_STATIC_SEM)`
This message indicates the application has run out of semaphore/mutex descriptors allocated by `OS_STATIC_SEM`.

Message 59: `Out of stack memory (increase OS_STATIC_STACK)`
This message indicates the application has run out of memory needed supply all the tasks stacks, which was reserved by `OS_STATIC_STACK`.

Message 60: `Out of tasks (increase OS_STATIC_TASK)`
This message indicates the application has run out of task descriptors allocated by `OS_STATIC_TASK`.

Message 61: `Out of timers (increase OS_STATIC_TIM_SRV)`
This message indicates the application has run out of timer service descriptors allocated by `OS_STATIC_TIM_SRV`.

Message 62: `ISR queue overflow (increase OS_MAX_PEND_RQST)`
This message indicates the application has run out of room in the queue used to collect the kernel requests during interrupts. The size of the queue is specified by `OS_MAX_PEND_RQST`.

Message 63: `Out of groups (increase OS_GROUP)`
This message indicates the application has run out of the trigger descriptors used by groups. The number of pre-allocated trigger descriptors is defined by `OS_GROUP` set to a positive value.

6.12.15.14 Group messages

Message 64: `Aborting wait on group for task "TTT" owner is "ZZZ"`

This message indicates the call by task `TTT` to `GRPwait()` cannot be fulfilled because the task `ZZZ` is already waiting on the group.

Message 65: `Mailbox "MMM" already attached to another group`

This message reports the mailbox `MMM` is requested to be attached to a group through `GRPaddMBX()` component but the operation is aborted because the mailbox is already attached to another group.

Message 66: `Semaphore "SSS" already attached to another group`

This message reports the semaphore `SSS` is requested to be attached to a group through `GRPaddSEM()` or `GRPaddSEMbin()` components but the operation is aborted because the semaphore is already attached to another group.

Message 67: `"TTT" blocks on group`

This message reports the task `TTT` is performing a call to `GRPwait()` and becomes blocked as none of the triggers in the group are valid.

Message 68: `"TTT” not blocking on group, data in mailbox "MMM"`

This message reports the task `TTT` is performing a call to `GRPwait()` and does not block as the mailbox `MMM` , which is attached to the group, is not empty.

Message 69: `“TTT” not blocking on group, semaphore “SSS" is posted`

This message reports the task `TTT` is performing a call to `GRPwait()` and does not block as the semaphore `SSS` , which is attached to the group, is already posted.

Message 70: `Removing mailbox \"%s\" from group`

This message indicates a request through `GRPrm()` or `GRPrmAll()` to remove the trigger the mailbox `MMM` is attached to.

Message 71: `Removing semaphore \"%s\" from group`

This message indicates a request through `GRPrm()` or `GRPaddSEMbin()` to add the semaphore `SSS` to a group as one of the triggers.

Message 72: `Adding mailbox \"%s\" to group`

This message indicates a request through `GRPaddMBX()`to add the mailbox `MMM` to a group as one of the triggers.

Message 73: `Adding semaphore \"%s\" to group`

This message indicates a request through `GRPasSEM()` or `GRPaddSEMbin()` to add the semaphore `SSS` to a group as one of the triggers.

Message 74: `Mailbox \"%s\" unblocking task \"%s\" waiting on group`

A request to the component `MBXput()` on the mailbox `MMM`, which is attached to a group, unblocks task `TTT` that was waiting on the group through the use of the component `GRPwait()`.

Message 75: `"Semaphore \"%s\" unblocking task \"%s\" waiting on group",`

A request to the component `SEMpost()` on the semaphore `SSS`, which is attached to a group, unblocks task `TTT` that was waiting on the group through the use of the component `GRPwait()`.

6.12.15.15 Wait Abort messages

Message 76: `Aborting wait on semaphore “SSS” for task “TTT",`

This message reports a request to `SEMabort()` is provoking the unblocking of task `TTT`.

Message 77: `Aborting wait on mutex “MMM” for task “TTT”,`

This message reports a request to `MTXabort()`is provoking the unblocking of task `TTT`.

Message 78: `Aborting wait on events for task "TTT"`

This message reports a request to `EVTabort()`is provoking the unblocking of task `TTT`.

Message 79: Aborting wait on mailbox "MMM" for task "TTT"

This message reports a request to `MBXabort()`is provoking the unblocking of task `TTT`.

6.12.15.16 Timeout messages

Message 80: `New timeout on task "TTT": ###`

This message reports a request to `TSKtout()` or `TSKtoutKill()` to change the timeout on the task `TTT`., which is blocked with an expiry time-out. The new timeout is the value ### expressed in number of timer ticks and when that value is 0 the request triggers an instant expiry.

6.12.15.17 Mutex deadlock messages

Note: all three (3) log messages, #81, #82, and #83, for deadlock detection should be enabled to see the complete report on the tasks and the mutexes involved in the deadlock condition. As a reminder, a mutex deadlock occurs when task "T1" tries to lock the mutex "M1" when task "T2" has a lock on it. At the same time, task "T2" is blocked on the mutex "M2", which is locked by task "T1" ("T1" is the task trying to obtain the lock on "M1").

Message 81: `Deadlock – task "TTT" trying to lock mutex "MMM"`

This message reports the task (`"TTT"`) that would create a deadlock when trying to lock mutex `"MMM".` In the reminder example, they are the task "T1" and the mutex "M1"

Message 82: `Deadlock – mutex "MMM" is locked by task "TTT"`

This message is used twice to report both existing locks that are the inner cause of the deadlock, i.e. which task locks which mutex, In the reminder example they are the pairs "T2" - "M1" and "T1" - "M2"

Message 83: `Deadlock – task "TTT" is blocked on mutex "MMM"`

This message reports the "back lock" involved in the deadlock. In the reminder example, they are task T2 and mutex M2.

6.12.16 Logging examples

6.12.16.1 Direct writing

Logging with direct writing is straight forward, as the only controls are the On/Off, and individual message enable/disable. The following table shows a typical use of the logging facilities when configured in direct write:

Table 6-29 Direct writing Example

```
#include "Abassi.h"

…

   LOGallOff();                           /* Disable all messages                     */
   LOGenb(24);                            /* Enable event set message                 */
   LOGenb(25);                            /* Enable event set in ISR message          */
   LOGenb(26);                            /* Enable unblocking by events message      */

   …

   LOGon();                               /* Allow the writing to the output device   */

   …

   LOGoff();                              /* Stop the writing to the output device    */


   LOGdis(25);                            /* Disable the event set in ISR message     */

   LOGenb(17);                            /* Enable the timer added in list           */
   LOGenb(18);                            /* Disable the timer removed from list      */

   …

   LOGon();                               /* Allow the writing to the output device   */
```

6.12.16.2 Circular buffer

Reusing the previous example, but this time using the circular buffer:

Table 6-30 Circular writing Example

```
#include “Abassi.h”

…

   LOGallOff();                       /* Disable all messages                     */
   LOGenb(24);                        /* Enable event set message                 */
   LOGenb(25);                        /* Enable event set in ISR message          */
   LOGenb(26);                        /* Enable unblocking by events message      */

   …

   LOGonce();                         /* Configure the logging to stop when full   */
                                      /* and start the recording                  */
   …

   LOGoff();                          /* Stop the recording                       */


   LOGdis(25);                        /* Disable the event set in ISR message     */

   LOGenb(17);                        /* Enable the timer added in list           */
   LOGenb(18);                        /* Disable the timer removed from list      */

   …

   LOGon();                           /* Allow the recoriding                     */

   …

   LOGoff();                          /* Stop the recording                       */
   LOGdumpAll();                      /* Format & send out the buffer             */

   …

   LOGcont();                         /* Clear the buffer & set in continuous mode */
                                      /*and start the recording                   */
   …

   LOGoff():                          /* Stop the recoring                        */

                                      /* Retrieve the recoderded information      */
   while (NULL != (String = LOGgetNext)) {
       Output(String);                /* Output to any ASCII device               */
   }
```

6.13 Performance Monitoring

Performance monitoring is a facility that collects the operational statistics of all tasks in an application and it can be added in Abassi through the setting of the build option `OS_PERF_MON` (See section 4.1.31). When the performance monitoring facilities are added in Abassi, the file `PerfMon.c`, supplied in the distribution, must be included in the build process. The statistics collection is performed by the code in file `PerfMon.c` and not inside Abassi.c. This approach was retained to eliminate all risks of breaking the Abassi kernel code if the performance monitoring needs to be customized for an application.

6.13.1 Description

The performance monitoring collects, in real-time, 4 key statistics on the tasks operations:

- Latency between being unblocked and becoming running
- Run time from unblocked to blocked
- Elapsed time from unblocked to blocked
- Pre-emption time

The last run, maxima, minima and averages are computed for all 4 statistics measurements. A diagram with changes of state of a task is shown in the following figure:

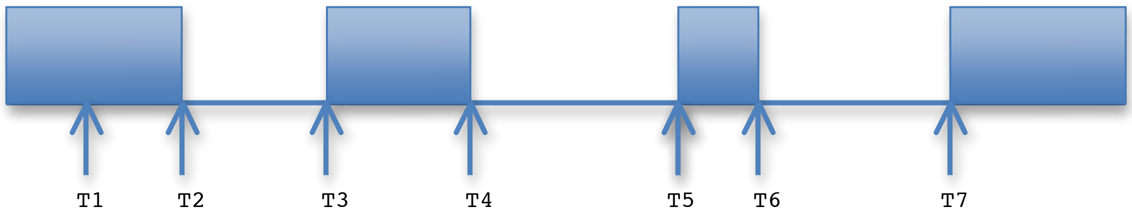

Figure 6-1 Performance Metrics Measurements

Each time marker (T#) corresponds to the following change of state of a task:

`T1`	: The task is unblocked, ready to run
`T2`	: The task starts to run
`T3`	: The task is pre-empted by another task
`T4`	: The task is resumed
`T5`	: The task is pre-empted by another task
`T6`	: The task is resumed
`T7`	: The task gets blocked

Using the above figure, the 4 key statistics are measured as flows:

Latency:	The latency is defined by the time elapsed from the task getting unblocked (inside the kernel) to when it starts running. This is the time between `T1` and `T2` in the figure above.
Run Time:	The run time is the total time a task is in the running task. This is the total time between `T2` and `T3`, plus the time between `T4` and `T5`, plus the time between `T6` and `T7` in the figure above.
Elapsed Time:	The elapsed time is the time elapsed between when the task starts to run after being unblocked until it gets blocked. In the above figure, it is the time between `T2` and `T7`.
Pre-emption Time:	The pre-emption time is the individual time spans when a task is preempted; it is not the total time a task is pre-empted during the elapsed time from being unblocked to

being blocked. The pre-emption times are the time between `T3` and `T4` or the time between `T5` and `T6`. If the total pre-emption time is desired, it can be obtained using the difference between the elapsed time and the run time.

Note: The time spent in the interrupt handler is not taken off from these measurements. It is assumed interrupt handler should be as short as possible; as such, their impact on the overall performance of the application should be negligible.

When performance monitoring is part of the build with the build option `OS_PERF_MON` set to a non-zero value, all tasks upon creation are immediately monitored. It is possible to stop and restart the statistics collection with the components `PMstop()` (Section 6.13.4) and `PMrestart()` (Section 6.13.3).

6.13.2 Measurements

The following table lists all the entries held in the task descriptors. Depending on the timer/counter selected as the performance monitoring time reference, either the RTOS timer tick (`G_OStimCnt`) or the port specific timer, the data type `OSperfMon_t` has a different word length. When using the RTOS timer tick, `OSperfMon_t` is an `int`; when using the port specific timer, the `OSperfMon_t` data type is typically 64 bits wide.

The average measurements are computed using a "low-pass" filter:

Table 6-31 Computation of the average statistics

```
if n == 0                        /* First measurement          */
   Average(0) = NewValue(0)
else
   Average(n) = 63*Average(n-1)/64 + NewValue(n)/64
```

The following table lists and describes each of the entries in a task descriptor that holds the run-time statistics.

Table 6-32 Performance Monitoring Task Descriptors entries

Name	Type	Description
`PMcumul[]`	`OSperfMon_t`	Cumulative run time of the task since its creation or since the statistics have been reset. On single core targets the array is dimensioned to 1 and on multi-core, each entry is the run time on each core. The name is new in Abassi version 1.278.266 and mAbassi version 1.112.111. For backward compatibility, the Abassi original entry name `PMtotalRun` and mAbassi `PMcoreRun[]` are still available as "C" defines.
`PMstartTick`	`OSperfMon_t`	Performance monitoring timer tick when the was started.
`PMlastTick`	`OSperfMon_t`	Tick counter value when the measurement was stopped.
`PMlatentLast`	`OSperfMon_t`	Last latency time measurement
`PMlatentMin`	`OSperfMon_t`	Shortest latency time of the task since its creation or since the statistics have been reset.
`PMlatentMax`	`OSperfMon_t`	Longest latency time of the task since creation or since the statistics have been reset.
`PMlatentAvg`	`OSperfMon_t`	Average latency time of the task since creation or since the statistics have been reset.

`PMlatentStrt`	`OSperfMon_t`	Timer/counter value when the task got unblocked
`PMaliveLast`	`OSperfMon_t`	Last elapsed time measurement
`PMaliveMin`	`OSperfMon_t`	Shortest elapsed time of the task since its creation or since the statistics have been reset.
`PMaliveMax`	`OSperfMon_t`	Longest elapsed time of the task since creation or since the statistics have been reset.
`PMaliveAvg`	`OSperfMon_t`	Average elapsed time of the task since creation or since the statistics have been reset.
`PMaliveStrt`	`OSperfMon_t`	Timer/counter value when the task started running after being unblocked
`PMrunLast`	`OSperfMon_t`	Last run time measurement
`PMrunMin`	`OSperfMon_t`	Shortest run time of the task since its creation or since the statistics have been restarted.
`PMrunMax`	`OSperfMon_t`	Longest run time of the task since creation or since the statistics have been restarted.
`PMrunAvg`	`OSperfMon_t`	Average run time of the task since creation or since the statistics have been restarted.
`PMrunCum`	`OSperfMon_t`	Accumulated run time since the task started after being unblocked. Needed to not take into account the time the task is pre-empted
`PMrunStrt`	`OSperfMon_t`	Timer/counter value when the task started running after being unblocked
`PMpreemLast`	`OSperfMon_t`	Last pre-emption time measurement
`PMpreemMin`	`OSperfMon_t`	Shortest time the task has been preempted since its creation or since the statistics have been restarted.
`PMpreemMax`	`OSperfMon_t`	Longest time the task has been preempted since its creation or since the statistics have been restarted.
`PMpreemAvg`	`OSperfMon_t`	Average time the task has been preempted since its creation or since the statistics have been restarted.
`PMpreemStrt`	`OSperfMon_t`	Timer/counter value when the task started being pre-empted
`PMpreemCnt`	`uint32_t`	Number of times the task has been preempted since its creation or since the statistics have been restarted.
`PMblkCnt`	`uint32_t`	Number of times the task has been blocked since its creation or since the statistics have been restarted.
`PMsemBlkCnt`	`uint32_t`	Number of times the task has been blocked on semaphores since its creation or since the statistics have been restarted.
`PMmtxBlkCnt`	`uint32_t`	Number of times the task has been blocked on mutexes since its creation or since the statistics have been restarted.
`PMevtBlkCnt`	`uint32_t`	Number of times the task has been blocked on its events since its creation or since the statistics have been restarted. This entry is only available if the build option `OS_EVENTS` is non-zero
`PMgrpBlkCnt`	`uint32_t`	Number of times the task has been blocked on groups since its creation or since the statistics have been restarted. This entry is

		only available if the build option `OS_GROUP` is non-zero
`PMmbxBlkCnt`	`uint32_t`	Number of times the task has been blocked on mailboxes since its creation or since the statistics have been restarted. This entry is only available if the build option `OS_MAILBOX` is non-zero
`PMstrvCnt`	`uint32_t`	Number of times the task has been under the starvation protection mechanism since its creation or since the statistics have been restarted. The count indicates how many times the task was put under protection, not how many time it ran due to starvation protection. This entry is only available if the build option `OS_STARVE_WAIT_MAX` is non-zero
`PMstrvRun`	`uint32_t`	New in Abassi version 1.278.266 and mAbassi version 1.112.111. Number of times the task has run with its priority raised by the starvation protection mechanism since its creation or since the statistics have been restarted. This entry is only available if the build option `OS_STARVE_WAIT_MAX` is non-zero
`PMstrvRunMax`	`uint32_t`	New in Abassi version 1.278.266 and mAbassi version 1.112.111. Number of times the task has run the maximum time allowed with its priority raise under the starvation protection mechanism since its creation or since the statistics have been restarted. This entry is only available if the build option `OS_STARVE_WAIT_MAX` is non-zero
`PMinvertCnt`	`uint32_t`	Number of times the task has its priority changes for mutex inversion protection since its creation or since the statistics have been restarted. This entry is only available if the build option `OS_MTX_INVERSION` is non-zero
`PMcontrol`	`int`	Internally used by the performance monitoring to know the state of the statistic collection (stopped, armed or running).
`PMretart`	`int`	Internally used by the performance monitoring to control the reset of the statistics. Set to a non-zero value and all task statistics will get reset the next time the task gets blocked/preempted or becomes running.

6.13.3 PMrestart

Synopsis

```
#include “Abassi.h”

void PMretart(TSK_t *Task);
```

Description

`PMrestart()` reset and restart the collection of statistics on the task specified by the argument `Task`. Once the statistics collection is restarted, the performance monitoring goes into an “armed” state for the collection and will start the collection only when the task transits from a blocked or ready-to-run state to the running state.

Availability

Only available when the build option `OS_PERF_MON` is non-zero.

Arguments

`Task`	Descriptor of the task to restart the statistics collection.

Returns

`void`

Component type

Macro (unsafe)

Options

N/A

Notes

See also

`OS_PERF_MON` (Section 4.1.31)
`PMstop` (Section 6.13.4)

6.13.4 PMstop

Synopsis

```
#include "Abassi.h"

void PMstop(TSK_t *Task);
```

Description

`PMstop()` stop the collection of statistics on the task specified by the argument `Task`. And freezes the resuls. Once the statistics collection is stopped, they can only be restarted meaning the collection start fresh, dropping pas measurements. If the collection is already stopped, using this component has not effect.

Availability

Only available when the build option `OS_PERF_MON` is non-zero.

Arguments

`Task`	Descriptor of the task to stop the statistics collection.

Returns

`void`

Component type

Macro (unsafe)

Options

N/A

Notes

See also

`OS_PERF_MON` (Section 4.1.31)
`PMretart` (Section 6.13.3)

6.14 Mix Bag

This section describes components that don't really fit in any category of services.

6.14.1 G_OSmutex

Synopsis

```
#include "Abassi.h"

MTX_t *G_OSmutex;
```

Description

`G_OSmutex` is the sole mutex the Abassi RTOS uses to protect the access of all shared resources it handles. These shared resources are only accessed by Abassi when components are created.

Availability

Always

Arguments

N/A

Returns

N/A

Component type

Mutex descriptor

Options

N/A

Notes

`G_OSmutex` is the mutex one should use to protect non- multithread-safe functions in the standard "C" libraries. The reason is if `malloc()` is the memory allocator selected to be used by Abassi, then the RTOS protects `malloc()` with this mutex. Therefore other calls to `malloc()` should be protected with the same mutex. And, as an extension, many other functions in the standard "C" library.

See also

`OS_ALLOC_SIZE` (Section 4.1.1)
`OSalloc` (Section 6.14.3)

6.14.2 G_OSnoName

Synopsis

```
#include “Abassi.h”

const char G_OSnoName[];
```

Description

`G_OSnoName` is the preferable character string to use for unnamed service. This is preferable to using NULL as some `printf()` implementations do not detect the NULL pointer and print gibberish (data at address 0) when encountering a NULL pointer for a string to print.

Availability

Always

Arguments

N/A

Returns

N/A

Component type

Variable

Options

Notes

See also

6.14.3 OSalloc

Synopsis

```
#include "Abassi.h"

void *OSalloc(size_t Size);
```

Description

`OSalloc()` is the dynamic memory allocator component internally used by the Abassi RTOS. It behaves the same as the standard "C" `malloc()`.

Availability

Always, but see **Options** below

Arguments

`Size`	Size in `char` of the memory block to allocate

Returns

Pointer to the memory block allocated by `OSalloc()`

Component type

Definition
- Cannot be used in an interrupt -

Options

The build option `OS_ALLOC_SIZE` controls two aspects of `OSalloc()`. If the build option `OS_ALLOC_SIZE` is set to a value of zero, then `OSalloc()` is exactly the same as the standard "C" function `malloc()`, being defined as such, but protected with the mutex `G_OSmutex`. If the build option `OS_ALLOC_SIZE` is positive, then the value the build option is set to is the amount of memory reserved at compile/link time for `OSalloc()`. `OSalloc()` then performs true memory allocation, pulling memory blocks from the memory that was reserved; it is not anymore mapped to `malloc()`, but the allocator is still protected by the mutex `G_OSmutex`.

Notes

To minimize the impact on real-time operation, `OSalloc()` is a very simple definition, so no checks are performed to verify if the memory reserved, as indicated by `OS_ALLOC_SIZE`, has been exhausted.
The memory blocks retuned by `OSalloc()` are always aligned to the largest possible data type (aligned on 8 bytes) in order to fulfill the requirements of some processors. This means some extra memory should be added to the strict minimum memory allocation requirements of the application if the memory is not always allocated in block size multiple of 4 or 8.
The memory reserved by `OS_ALLOC_SIZE` has no relation with the memory reserved by any of the `OS_STATIC_XXX` build options.

`OSalloc()` is the preferred way to allocate dynamic memory as it is always protected by a mutex, and this is true even when `OS_ALLOC_SIZE` is set to zero, making `malloc()` the memory allocator.

See also

`OS_IDLE_STACK` (Section 4.1.16)
`OS_ALLOC_SIZE` (Section 4.1.1)
`OS_STATIC_BUF_MBX` (Section 4.1.45)
`OS_STATIC_MBX` (Section 4.1.47)
`OS_STATIC_NAME` (Section 4.1.48)
`OS_STATIC_SEM` (Section 4.1.49)
`OS_STATIC_STACK` (Section 4.1.50)
`OS_STATIC_TASK` (Section 4.1.51)
`G_OSmutex` (Section 6.14.1)

6.14.4 OSallocAvail

Synopsis

```
#include “Abassi.h”

int OSallocAvail(void);
```

Description

`OSallocAvail()` is a component that reports how many char are left in the RTOS static memlory allocator

Availability

`OSallocAvail()` is only available when the build option `OS_ALLOC_SIZE` is non-zero in. It is not available in any releases before Abassi version 1.273.262 and mAbassi version 1.94.97.

Arguments

`Mbox`	Descriptor of the mailbox to report the number of free elements.

Returns

Size in byte available for future allocation.

Component type

Atomic macro (safe)

Options

Notes

See also

`OS_ALLOC_SIZE` (Section 4.1.1)
`OSalloc()` (Section 6.14.3)

6.14.5 OSputchar

Synopsis

```
#include “Abassi.h”

int OSputchar(int Character);
```

Description

`OSputchar()` is a definition for the character output interface to use for ASCII logging, which is enabled when the build option `OS_LOGGING_TYPE` is set to 1.

Availability

Only available/needed when the build option `OS_LOGGING_TYPE` is 1.

Arguments

`Character`	Single character to send on the output device.

Returns

N/A

Component type

Definition

Options

Notes

`OSputchar()` is by default mapped to the standard “C” I/O function `putchar()`.
If a different I/O interface is used in the application, simply replace the definition in the file `Abassi.h`.
For libraries that have a mutex based multithreading protection for reentrance, the multithreading reentrance protection is disabled when using the standard I/O inside the kernel. This was necessary to eliminate an infinite re-entrance in the kernel, as having the multi-threading protection active in the kernel would lock / unlock a mutex. This would create a call to the kernel from within the kernel, and on and on. The multi-threading protection being temporary turned off, it means it is highly probable the logging facilities will corrupt the printed text; there should not be any crash issues though.

See also

`OS_LOGGING_TYPE` (Section 4.1.17)

6.14.6 OStrap

Synopsis

```
#include "Abassi.h"

int OStrap(int Error);
```

Description

`OStrap()` is a leaf function that never returns. It is called by the RTOS when an un-recoverable error is encountered. The error is described by the value of argument `Error`.

`OStrap()` supports multiple type of error trapping, and what checks are performed depends on the setting of build options. The following table provides the details:

Table 6-33 OStrap vs. build options

File Name	Description
`OS_CHECK_DESC != 0`	Checks the validity of descriptors used in the RTOS Detects the use of blocking services in interrupts
`OS_STACK_CHECK != 0`	Detects task's stack overflow Detects the use of blocking services in interrupts
`OS_OUT_OF_MEM != 0`	Detects out of memory conditions Detects the use of blocking services in interrupts
`OS_MTX_DEADLOCK < 0`	When a mutex deadlock is detected

Availability

Available since 2019

Arguments

`Error`	cause of the error trap.

Returns

`N/A`

Component type

Function

Options

Notes

The function `OStrap()` is coded in assembler to trigger a breakpoint whenever it's possible and / or put the processor in lower power mode. The error number is passed as the argument, as it is coded in assembly, the specific register holding the error number is port dependent.
When an error indicates an invalid descriptor, the problem is either due to a NULL pointer or the data structure indicated by the pointer does not have the proper marker(s) for the service involved.

An easy way to isolate where the un-recoverable error occurred is to use the "call stack trace-back" feature of the debugger.
A reminder: all blocking operation are semaphore based. A trapping error occurring during blocking will likely indicate something about a semaphore but it could be the semaphore used by a mailbox, a task's event, a memory block etc.

The following list describes all errors trapped as indicated by the argument `Error`:

`Error`	cause of the error trap.
`0 — 0x00`	Out of heap - `malloc()` called in `OSalloc ()` returned `NULL`. This error occurs when `OS_ALLOC_SIZE` is set to a negative value and `OSalloc()` is called: this configuration uses `malloc()`for dynamic memory allocation. Either reduce the memory allocated in the application or increase the size of the "C" heap. Calling directly `malloc()` cannot trigger this error.
`1 – 0x01`	Out of memory to allocate This error occurs when `OS_ALLOC_SIZE` is set to a positive value and `OSalloc()` is called> in this configuration the size of memory to allocated is set by `OS_ALLOC_SIZE`. Either reduce the memory allocated in the application or increase the value assigned to `OS_ALLOC_SIZE`.
`2 – 0x02`	ISR queue overflow This error occurs when the queue used to send the requests to the kernel by the interrupt handlers becomes full. The size of the ISR queue is set with the build option `OS_MAX_PEND_RQST`. The value assigned to it is either too small, or if it is large enough, then the application has one or more source of interrupts flooding the kernel with requests faster than the kernel can process them. In the later case it is most likely an indication the interrupt source is non-stop triggering interrupts alike if it's not informed the interrupt has been handled.
`4 – 0x03`	`OSalloc()` called in an interrupt This error occurs when `OSalloc()` is called in an interrupt context. `OSalloc()` is protected by a mutex therefore it can't be used in an interrupt.
`16 – 0x10`	Out of "name" memory This error occurs the build options `OS_STATIC_NAME` and `OS_NAMES` are both positive. The value assigned to `OS_STATIC_NAME` sets the amount of memory available for holding the character strings of all names. Either increase `OS_STATIC_NAME` or shorten the names of the services.
`32 – 0x20`	Too many tasks This error occurs when the build option `OS_STATIC_TASK` is positive and more tasks are created than the number assigned to `OS_STATIC_TASK`. Either reduce the number of tasks in the application or increase the value assigned to `OS_STATIC_TASK`.
`33 – 0x21`	Out of static stack memory This error is not a stack overflow i.e. a task using a larger stack than was allocated to it. This error occurs when the memory used by the stacks of all tasks is allocated from a memory pool sized according to the build option `OS_STATIC_STACK` (when set to a positive value). Increase the value assigned to `OS_STATIC_STACK`, or if possible, reduce the stack size allocated to the tasks when using `TSKcreate()`.

`34 – 0x22`	`EVTwait()` used in an interrupt This error occurs when `EVTwait()` is called in an interrupt context. `EVTwait()` is a blocking service therefore it cannot be used in an interrupt.
`35 – 0x23`	`TSKyield()` used in an interrupt This error occurs when `TSKyield()` is called in an interrupt context. `TSKyield()` applies to the task calling `TSKyield()`therefore it cannot be used in an interrupt.
`36 – 0x24`	`TSKcreate()` used in an interrupt This error occurs when `TSKcreate()` is called in an interrupt context. `TSKcreate()` is protected by a mutex therefore it cannot be called in an interrupt.
`37 – 0x25`	Invalid task descriptor used in `EVTabort()` This error occurs when `EVTabort()` is used with an invalid task descriptor.
`38 – 0x26`	Invalid task descriptor used in `TSKtout()` / `TSKtoutKill()` This error occurs when `TSKtout()`, or `TSKtoutKill()`, or internal timeout operations, is used with an invalid task descriptor.
`39 – 0x27`	Invalid task descriptor used in `TSKresume()` / `TSKsusp()` This error occurs when `TSKresume()` or `TSKsusp()`is used with an invalid task descriptor.
`40 – 0x28`	Invalid task descriptor used in `EVTset()` This error occurs when `EVTset()`is used with an invalid task descriptor.
`41 – 0x29`	Invalid task descriptor used in priority change operation This error occurs when `TSKsetPrio()` is used, or internally through the starvation or priority inversion protection, with an invalid task descriptor.
`42 – 0x2A`	Invalid task descriptor used when unblocking a task This is an internal error when the kernel has to unblock a task with an invalid descriptor.
`43 – 0x2B`	Invalid task descriptor used when blocking a task This is an internal error when internally the kernel has to unblock a task with an invalid descriptor.
`44 – 0x2C`	Invalid task descriptor in semaphore/ mutex blocked linked list All tasks blocked on a service are held in a linked list and this error indicates there is an invalid task descriptor is in the linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`45 – 0x2D`	Invalid task descriptor in mutex owner entry When a mutex is locked, an entry in the mutex descriptor holds the descriptor of the task locking the mutex (the mutex owner) and this error indicates an invalid task descriptor for the mutex owner. The most likely culprit is a memory corruption that would have occurred in the application.
`46 – 0x2E`	Invalid task descriptor in the timeout linked-list All tasks blocked with non-infinite expiry time are held in a linked list and this error indicates there is an invalid task descriptor is in the linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`47 – 0x2F`	Invalid task descriptor in ready to run linked list All tasks ready to run are held in linked lists, one linked list per priority, and this error indicates there is an invalid task descriptor is in the linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`48 – 0x30`	Too many semaphores / mutexes

		This error occurs when the build option `OS_STATIC_SEM` is positive and more semaphores and mutexes are created than the number assigned to `OS_STATIC_SEM`. Increase the value assigned to `OS_STATIC_SEM`, or if possible reduce the number of semaphores & mutexes required by the application.
`49 – 0x31`	Mutex deadlock detected	This error occurs when a mutex deadlock condition is detected (Section 9). The only way to eliminate this error is at the application architectural level because the problem is related on the way multiple tasks are accessing shared resources protected by mutexes.
`50 – 0x32`	`MTXlock()` used in an interrupt	This error occurs when `MTXlock()` is called in an interrupt context. `MTXlock()` is a blocking service therefore it cannot be used in an interrupt.
`51 – 0x33`	`SEMwaitBin()` used in an interrupt	This error occurs when `SEMwaitBin()` is called in an interrupt context. `SEMwaitBin()` is a blocking service therefore it cannot be used in an interrupt.
`52 – 0x34`	`SEMwait()` / `TSKselfSusp()` / `TSKsleep()` used in an interrupt	This error occurs when `SEMwait()` / `TSKselfSusp()` / `TSKsleep()` are called in an interrupt context. `SEMwait()` / `TSKselfSusp()` / `TSKsleep()` are blocking services therefore they cannot be used in an interrupt.
`53 – 0x35`	`SEMopen()` / `MTXopen()` used in an interrupt	This error occurs when `SEMopen()` / `MTXopen()` are called in an interrupt context. `SEMopen()` / `MTXopen()` are protected by a mutex so they can't be used in an interrupt.
`54 – 0x36`	Invalid semaphore descriptor used in `SEMabort()`	This error occurs when `SEMabort()` is used with an invalid semaphore descriptor.
`55 – 0x37`	Invalid mutex descriptor used in `MTXabort()`	This error occurs when `MTXabort()` is used with an invalid mutex descriptor.
`56 – 0x38`	Invalid mutex descriptor used in `MTXunlock()`	This error occurs when `MTXunlock()` is used with an invalid mutex descriptor.
`57 – 0x39`	Invalid task descriptor used in `SEMpost()` / `SEMpostAll()`	This error occurs when `SEMpost()` or `SEMpostAll()` is used with an invalid sempahore descriptor.
`58 – 0x3A`	Invalid mutex descriptor used in `MTXlock()`	This error occurs when `MTXlock()` is used with an invalid mutex descriptor.
`59 – 0x3B`	Invalid task descriptor used in `SEMwait()` / `SEMwaitBin()`	This error occurs when `SEMwait()` or `SEMwaitBin()` is used with an invalid semaphore descriptor.
`60 – 0x3C`	Invalid mutex descriptor used during priority inversion protection	This is an internal error that occurs when the kernel is performing priority inversion protection on a mutex with an invalid descriptor
`61 – 0x3D`	Invalid semaphore / mutex descriptor held in the task's blocker entry	When a task is blocked, an entry in its descriptor holds the descriptor of the service it is blocked on. The most likely culprit is a memory corruption that would have occurred in the application.
`62 – 0x3E`	Invalid mutex descriptor in task's mutex linked list	All mutexes locked by a task are held in a linked list and this error indicates there is an invalid mutex descriptor in the linked list.

The most likely culprit is a memory corruption that would have occurred in the application.

`63 – 0x3F` Invalid semaphore descriptor to block on

This is an internal error that reports when a task is getting blocked and the semaphore (all blocking services are semaphore based) to block on has an invalid descriptor. This error occurs during internal kernel operations. The most likely culprit is a memory corruption that would have occurred in the application.

`64 – 0x40` Too many mailboxes

This error occurs when the build option `OS_STATIC_MBX` is positive and more mailboxes are created than the number assigned to `OS_STATIC_MBX`. Increase the value assigned to `OS_STATIC_MBX`, or if possible reduce the number of mailboxes required in the application. This error is not about the memory used by the internal buffers of the mailboxes.

`65 – 0x41` Out of static mailboxes buffer memory

This error occurs when the build option `OS_STATIC_BUF_MBX` is positive and the total size of all mailboxes created exceeds the number assigned to `OS_STATIC_BUF_MBX`. Increase the value assigned to `OS_STATIC_BUF_MBX`, or if possible reduce the sizes of the mailboxes in the application. This error is not about the number of mailboxes themselves.

`66 – 0x42` `MBXopen()` used in an interrupt

This error occurs when `MBXopen()` is called in an interrupt context. `MBXopen()` is protected by a mutex therefore it can't be used in an interrupt.

`67 – 0x43` Invalid mailbox descriptor used in `MBXget()`

This error occurs when `MBXget()` is used with an invalid mailbox descriptor.

`68 – 0x44` Invalid mailbox descriptor used in `MBXput()`

This error occurs when `MBXput()` is used with an invalid mailbox descriptor.

`69 – 0x46` Invalid mailbox descriptor used in `MBXabort()`

This error occurs when `MBXabort()` is used with an invalid mailbox descriptor.

`70 – 0x46` Invalid group descriptor used in `MBXput()`

When a mailbox is attached to a group, the mailbox descriptor memorizes the group it is attached to. This error occurs when `MBXput()` is used with a valid mailbox descriptor but the group descriptor is invalid. The most likely culprit is a memory corruption that would have occurred in the application.

`71 – 0x46` Invalid group descriptor used in `MBXget()`

When a mailbox is attached to a group, the mailbox descriptor memorizes the group it is attached to. This error occurs when `MBXget()` is used with a valid mailbox descriptor but the group descriptor is invalid. The most likely culprit is a memory corruption that would have occurred in the application.

`80 – 0x50` Too many timer services

This error occurs when the build option `OS_STATIC_TIM_SRV` is positive and more timer services are created than the number assigned to `OS_STATIC_TIM_SRV`. Either increase the value assigned to `OS_STATIC_TIM_SRV`, or if possible reduce the number of timer services required by the application.

`81 — 0x51` Kernel request in a Timer Service callback function

This error occurs when a callback function attached to a Timer Service performs a kernel request. When the callback function is executed it is done from within the kernel. Because the kernel is not reentrant, Timer Services call back functions are not authorized to perform kernel requests. Remove the kernel request and if a kernel request is needed upon timer expiry create a task blocked on a semaphore or a mailbox and use a semaphore posting or mailbox deposit as the callback mechanism.

`82 – 0x52` `TIMopen()` used in an interrupt
This error occurs when `TIMopen()` is called in an interrupt context. `TIMopen()` is protected by a mutex so it can't be used in an interrupt.

`83 – 0x53` Invalid timer descriptor used in timer service operation
This error occurs any of timer service operation is called with an invalid timer service descriptor.

`84 – 0x54` Invalid descriptor in the timer service linked list
All active timer services are kept in a linked list and this error indicates there is an invalid timer service descriptor is in the linked list. The most likely culprit is a memory corruption that would have occurred in the application.

`96 – 0x60` Too many memory blocks
This error occurs when the build option `OS_STATIC_MBLK` is positive and more memory blocks are created than the number assigned to `OS_STATIC_MBLK`. Either increase the value assigned to `OS_STATIC_MBLK`, or if possible reduce the number of memory blocks required by the application. This error is not about the memory held in the memory blocks.

`97 – 0x61` Out of memory block memory
This error occurs when the build option `OS_STATIC_BUF_MBLK` is positive and the total size of all the memory required by the all memory blocks created exceeds the number assigned to `OS_STATIC_BUF_MBLK`. Either increase the value assigned to `OS_STATIC_BUF_MBLK`, or if possible reduce the sizes of the memory reserved required by the memory. This error is not about the number of memory blocks themselves.

`98 – 0x62` `MBLKopen()` used in an interrupt
This error occurs when `MBLKopen()` is called in an interrupt context. `MBLKopen()` is protected by a mutex so it can't be used in an interrupt.

`99 – 0x63` Invalid memory block descriptor used in `MBLKalloc()`
This error occurs when `MBLKalloc()` is used with an invalid memory block descriptor.

`100 – 0x64` Invalid memory block descriptor used in `MBLKfree()`
This error occurs when `MBLKfree()` is used with an invalid memory block descriptor.

`101 – 0x65` Invalid task descriptor blocked on a memory block
When a task gets blocked trying to obtain a memory block, that task descriptor is held in the memory block descriptor. This error indicates the task blocked is identified with an invalid task descriptor. The most likely culprit is a memory corruption that would have occurred in the application.

`112 – 0x70` Too many groups
This error occurs when the build option `OS_GROUP` is positive and more groups are created than the number assigned to `OS_GROUP`. Either increase the value assigned to `OS_GROUP`, or if possible reduce the number of groups in the application.

`113 – 0x71` `GRPadd()` used in an interrupt

	This error occurs when `GRPadd()` is called in an interrupt context. `GRPadd()` is protected by a mutex so it can't be used in an interrupt.
`114 – 0x72`	`GRPrm()` used in an interrupt This error occurs when `GRPrm()` is called in an interrupt context. `GRPadd()` is protected by a mutex so it can't be used in an interrupt.
`115 – 0x73`	`GRPwait()` used in an interrupt This error occurs when `GRPwait()` is called in an interrupt context. `GRPwait()` is a blocking service therefore it cannot be called in an interrupt.
`116 – 0x74`	Invalid service descriptor used in group-add This error occurs when `GTPaddMbx()` / `GRPaddSEM()` / `GRPaddSEMbin()` / is used with an invalid service (semaphore or mailbox).
`117 – 0x75`	Invalid group owner descriptor in a semaphore / mailbox When a semaphore or a mailbox is attached to a group, the semaphore / mailbox descriptor memorizes the group it is attached to. This internal error occurs when the kernel is internally processing groups. The most likely culprit is a memory corruption that would have occurred in the application.
`118 – 0x76`	Invalid task owner descriptor in a group When a group is created, the descriptor of the task that has created the group is held in the group descriptor. This internal error occurs when the kernel is processing groups. The most likely culprit is a memory corruption that would have occurred in the application.
`119 – 0x77`	Invalid group descriptor used in group-add This error occurs when when `GRPwait()` / `GRPaddSEM()` / `GRPaddSEMbin()` is used with an invalid group descriptor.
`120 – 0x78`	Invalid group descriptor used in `GRPrm()` This error occurs when `GRPrm()` is used with an invalid group descriptor.
`121 – 0x79`	Invalid group descriptor used in `GRPwait()` This error occurs when `GRPwait()` / `GRPaddSEM()` / `GRPaddSEMbin()` / is used with an invalid group descriptor.
`122 – 0x7A`	Invalid mailbox descriptor attached to a group When a mailbox is attached to a group, the group memorizes the mailbox that was attached. This internal error occurs when the kernel is processing groups and it detects an attached invalid mailbox descriptor. The most likely culprit is a memory corruption that would have occurred in the application.
`123 – 0x7B`	Invalid semaphore descriptor attached to a group When a semaphore is attached to a group, the group memorizes the semaphore that was attached. This internal error occurs when the kernel is processing groups and it detects an attached invalid semaphore descriptor. The most likely culprit is a memory corruption that would have occurred in the application.
`124 – 0x7C`	Invalid group descriptor in the group trigger list When services are attached to a group, each service is represented by a group descriptor and these are attached together in a linked list. This error indicates there is an invalid group descriptor in that linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`125 – 0x7D`	Invalid group descriptor in the group parking lot

	All groups "deleted" with `GRPrm()` are kept in a parking lot linked list for later re-use with `GRPadd()`. This error indicates there is an invalid group descriptor in that linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`126 – 0x7D`	Task blocked on an invalid group descriptor When a task is blocked on a group, the task the group memorizes the group it is blocked on. This internal error occurs when the kernel detects an attached invalid group descriptor. The most likely culprit is a memory corruption that would have occurred in the application.
`128 – 0x80`	Invalid task descriptor to perform starvation protection This internal error occurs when the kernel is expected to perform starvation protection on an invalid task descriptor. The most likely culprit is a memory corruption that would have occurred in the application.
`129 – 0x81`	Invalid task descriptor to in the starvation protection linked list All ready to run tasks under starvation protection are held in a linked list and this error indicates there is an invalid task descriptor is in that linked list. The most likely culprit is a memory corruption that would have occurred in the application.
`240+ — 0xE0+`	Core #N stack overflow (Single core: N always equates 0) This error occurs when a stack overflow is detected. The error number indicates the core number, i.e. the core number is `error number - 240 (or 0xE0)`. The task suffering from the stack overflow can be determined by looking at the global variable `G_OStaskNow`, which holds the task descriptor. Either increase the stack size when creating the task with `TSKcreate()` or reduce the stack use by the task.

See also

7 Appendix A: Priority inversion

The definition of priority inversion, as stated in Wikipedia [R2]:

> "*In computer science, priority inversion is a problematic scenario in scheduling when a higher priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.*
>
> *This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.*"

There are two well-known safeguard techniques that can be used against priority inversion. The first one is called *priority inheritance*, where the low-priority task locking a mutex inherits the priority of the higher-priority task that becomes blocked when it tries to lock the mutex. As defined in Wikipedia [R3]:

> *"In real-time computing**, priority inheritance** is a method for eliminating priority inversion problems. Using the programming method, a process scheduling algorithm will increase the priority of a process to the maximum priority of any process waiting for any resource on which the process has a resource lock"*

The second is called *priority ceiling*, where the shared resource is assigned a priority. Any task that accesses that share resource gets its priority set to the shared resource priority. As defined in Wikipedia [R4]:

> *"In real-time computing, the **priority ceiling protocol** is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections. In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task, which may lock the resource."*

The Abassi RTOS support both priority inversion protection methods. When the build option `OS_MTX_INVERSION` is positive, the priority inheritance mechanism is activated; when the build option is negative, the priority ceiling mechanism is activated.

7.1 Priority Inheritance

Priority inheritance is a mechanism where a task locking/owning a mutex will see its priority increased when a higher priority task tries to lock said mutex. The task owning the mutex gets its priority increased to the priority level of the task trying to lock the mutex. If another task tries to lock the same mutex, the mutex owner task gets its priority increased again if the task trying to lock the mutex is of higher priority than the new priority of the mutex owner. When the mutex owner unlocks the mutex, the priority of the task that has unlocked the mutex is brought back to its original priority. If that task locks another mutex, and the priority inheritance rules also apply to that mutex, then the task will have its priority level lowered to the priority level of the highest priority task trying to lock that new mutex. In effect, priority inheritance removes any possibility of a high priority task becoming the slave of a lower priority task.

Priority inheritance operates a bit differently if the RTOS is build for a single task per priority or for multiple tasks per priority. The latter case operates exactly as described in the previous paragraph. For the former, because two tasks cannot be at the same priority level, a task that has it priority level raised because of priority inheritance gets a priority level of one level higher than the priority of the highest priority task blocked on the mutex.

7.1.1 Single Task per priority

As explained in the previous section, when the RTOS is built for single task per priority, the priority inheritance mechanism assigns a priority level one level higher than normally required. This is done to not have two or more tasks at the same priority levels. This is true only if the application does not have a task assigned to that priority level + 1. So, as a simple rule, when priority inheritance is enabled in a RTOS built for single task per priority, the N priority levels above any task that can lock a mutex should remain unassigned, where the value for N is the maximum number of mutex a single task can have a lock on at the same time. N unused priority must be kept above all tasks that can lock a mutex.

To better understand this requirement, consider the following example:

- Task A (high priority) use Mutex X
- Task B (mid priority) uses Mutex X and Mutex Y
- Task C (low priority) uses Mutex Y

If the following sequence of events happen:

1) Task B locks Mutex X
2) Task A tries to lock Mutex X, priority of Task B raised above current Task A
3) Task C locks mutex Y
4) Task B tries to lock mutex Y, priority of Task C is raised above current Task B

The final priority order of the tasks will be:

Task B: original priority level of Task A + 1

Task C: current priority level of Task B + 1 (== original priority level of Task A + 2)

Task A: same priority

7.2 Priority Ceiling

The Abassi RTOS completely automates the priority ceiling mechanism, which means there is no need for the designer to set the priority at which a task locking a mutex must operate at. The RTOS memorizes the priority of the highest priority task locking the mutex. Also, the implementation of priority ceiling does not match exactly what the standard definition implies. When a task locks a mutex and no other tasks are blocked on that mutex, the priority of the locker is not modified; neither is the ceiling priority updated.

As explained in the previous section on priority inheritance, the priority ceiling mechanism operates a bit differently depending if the build allows multiple tasks running at the same priority. When multiple tasks are allowed to run at the same priority level, the priority ceiling raises the priority of the task that locks the mutex to exactly the same priority as the highest-priority task that ever used the mutex. But when the build option does not allow multiple tasks to run at the same priority, the priority is raised one level above the maximum, without exceed the numerical value zero, which is the highest priority level allowed in an application.

For priority ceiling, it is not possible to give a general rule that will allow priority ceiling to properly operate when a task can lock two or more mutexes at the same time. This is because the priority ceiling mechanism attaches a priority to a mutex and that priority is always one level above the highest priority task that can lock the mutex. So if the highest priority task can lock two or more mutexes, each one of these mutex may have the same priority attached to them; multiple mutexes means possibly multiple tasks locking one of the mutexes. The only solution when multiple mutexes can have the same ceiling priority attached to them is to modify these ceiling priority to be different through the use of the `MTXsetCeilPrio()` component (Section 6.5.16). Obviously there cannot be any task set to run at any of the different ceiling priority.

8 Appendix B: Task Starvation

The condition known as task starvation occurs when higher priority tasks consume all the CPU available, forcing the lower priority tasks to remain in the ready to run state for an extended period of time. On most applications, the duration of time when the task starvation condition occurs is short enough to not impact the overall behavior of the application. But, when an application is constantly operating very close to the maximum CPU all the time, task starvation may start to have a negative impact. If some lower priority tasks must run in order to maintain the proper behavior of the application, then there is obviously a need for a mechanism to guarantee some CPU for all tasks. The most common technique used to fulfill this requirement is called "priority aging".

Wikipedia defines Priority aging as [R5]:

> "***Aging*** *is the process of gradually increasing the priority of a task, based on its waiting time. The aging technique estimates the time a process will run based on a weighted average of previous estimates and measured values. Aging can be used to reduce starvation of low priority tasks. Aging is used to ensure that jobs in the lower level queues will eventually complete their execution*."

In a hard real-time environment, it may not be desirable to implement priority aging as is. For example, if a low priority task needs a fair amount of CPU, by increasing its priority, it may reach a priority level such that the CPU time consumed by it creates another task starvation, this one for a critical, high priority task.

A modified priority aging mechanism is available in Abassi when the build option `OS_STARVE_WAIT_MAX` (Section 4.1.43) is non-zero. The basic priority aging mechanism is retained where a ready to run task gets its priority increased one level at a time if it does not reach the running state. The value of build option `OS_STARVE_WAIT_MAX` specifies, in number of timer ticks, the maximum time a task must remain in the ready to run state before its priority is increased by 1 level. The priority level is increased one level at a time until it reaches the priority value set by the build option `OS_STARVE_PRIO` (Section 4.1.41), and remains at that priority until it reaches the running state. When the aged priority task becomes running, it is allowed to be running for a maximum duration; this duration is specified in number of timer ticks with the build option `OS_STARVE_TIME_MAX` (Section 4.1.42). Once it has run, the task priority returns to its original priority.

There is only one task at a time that can get its priority level increased with the priority aging process. This choice was made to eliminate the risk that promoting multiple tasks could result in many tasks reaching the running state at the same time. If that was to happen, the combined CPU usage could aggravate the problem of task starvation, affecting a higher priority task. So only a single task gets its priority increased step by step. After the task becomes running for a maximum of `OS_STARVE_TIME_MAX`, it returns at the end of an internal queue. That internal queue holds all the tasks in the application that are in the ready to run state with a priority value less than `OS_STARVE_PRIO`. The task that has been held in the ready to run state the longest is always the task on which priority aging is performed. If a task in the queue becomes running, it is removed from the queue, and when it reaches the ready to run state, it is inserted at the end of the queue.

As a simple design rule, the build option `OS_STARVE_PRIO` should always be set at the priority level where the tasks at higher priority than `OS_STARVE_PRIO` have a hard real-time requirement. The value of `OS_STARVE_PRIO` can also be increased (lower the priority) to a value where only the tasks below that priority level are known to risk suffering from starvation.

Task starvation is disabled on a task if the starvation priority is equal to `OS_PRIO_MIN`, or if it is greater or equal to the run priority of the task. The former was added as an invalid case because the Idle Task is always set at a priority value of `OS_PRIO_MIN` and should never be forced to run, as it may be where the processor is put into sleep mode. Having the starvation mechanism applied to the Idle Task would then put the processor into sleep once in a while when it is not appropriate.

9 Appendix C: Mutex Deadlock

The most simple mutex deadlock problem is when two tasks lock one mutex each and are mutually blocked on the mutex the other task locks. This can be extrapolated to the generic mutex deadlock problem when more than two mutexes and two tasks are involved. Here as a generic description of the mutex deadlock problem:

1) Task N locks Mutex N
2) For Task K in N+1 to M

   Task K locks Mutex K / becomes blocked trying to lock Mutex K-1
3) Task N blocked trying to lock Mutex M

The deadlock occurs at step 3) as the first task (Task N) tries to lock mutex M. In a mutex deadlock condition, there is a chain of locked mutex – task pairs that goes back to a mutex the task locks.

Abassi has a provision to detect any type of mutex deadlock. This feature is enabled when the build option `OS_MTX_DEADLOCK` (Section 4.1.25) is set to a non-zero value. As there are no very simple mathematical techniques to quickly detect a mutex deadlock (other than using matrices), the operation performed in the RTOS is to traverse the chains of locked mutex – task pairs.

The Abassi deadlock detection algorithm is show in its simplified form below:

Table 9-1 Mutex deadlock detection pseudo-code

```
if the task get blocked on a mutex
    if the task locks one or more mutexes
        Owner ← Owner of the mutex the task will block on
        while Owner valid
            Locker ← Task that Owner is blocked on
            if Locker is valid
                Mtx = Mutex locked by Locker
                while Mtx valid
                   if Mtx == Locker
                       DEADLOCK DETECTED
                   Mtx ← Next Mutex locked by Locker
                endwhile
                Owner ← Task locking mutex locker is blocked on
             endif
        endwhile
    endif
endif
```

10 Appendix D: Round Robin

Round robin is quite simple: it gives equal access to the CPU for all running / ready to run tasks at the same priority. Abassi offers a unique implementation of round robin:

- The CPU can be distributed unevenly amongst tasks at the same priority
- Round robin tasks can co-exist with tasks that will run until blocking/completion

The usefulness of uneven round robin CPU distribution amongst tasks at the same priority is to allocate the available CPU according to the needs of the tasks. Many applications have tasks that can run at low priority, but the relative complexities of these tasks are quite different. One task may be used for a few simple operations, while another needs a fair amount of CPU to complete its processing. Trying to build an application using different priority levels, to give fair access to the CPU according to their needs, is quite complex. However, using the component `TSKsetRR()` (Section 6.3.21) reduces the headache of the CPU distribution to a simple arithmetic problem.

For example, assume three tasks need to share the available CPU, and the complexity of the tasks is such that the first task needs 1/2 the CPU of third task to complete its processing, and the second needs 1/4 the CPU the third task to complete its CPU. This means the first task needs access to 2/7 of the CPU, the second needs access to 1/7 of the CPU, and the third for 4/7 of the CPU.

If the design is such that a full round robin cycle between these three task is desired within 40 timer ticks, then the arguments for the components `TSKsetRR()` are:

- 1st Task: (2/7) * 40 = 11 timer ticks
- 2nd Task: (1/7) * 40 = 6 timer ticks
- 3rd Task: (4/7) * 40 = 23 timer ticks

Having tasks that are allowed to run until blocking/completion co-existing with tasks at the same priority that a share the CPU in a round robin fashion is an extension of the uneven CPU distribution amongst these tasks.

11 Appendix E: Cooperative mode

The Abassi kernel offers two types of cooperative operation. The first one is a full emulation of a cooperative RTOS, and the second allows the use of a cooperative mode amongst tasks at the same priority.

11.1 Cooperative RTOS emulation

The Abassi kernel can be configured to fully emulate a cooperative OS by setting the build option `OS_COOPERATIVE` (Section 4.1.4) to a non-zero value. It is important to understand that this is an emulation, and not a real cooperative task dispatcher. As a drawback, the cooperative emulation cannot use a single stack, therefore every task in an application still requires their own individual stack. But the emulation removes the typical constrains a single stack cooperative RTOS has, namely:

- ➢ The CPU can be relinquished (or blocked due to the use of a synchronization component) at any call level: it is not necessary to relinquish the CPU/block only at the primary level (task level);
- ➢ Automatic variables are preserved upon relinquishing the CPU, therefore it is not necessary to declare the persistent variables as static;
- ➢ All preemption synchronization mechanisms are available: typically a single stack cooperative RTOS only offers events and mutexes with priority ceiling.

When the RTOS is configured to operate in cooperative mode, the native preemption mechanism of Abassi is disabled (truly, it is removed from the kernel code). Therefore, task switching can only happen when the running task relinquishes the CPU through the `TSKyield()` component (Section 6.3.32), or when it blocks on a semaphore, mutex, event, or mailbox. The standard interrupt dispatcher is still required because the dispatcher is the element that allows Abassi to not disable interrupts as the technique to protect the RTOS critical regions.

11.2 Same priority cooperative

Instead of using the native time sliced round robin of Abassi, it is possible to make tasks at the same priority decide when they relinquish the CPU to the others tasks at the same priority. First, for this type of behavior to exist, it is necessary to set he build option `OS_ROUND_ROBIN` (Section 4.1.36) to a negative value, to activate the programmable duration for the time slices in round robin. If one remembers, setting the round robin time slice duration, through the component `TSKsetRR()` (Section 6.3.21), to a zero value allows the task to keep using the CPU as long as it does not become blocked or preempted. When preempted, upon release of the preemption, the task still uses the CPU. When blocked, the task will relinquish the CPU to the next task at the same priority.

So, all there is to do to have multiples task at the same priority share the CPU in a cooperative manner is to set the round robin time slice duration of the task to a value of 0, though the component `TSKsetRR()`. Then, with the component `TSKyield()`, the sharing of the CPU can be controlled.

It is not necessary to set to zero the round robin time slice duration of all tasks at the same priority. If there is a mix of time slice durations of zero and non-zero, the two types co-exist without issue, as the tasks must either relinquish the CPU or Abassi's native round robin mechanism forces the task to relinquish the CPU upon reaching its maximum duration.

12 Appendix F: Protecting "C" libraries for multithreading

There are two issues to take into account when discussing protection of the "C" runtime library for multithreading. One aspect about multithreading protection involves non-reentrant functions, and the other aspect is related to the internal global variables used inside the library. Examples of non-reentrant functions are dynamic memory allocation functions such as `malloc()` and `free()`. These functions manipulate at least one internal linked list, and cannot be re-entered, otherwise the linked list could become corrupted. Examples for the internal global variables are the `errno` or `locale` services. Ideally, in a multithreading environment, each task would have access to its own copy of each one of the internal variables used by the library.

Some development tool suites (sometimes only for selected processors) have run-time libraries that can have hooks that allow them be protected for re-entrance. When a tool suite support only re-entrance protection, not the private set of variables, Abassi comes either pre-configured to use the protection mechanism, or the port document explains how to activate the protection. When re-entrance protection is not available, it becomes necessary to protect against re-entrance by using either a mutex or enabling/disabling interrupts. For all cases, all there is to do is as shown in the following tables, using `malloc()` as an example:

Table 12-1 Multithread protection with a mutex

```
MTXlock(G_OSmutex, -1)
Ptr = malloc(SIZE_OF_ARRAY);
MTXunlock(G_OSmutex)

…

MTXlock(G_OSmutex, -1)
free(Ptr);
MTXunlock(G_OSmutex)
```

Table 12-2 Multithread protection through interrupt disabling

```
ISRstate = OSintOff();
Ptr = malloc(SIZE_OF_ARRAY);
OSintBack(ISRstate);

…

ISRstate = OSintOff();
free(Ptr);
OSintBack(ISRstate);
```

The example with the mutex uses `G_OSmutex`, which is the mutex used internally by Abassi. It is highly recommended to use `G_OSmutex` to protect the library against re-entrance as this mutex is always available, and it is already used to perform multithreading protection. Using `G_OSmutex` eliminates the risk of encountering mutex deadlock issues.

Non-reentrant functions from the run-time library, when such functions are protected by a mutex, must never be used in an interrupt. The reason is quite simple: assuming a mutex is already locked because the function is already in use by a task, when trying to lock it again in an interrupt, it will block the task that was interrupted. It will not block the interrupt operation. Even if it was possible to block the interrupt operation, the application would lock-up.

Full multithreading support, combining protection against re-entrance with a private set of the library global internal variables for each task, is rare. Forcing the tasks in an application to have access to their own private set of variables is not often needed by an application. For that reason, the activation of the private set of variable is user selectable. Typically, the options offered for multi-threading protection are either no protection, full protection, or re-entrance protection with selected tasks with their own private set of the library internal variables. The port document always describes in detail all the options offered by Abassi.

If the run-time library does not support hooks to deliver private sets of its internal variables to each task, it is still possible to protect the library, but the code modifications becomes very cumbersome, as it is necessary to swap through copying the internal variables at every task switch. If the target application needs private sets of the internal variables for a library that does not support the feature, contact Code Time Technologies to learn how to perform the modifications.

13 Appendix G: Hashing

When runtime creation and names are supported (build options `OS_RUNTIME` and `OS_NAMES`), Abassi keeps track of every services and tasks that have been created. This is done mainly to support the run-time safe service creation, which removes the requirement to create a service before it is accessible by any of the application tasks. The run-time safe service creation done by automatically creating services when the first opening of the service is done. Abassi keeps track of the existing services through linked lists, one link list per services / tasks. When a service is opened, Abassi scans the relevant link list to see if the service to open already exists or not. If it does not exist, it creates it and adds it to the linked list. If the service has already been created, it reports the existing service for the opening operation. Traversing the linked list for the check is fairly lightweight as long as there isn't too many services already created.

If the number of services of the same type created in an application is quite large, alike 100's, then using hashing will help reduce the time required to open a service. To reduce the serach time, all there is to do is to define and set to a value greater than 1 the build option `OS_HASH_XXX` associated to the service one wants to reduce the search time. Statistically, the search time becomes `1/OS_HASH_xxx` the time required using a single linked list.

When hashing is enable, the "C" string of the name of the service is used to compute the hash code, which is simply done by using the sum of each ones of the ASCII characters in the string, modulo the hashing table size, which is sized to `OS_HASH_XXX` entries. This hashing code is then used to select the hashing table entry corresponding to one of the `OS_HASH_XXX` linked lists holding the already created services. The way the hashing is computed is definitely not optimal; it has been kept very simple to minimize the code size and the real-time requirements.

14 References

[R1] C Standard 1999, available at: http://www.open-std.org/JTC1/SC22/WG14/
[R2] http://en.wikipedia.org/wiki/Priority_inversion, definition of Priority Inversion.
[R3] Priority Inheritance on Wikipedia, http://en.wikipedia.org/wiki/Priority_inheritance
[R4] Priority Ceiling on Wikipedia, http://en.wikipedia.org/wiki/Priority_ceiling_protocol
[R5] http://en.wikipedia.org/wiki/Aging_(scheduling), definition of Priority Aging.
[R6] Abassi System Calls Layer, available on http://www.code-time.com