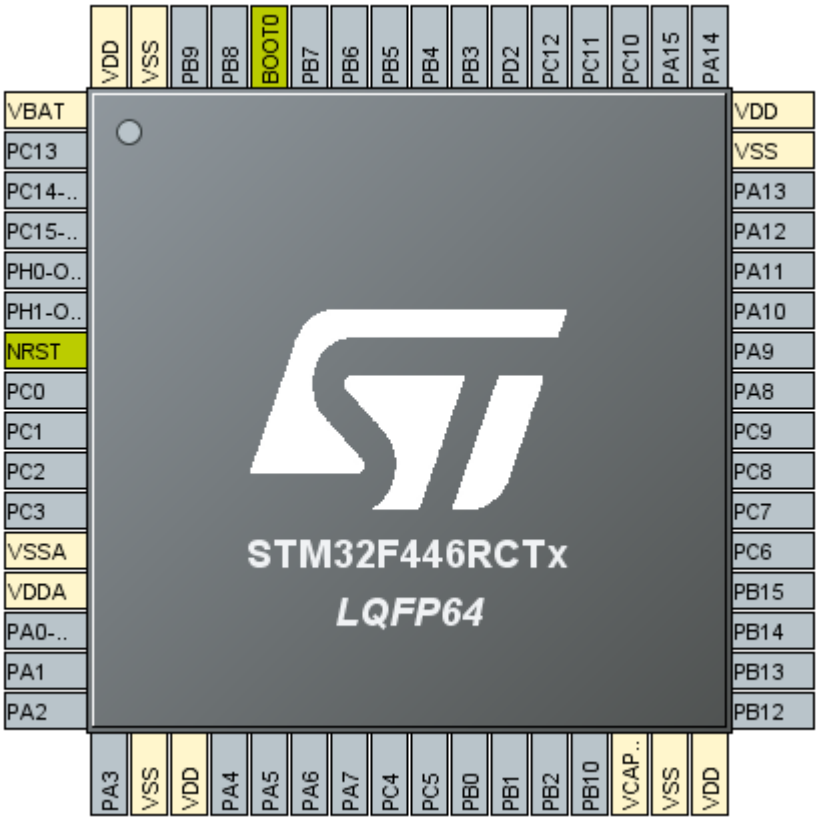
AdAstra RTK

Real Time Kernel

STM32 edition 2022-03



Visit us at **AdAstra-Soft.com**

Limitation of liability

Legal notice

Table of content

1 User's Guide

1.1 Introduction to RTOS

Why use a real-time kernel, or RTOS?

The next few lines present the arguments most often proposed on the question.

A traditional application without RTOS on a microcontroller usually uses two principles:

- The use of interrupt event management
- A "super loop" that periodically and consecutively calls the processes to be performed, which must be written in the form of state machines to allow a sort of cooperative "round robin" multitasking.

These applications, when they become more complex over time, and the maintainers follow one another reveal their drawback:

- The timings become difficult to master in the super loop.
- The interactions between the different parts of the application become inextricable.
- To solve the above difficulties, we develop exclusion and time management functionalities. These functionalities are generally non-generic and need to be increased with each new use.
- The realization of these functionalities uses notions of CPU architecture and is time consuming.

Finally we note that many parts of the functionalities of a RTOS have been developed: we reinvent the wheel. But without having the advantages that one is entitled to expect from a real-time kernel.

Here's what a real-time kernel brings:

- The efficient management of multiple tasks makes it possible to break down the application into simpler tasks to develop, with an automatic scheduling. You can write your tasks and let the kernel make them work together.
- An RTOS provides ready-made and proven services: management of interrupts, timers, communication between tasks (semaphore, mutex, message queue, mailbox)... Examples allow to implement them in your application. Often the kernel-specific drivers for the most used or complex devices are available.
- An RTOS provides sophisticated diagnostic tools, which further reduce development time: debugging, multilevel configurable traces, stack monitoring. Diagnostics is often the most valuable service for the developer.
- The RTOS provides a BSP that dispenses you from writing low-level layers and having to master the architecture of increasingly complex processors. If a specific BSP does not exist, examples are provided and can be adapted. Your

application is portable (as far as the kernel is concerned) on all the processors supported by the kernel.

In conclusion, adding an RTOS to your project means adding one or more highly skilled engineers for a fraction of the cost. A good RTOS has already solved many hidden problems that take a long time to resolve. In addition, the supplier can provide support. Using proven solutions reduces risk and gives your project the best chance of success.

It must be admitted, however, that the super loop is suitable for a certain number of projects, especially if the MCU has implemented a little memory (a few K bytes) or if the project is very simple and will not evolve. In any case we must think carefully to adopt the best solution.

1.2 Kernel overview

AdAstra RTK is a real-time kernel particularly well suited to mid-range 32-bit processors: from a few tens of KB to about 1 MB of RAM. In general, applications using processors with fewer resources do not need to implement a real-time kernel, and applications requiring significant resources also require more advanced functions (partitioning, virtual memory ...).

AdAstra RTK has been developed with specific goals:

- Adapted natively to 32-bit ARM Cortex-M processors. This allows having a well-structured code and not polluted by innumerable sections of conditional compilation.
- Architecture concepts are deliberately simple to make it understandable to all people: users: students, maintainers ...
- The coding standard adopted is intended to make the code maintainable, strengthening its readability and the obligation to comment judiciously. The adoption of recognized rules (MISRA) goes in the same direction.

The main features of AdAstra RTK are:

- Strictly preemptive and deterministic: The highest priority task is guaranteed to run and not be interrupted until it gives up its right of execution, or a higher priority task becomes ready to run.
- Almost unlimited number of tasks (but depending on the available memory).
- Up to 256 priority levels. Priority management is optimized if the number of levels is less than or equal to the size of the native word (32 levels on a 32-bit system for example).
- Recursive mutex with priority inheritance, to avoid the phenomenon of unbounded priority reversal. This mutex has a complete implementation of the algorithm.
- Complete set of inter-task communication: counting semaphore, mutex, inter-task signals, message queues, message pool management.

- Timeout management for all potentially blocking APIs. A null or unlimited timeout is allowed.
- Fast software timer, without additional task.
- It is possible to set the kernel to optimize system ticks, and save energy (sometimes this is called tickless method, but this is not really tick less).
- Static allocation of all kernel object structures, configurable by the user. Stack allocation of tasks can also be static, and the user can choose to do without dynamic memory allocation altogether. This is done using a single API: there is no functions specific for static allocation and others for dynamic allocation.
- Adjustable to the needs: definitions allow to specify the quantity of each type of objects of the kernel (tasks, semaphores, queues, etc.).
- Dynamic allocation of memory by the TLSF algorithm which is fast, efficient and above all deterministic. The kernel only uses dynamic memory allocation on request from the application. This dynamic allocation can be inhibited during kernel configuration.
- Debugging capabilities: stack occupancy analysis, deferred log task, resource utilization statistics, and use of hardware specific characteristics (SWO). Heavy use of checks by ASSERT, which allows to delete these tests when generating the final code. Centralized management of fatal or not fatal errors.
- A clear and consistent API (orthogonality : https://en.wikipedia.org/wiki/Orthogonality_(programming)). Once familiar with the coding and naming conventions the user can guess the names of the functions and the parameters to be provided.
  For driver development there is no dedicated API for interrupt functions, which use the standard API. If there is a violation of the restrictions on use by the interrupt handler this is reported.
- The coding is in ANSI-C, with the least assembler possible: A typical port uses less than 50 assembler lines. This feature facilitates, in addition to reading and therefore maintainability, porting to other families of processors.

1.3 The Board Support Package

The AdAstra RTK kernel is strictly independent of the processors. To port the kernel on a processor family, you must create a specific Board Support Package (BSP) for this family. The BSP provides:

- A set of predefined functions to adapt during porting
- According to the needs of the user additional functions.

The BSP is often based on software supplied by the manufacturers: ARM CMSIS, STM32 Low Level Drivers, etc.

A standard BSP is generally adapted to an evaluation board and proposes:

- A console on UART (UART to select from all those of the processor),
- Access to LEDs, and other GPIO outputs,
- Access to buttons, and other GPIO inputs.

- A standard demonstration application using the elements above

The BSP offers other generic APIs such as:

- Configuration of the system tick, and functions to query or modify it
- Time measurement functions (time stamping)
- Interrupt management
- Debug configuration (SWO…).

1.4 Static or dynamic allocation?

The configuration of the kernel is static: the quantity of each object to be made available in the kernel (task, mutex, queue, timer ...) is specified during the configuration, and they are allocated during the compilation. So the size of the kernel is known from the compilation, and it is certain that these objects will be available at runtime.

A special case concerns task stacks. Several choices are available:

- Dynamic memory management is not implemented in the kernel. The user must then himself predict the memory spaces to be used as stack of tasks.
- The dynamic memory management is implemented, but it is possible not to use it on a case by case basis, which goes back to the previous case.
- Dynamic memory management is implemented and used automatically by the kernel, the user only communicates the size of each stack.

It is therefore the configuration of the kernel associated with the use made of it which makes it possible to have an entirely static or dynamic allocation.

1.5 The identifiers

Most objects provide an identifier at the time of their creation (a handle), which allows them to be referenced for subsequent uses. These identifiers should be considered as meaningless opaque references outside of the kernel APIs.

1.6 Standard library

The kernel does not implement a standard library such as "newlib" or "newlib-nano", for several reasons:

- The kernel does not require any of these libraries.
- Some of these libraries and the inclusion files accompanying them are prohibited by standards such as MISRA.
- The memory resources occupied by these libraries can be incompatible with some processors.

Some functions of these libraries can be replaced by those provided by the kernel, such as these:

- aaGetChar Gets a console character
- aaPutChar Send a character to the console
- aaPrintf Equivalent, with some restrictions, to the standard printf ()
- aaPrintfEx Printf on any device
- aaSnPrintf Equivalent to snprintf ().
- aaGets Equivalent to gets () but with the security of fgets()

However, the kernel uses some intrinsic functions of the compilers, which must be provided by the BSP. The list of these functions is given in the appendix.

The standard library "stdlib.h" provides the dynamic memory management functions of the *malloc()* family. These functions rely on the *sbrk()* system call, which manages the heap. But the heap is already managed by the kernel.

Therefore, to allow the use of functions of the *malloc()* familly, the _syscall.c file has functions rerouting the management of the dynamic memory of the standard library to the management provided by the kernel.

It should be noted that the use of *printf()* results in dynamic allocation of several blocks, and intensive use of the stack. Most of the time it is better to use *aaPrintf()* or *aaSnPrintf()*.

1.6.1 1.6.1 Note on using newlib.

Newlib has two main problems when used with a real-time kernel:

- The reentrancy of certain functions: newlib has functions which are not “thread-safe”.
- The freeing of the memory allocated by newlib when the task which caused this allocation is deleted.

The problem of re-entrance of functions is easily solved, because planned during the design of newlib: Just configure AA_WITH_NEWLIB_REENT in aacfg.h. After that a structure of type *_reent* is added to the control block of each task, and the kernel ensures the update of the variable *_impure_ptr* of newlib at each context switch.

Newlib has made no provision for freeing some memory blocks it has allocated. You must therefore be very careful when a task uses newlib functions that allocate memory and then this task is destroyed: there may be memory leaks!

This is probably because newlib is designed for process / thread based systems. The memory allocated by newlib is freed automatically only at the end of the process, by the system and not by newlib.

The model of an RT can be compared to a system with a single process, whose threads are the tasks. Since the process is never terminated, the memory is not freed.

If you really want to use all of newlib's features, you'll have to monitor memory allocations and manage freeing yourself.

1.7 MISRA compatible?

For short no.

Mainly because the applied development cycle is different in some respects from that described in the 2012 version of this standard.

But the state-of-the-art development rules have been applied, especially to the coding rules. The coding rules applied are intended to:

- Ensure code performance
- Ensure the ease of reading the code which ensures its reliability and maintainability.

For that, some rules have been knowingly relaxed:

- Several exit points per function are allowed (rule 15.5). This makes it possible to control the parameters of the functions by limiting the depth of indentation. A large depth of indentation is detrimental to the clarity of the code and is error prone in case of modification of the code.
- Several "break" (rule 15.4) by structure "for" or "while" are allowed. This ensures greater clarity of the code.
- The use of union (rule 19.2) is as limited as possible: only one place. This use is necessary for reasons of code performance. The use of this union is always coherent: the writing then the reading is always done on the same element.
- To respect the encapsulation of data and the opacity of the identifiers, the conversion of a pointer-to-void type to another type of pointer is allowed (rule 11.5).

1.8 Task management

1.8.1 Creating a task

A task is created with the *aaTaskCreate()* function. This function has among other parameters:

- The name of the task.
- The entry point of the task: the name of the function to be executed.
- An argument to be supplied to the function of the task.
- The size of the stack that must be allocated.
- A pointer to the stack. This allows a completely static allocation: the user takes care of the allocation of the stack at the time of the creation of the task, and the release of the stack during the destruction of the task. If the stack pointer is NULL and dynamic memory allocation is allowed, then the kernel takes care of dynamically managing the stack.

In return for the creation of the task a task identifier is provided to the user, it must be used with the task management API. A special identifier AA_SELFTASKID identifies the current task, the actual identifier of the current task can be obtained with *aaTaskSelfId()*.

The maximum number of tasks handled by the kernel is specified during kernel configuration (*aacfg.h* file).

1.8.2 Deleting a task

The destruction of a task can be done by several means:

- The task exits the function specified during creation by return, or by reaching the end of the function.
- By running *aaTaskDelete()* with AA_SELFTASKID.
- Another task runs *aaTaskDelete()* with the task identifier.

A task must be destroyed with great caution, because of the difficulty of recovering the resources it holds. This is true especially if the destroyed task holds mutexes or semaphores.

If a task uses a static stack allocated by the user, during the destruction of the task the *aaUserReleaseStack()* callback is called. This gives the user the opportunity to know that a memory block is free and manage it accordingly.

1.8.3 Suspending a task

A task can be suspended by *aaTaskSuspend()*. In this state, even if it is ready to execute, it does not.

We can get him back to work with *aaTaskResume()*.

When a task is suspended while it is in a waiting state (*aaTaskDelay()* for example), it does not immediately go into a suspended state. It does not go into the suspended state until the end of the waiting state: end of the delay, expiry of the timeout, obtaining the resource...

1.8.4 Task priority management

AdAstra can manage up to 256 priority levels, with 0 being the lowest priority and 255 being the highest priority.

AdAstra is strictly preemptive and deterministic: The highest priority ready task is guaranteed to run and not be suspended until it gives up its right of execution or a higher priority task becomes ready. This means that a task can prevent all low priority tasks from running if it does not go to sleep or hangs waiting for resources for example.

The only exception is interrupts that can, when not inhibited, temporarily suspend the highest priority active task.

If multiple tasks of the same priority are ready, the one at the head of the list of this priority runs. It will execute until it gives up its execution right: *aaTaskDelay()*, *aaTaskYield()* or wait for resource for example. At this point it is inserted at the end of the list, and the task at the head of the list runs. This mechanism allows for a cooperative "round-robin" management. This respects the strictly preemptive and deterministic management principle expressed above.

The number of priority levels to manage is configured by AA_PRIO_COUNT in aacfg.h. The scheduler is more efficient if the number of priorities is at most equal to the number of bits in a native word of the processor (32 bits on a 32-bit processor). The general algorithm applies beyond 2 words (64 priority levels on a 32-bit processor). Limiting the number of priorities saves memory because the kernel maintains a list by priority.

Priority 0 is reserved for the task "idle" which is executed when no other task is ready to execute, i.e. it is not possible to create another task of priority 0. The task "idle" is created automatically upon system initialization, and cannot be destroyed.

A task has a basic priority, which is assigned to it when it is created, or calling *aaTaskSetPriority()*. The basic priority can be obtained calling *aaTaskGetBasePriority()*. When using a mutex, the priority of a task may change because of the priority inheritance mechanism. The actual current priority of a task can be obtained with *aaTaskGetRealPriority()*.

1.8.5 The task states

The tasks are in one of the following states:

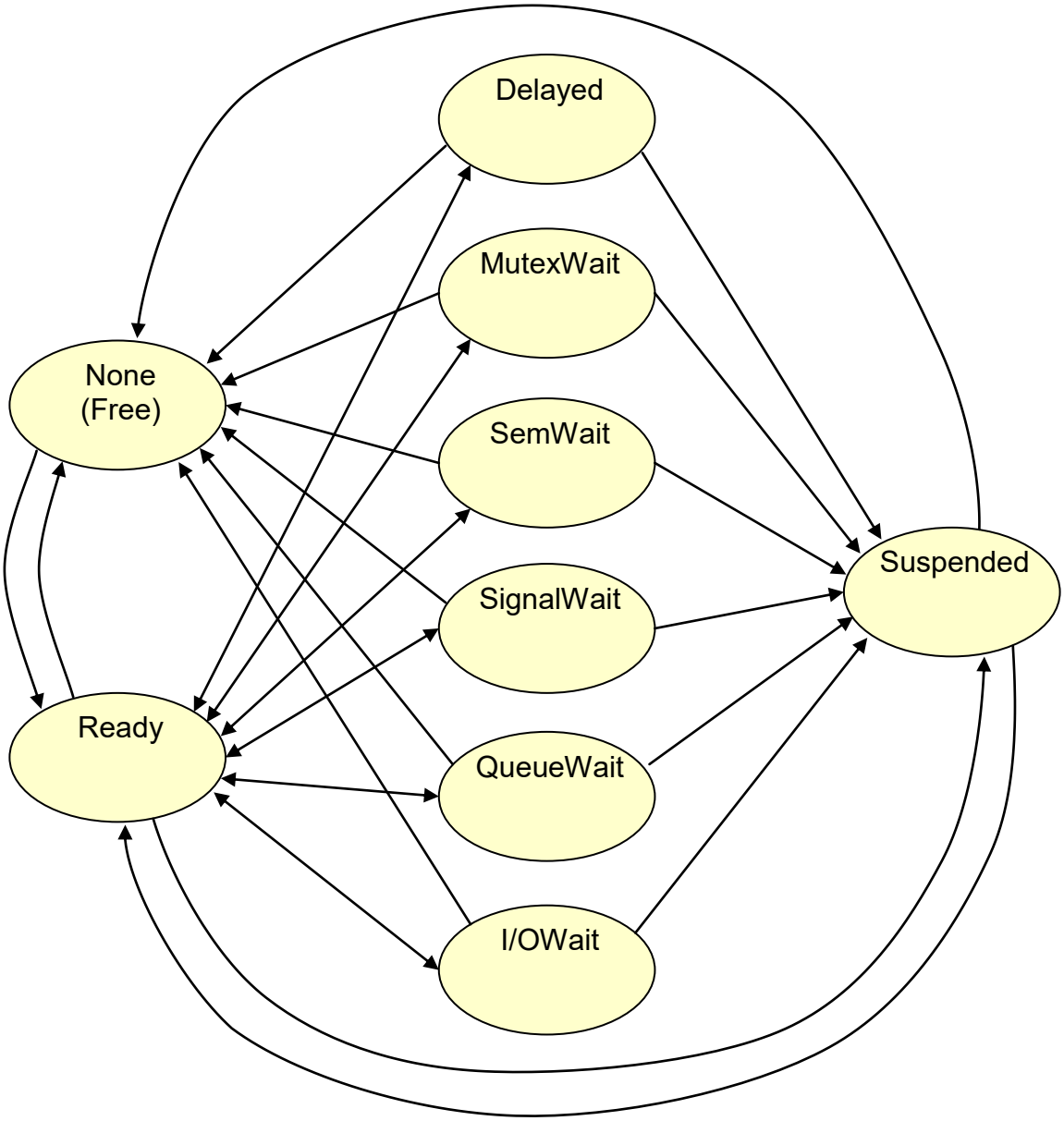
aaNoneState	These tasks are free, i.e. not created.
aaReadyState	These tasks are ready to run. Only one task runs: One of the tasks with the highest priority, and is in this state.
aaDelayedState	These tasks are waiting for a certain amount of time, which can be infinite.

aaMutexWaitingState	These tasks wait for a mutex owned by another task.
aaSemWaitingState	These tasks wait for the counter of a semaphore to become positive and obtain it.
aaSignalWaitingState	These tasks are waiting for a combination of their signals to be positioned by other tasks.
aaQueueWaitingState	These tasks are waiting to be able to write or read a message in a message queue.
aaIoWaitingState	These tasks wait for an event in a driver.
aaSuspendedState	These tasks are suspended: even if they meet the conditions to run, they remain inactive.

Suspending a task is a special mechanism:

If a task is in the aaReadyState state when it is paused, it immediately goes into the suspended state.

If a task is in a pending state when it is paused, it continues to wait, and as soon as the exit condition of the waiting state is fulfilled it enters the suspended state. Even if the suspension of the task is carried out in two stages, the task is effectively suspended as soon as the request for suspension is made.



This diagram shows all legal state transitions for a task.

1.9 Interrupts

This section is mainly intended for system or driver developer.

Interrupt handling is reserved for the kernel and drivers, which must include *aakernel.h*, which itself includes *bspcfg.h* which declares interrupt handling functions:

bspEnableIrq()	Allow interrupts
bspDisableIrq()	Inhibits interrupts
aaCriticalEnter()	Enters a critical section
aaCriticalExit()	Leaves a critical section
bspSaveAndDisableIrq()	Stores interrupt status word and inhibits interrupts
bspRestoreIrq()	Restores the interrupt status word
aaIntEnter()	To call when entering an Interrupt function
aaIntExit()	To call when leaving an Interrupt function

1.9.1 Types and priorities of interrupts

The kernel allows two types of interrupts.

- The "zero latency" interrupts. These interrupts have the highest priorities, greater than BSP_MAX_INT_PRIO, and are never disabled by the kernel. The latency of these interruptions therefore depends only on the hardware.
  These interrupts should never call the kernel API.
- The interrupts handled by the kernel, which are generally those used by device drivers. They have a priority between BSP_MAX_INT_PRIO and BSP_MIN_INT_PRIO. These interrupts are disabled by the kernel inside critical sections.

The priority BSP_MIN_INT_PRIO is defined in *bsp.h*. It is imposed by the number of bits of the processor interrupt manager priority mask. This is the lowest priority, or the least priority.

The priority BSP_MAX_INT_PRIO is defined in *bsp.h*, and can be modified as needed. This is the highest priority, or the most urgent.

1.9.2 Critical sections

1.9.2.1 Standard critical section

The recommended way to temporarily disable interrupts is to use a standard critical section: A critical section disables interrupts with priority lower than

BSP_MAX_INT_PRIO. The critical sections are reentrant: you must call *aaCriticalExit()* as many times as there was a call to *aaCriticalEnter()*.

1.9.2.2 Fast critical section

The *bspSaveAndDisableIrq()* and *bspRestoreIrq()* functions create fast critical sections (one to two assembly instructions). However, critical sections of this type are not taken into account by the mechanism for calculating the maximum inhibition time of the interrupts if it is validated.

These quick critical sections should be reserved for very short sections of code.

It is possible to include a fast critical section in a standard critical section, but the opposite is forbidden.

Fast critical sections are reentrant.

Example of use:

```
void    fn (void)
{
    aaCpuStatus_t   intState ;

    intState = bspSaveAndDisableIrq () ;
    .
    // Critical section code
    .
    bspRestoreIrq (intState) ;
}
```

1.9.3 Enabling and disabling interrupts

All interrupts can be globally enabled and disabled with *bspEnableIrqAll()* and *bspDisableIrqAll()*. These acts on the general interrupt validation flag of the CPU and therefore concerns all interrupts, even those called "zero latency". It is therefore recommended not to use these functions.

Interrupts managed by the kernel can be enabled and disabled with *bspEnableIrq()* and *bspDisableIrq()*. This works on the processor interrupt priority mask.

These two sets of functions are independent.

These functions are not reentrant: The use of these functions must be done with caution and must be reserved for very short suspensions which do not call for functions which can themselves use these functions.

One must not interleave a critical section (standard or fast) and these functions: their use is exclusive.

1.9.4 Interrupt Handler

When writing an interrupt handler it is absolutely necessary to call *aaIntEnter()* at the very beginning of the handler, then call *aaIntExit()* at the end of the handler.

Example:

```
//  SysTick interrupt handler

void    SysTick_Handler (void)
{
    aaIntEnter () ;
    aaTick () ;
    aaIntExit () ;
}
```

On ARM CORTEX-M interrupts are reentrant: several interrupts of increasing priority can nest.

1.9.4.1 Setting the priority of an interrupt

Managing interrupt priorities can be disturbing at first. Indeed, the relationship between priorities and their numerical value is not always intuitive. On the other hand, the MAX and MIN priorities are adjustable, so it is difficult to use predefined constants to define intermediate priorities.

To unify and simplify these notions an abstraction of the priorities of the interrupts is used by AdAstra.

The highest priority level is defined by BSP_MAX_INT_PRIO.

The lowest priority level is defined by BSP_MIN_INT_PRIO.

It is agreed that adding an offset to the lowest priority will progress to the highest priority. And conversely, by subtracting an offset at the highest priority, progress is made towards the lowest priority.

To implement this notion two functions are used:

bspIrqPrioMaxMinus(x)

The x parameter of this function indicates that we want to get the priority level which is x lower than the maximum level.

bspIrqPrioMaxMinus (0) corresponds to BSP_MAX_INT_PRIO, bspIrqPrioMaxMinus (1) corresponds to the level immediately lower than BSP_MAX_INT_PRIO, etc.

This function is useful for setting the higher interrupt levels.

bspIrqPrioMinPlus(x)

The x parameter of this function indicates that we want to get the priority level which is x higher than the minimum level.

bspIrqPrioMinPlus (0) is BSP_MIN_INT_PRIO, bspIrqPrioMinPlus (1) is the level immediately above BSP_MIN_INT_PRIO, and so on.

This function is useful for setting the lower interrupt levels.

Both functions guarantee a valid priority level, in the range BSP_MAX_INT_PRIO to BSP_MIN_INT_PRIO.

These functions do not allow to initialize constants. For this, two equivalent macros are defined:

BSP_IRQPRIOMAX_MINUS

BSP_IRQPRIOMIN_PLUS.

Example :

```
#define TIMER_PRIORITY  BSP_IRQPRIOMAX_MINUS (2)
```

1.10 System timer and low power

AdAstra kernel requires a timer to provide its services. This timer generates an interrupt, which is necessary for certain functions such as timeout management.

This timer must be provided by the hardware architecture: SysTick (on ARM Cortex-M architectures), standard timer or low energy timer. It is configured by the BSP.

In usual microcontroller applications, the CPU performs a task continuously. When the application has nothing to do, the "idle" task of the RTOS is executed: The timer is periodic, and the consumption is almost constant at a high level.

Some applications need to reduce the power consumption of the system when its workload decreases. There are several possible approaches.

Put the processor in sleep mode between the system ticks

The first easy approach is to ask the "idle" task to put the CPU to sleep, the CPU will be woken up every time the system timer or a device throw an interrupt.

This represents a significant gain in consumption. However, if the system timer frequency is high (usually 1 kHz), the CPU wakes up frequently for next to nothing.

Stretch the system timer.

The idea is to put the CPU sleeping as long as possible, so do not wake up for useless ticks. This technique is called "tick stretching" because timer interrupts are no longer periodic.

When the task "idle" must put the CPU to sleep, it asks the kernel to tell how many ticks it wants to be woken up. This is the shortest timeout or software timer.

The timer is then programmed for this duration, and the CPU is set to sleeping mode.

If the timer expires the BSP tells the kernel how many ticks has elapsed, and the kernel updates its state. Then the timer is reconfigured to resume the periodic rhythm (1 ms for example).

But the CPU can be woken at any time by another interruption than the timer. In this case you must:

- Calculate how many ticks have elapsed since the CPU was put to sleep, and warn the CPU to update its status.
- Reconfigure the timer to resume the periodic rhythm (at 1 ms for example).
- Release interrupts, allowing the interrupt waking up the CPU to be processed

All this with a number of constraints:

- The measurement of time must remain accurate. There is no question of losing a fraction of a tick when the timer is reconfigured, when the CPU is put to sleep or wakes up. The count of time is therefore the same, whether the ticks are periodic or stretched.

- AdAstra supports ZLI (Zero Latency Interrupts). These interrupts should not be inhibited to preserve this functionality. However on ARM Cortex-M, it is mandatory that the interrupts be inhibited during sleep by the WFI instruction. The implementation of stretched ticks must inhibit interruptions, but the design used minimizes interrupt inhibit time.

  Nothing is free or easy: To benefit from the energy saving provided by the stretching of the ticks, it is necessary to accept a higher latency of a few tens of nano seconds for ZLI interrupts.

- The ideal is to be able to put the CPU to sleep for as long as possible. However, there is a compromise to be made considering a) the granularity of the tick which must be fine if one needs a precise measurement of time, b) the number of bits of the counter of the timer on which the maximum duration of sleep depends. For example if the counter has 16 bits, and the tick granularity is 1 ms, the stretch of the tick can put the CPU dormant for up to 32 seconds.

- The nominal frequency of the tick can no longer be changed with *bspSetTickRate()*. For performance reasons this frequency is fixed by coding in the *bspTickStretch()* function of the BSP, and BSP_TICK_RATE must have the corresponding value.

The choice to configure the kernel to use the tick stretching is simply done by setting the value of AA_TICK_STRETCH in the file aacfg.h:

0 Periodic ticks.

1 Tick stretching.

Stretching ticks is handled by the BSP in the function *bspTickStretch_()* which is called by the task "idle". This function is part of the BSP because it is dependent on the timer used.

This function can use either of the above methods.

1.11 Mutual exclusion: The Mutex

A full mutex is an object providing exclusive access to a resource such as a device or data structure. It is created by *aaCreateMutex()*.

A mutex is a variant of the semaphore but with the following restrictions:

- It can only be used to ensure mutual exclusion
- It can only be given by the task owning it
- It cannot be acquired or given by an interruption
- You cannot flush on a mutex.

On the other hand, it has particular characteristics:

- It can be acquired recursively: the same task can acquire several times the same mutex, and then release it as many times as it has acquired.
- The mutex uses a task priority inheritance algorithm, to avoid the unbounded priority inversion phenomenon, which causes a low priority task to prevent another higher priority task from executing.

The priority inheritance mechanism is expensive in resources. It should consider many cases, such as tasks owning multiple mutexes, a task changing priority, or a pending mutex task that aborts on timeout. All of these cases require to recalculate the priority of the tasks owning mutexes that are also expected by other tasks.

If several tasks must acquire several mutexes, there is a risk of interlocking (deadlock). The user must implement a strategy to avoid this, for example by ensuring that all tasks acquire the mutexes in the same order, and release them in reverse order.

The maximum number of mutexes managed by the kernel is specified during kernel configuration.

1.12 Semaphores

AdAstra provides a counter-based semaphore, which is useful for guarding a multi-instance resource, such as an array of multiple elements. It is created by *aaSemCreate()*.

A semaphore can be seen as a counter. When a task acquires a semaphore using *aaSemTake()*:

- If the counter is greater than 0, the counter is decremented, and the task can continue execution.
- If the counter is less than or equal to 0, the task is suspended until the counter becomes positive or the optional timeout expires.

At the return of *aaSemTake()* an error code indicates the result of the operation.

Any number of tasks can be queued to acquire the semaphore.

When an interrupt task or function releases a semaphore with *aaSemGive()*:

- If no task is waiting for the semaphore, the semaphore counter is incremented.
- If a task is suspended pending the semaphore, this task is enabled, and returns with a code indicating the success of the operation. The counter is unchanged

If more than one task is waiting, the highest priority task is selected to be activated. If the selected task has a higher priority than the current task releasing the semaphore, the selected task is immediately activated.

A semaphore can be released by an interrupt function, but cannot be acquired by an interrupt function.

If a task is destroyed while it owns a semaphore, the semaphore remains in the state, which usually has an unpredictable effect on the application.

The maximum number of semaphores managed by the kernel is specified during kernel configuration.

In versions of AdAstra-RTK prior to 1.10 there was a variant of semaphore called simple mutex: a mutex without priority inheritance. This simple mutex has been removed.

1.13 Inter-task signals

Inter-task signals are an inter-task communication medium, which is very different from POSIX signals which are more like interrupts to handle errors and exceptions.

Each task has a number of signals, the number of which is defined by the size of the type *aaSignal_t* defined in *aabase.h*. (16 if *aaSignal_t* is defined as *uint16_t* for example).

A task can wait for one or more of its own signals to be positioned. Any task or interrupt can position a signal in a task. This mechanism is fast and allows for example a task to be activated at each occurrence of an interrupt, without having to use a semaphore more expensive in resources.

Two modes are used to wait for a signal:

AA_SIGNAL_AND The task waits until all requested signals are signaled.

AA_SIGNAL_OR The task waits until at least one of the requested signals is signaled.

When the task returns from *aaSignalWait()* the signals that were used to trigger the return are transmitted, and deleted from the task descriptor.

Example: A task whose identifier is *mainTaskId* waits for 300 ticks for two other tasks each set a signal with *aaSignalSend()* (0x80 and 0x40):

```
    aaSignal_t  sigsOut ;
    aaSignalReset ();
    res = aaSignalWait (0xC0, & sigsOut, AA_SIGNAL_AND, 300) ;
```

If the function returns AA_TIMEOUT, it is possible to examine *sigsOut* to determine which spot has not set its signal.

One of the expected tasks is running:

```
    (void) aaSignalSend (mainTaskId, 0x40) ;
```

The other expected task is running:

```
    (void) aaSignalSend (mainTaskId, 0x80) ;
```

1.14 Dynamic memory management

It is often convenient to use the dynamic memory allocation: *malloc()*, *free()*, and so on.

However, some programming rules may prohibit it. To meet all these needs, three options are available:

- Do not use dynamic allocation: in this case the functions aaM*alloc()*, *aaFree()*, etc., are not available.
- Use a block allocation: each block can only be allocated once because it cannot be released. This provides the flexibility of dynamic allocation with *aaMalloc()*, but avoids fragmentation or reuse of memory blocks with *aaFree()* and *aaRealloc()*. This is a very common case in embedded system design, for example, where tasks and buffers are all created once at system startup.

- Full dynamic memory allocation, using the TLSF algorithm.

For dynamic memory allocation the kernel offers the functions:

aaMalloc()

aaFree() TLSF only

aaTryFree() TLSF only

aaRealloc() TLSF only

aaCalloc() TLSF only

These functions are “thread safe”.

1.14.1 TLSF Dynamic memory algorithm

The algorithm used is derived from the Two-Level Segregate Fit (TLSF) whose description can be found AT http://www.gii.upv.es/tlsf//.

This algorithm has the following advantages in a real time system:

- Deterministic: TLSF has a constant cost in O(1).
- Fast.
- Effective, it limits the fragmentation of memory.

The algorithm has been adapted to processors with little memory. In a standard implementation each block allocated has a header which contains the management information of the block (size, chaining ...), and occupies 8 bytes in the general case of a 32-bit processor. In this implementation the overhead has been reduced to 4 bytes, but in return the size of the manageable memory pool is reduced.

The characteristics of the TLSF allocator are:

- The supplied block is always 8-byte aligned, which makes it compatible with the alignment required for an ARM stack, for example.
- The header of each block is only 4 bytes.
- The maximum size of the memory partition is 262143 bytes (256 KB). Multiple partitions of the same size can be used.
- The descriptor block of the partition is taken from the partition itself when it is initialized.

The space occupied by the descriptor of the memory partition depends on the size of the memory partition. In the file *aatlsf.c* it is possible to indicate the maximum size of the memory space managed with FLI_MAX_INDEX, in order to optimize the size of the descriptor:

FLI_MAX_INDEX	Partition size (bytes)	Descriptor size (bytes)
17	262143	444 Bytes, 0.16%
16	131071	408 Bytes, 0.31%

15	65535	372 Bytes, 0.56%
14	32767	336 Bytes, 1.0%
13	16383	300 Bytes, 1.8%
12	8191	264 Bytes, 3.2%

When the user creates a TLSF partition managed by himself (without using the *aaMalloc()* API family), a specific function set allows access to it.

1.14.2 Block allocation algorithm

This allocation uses a very fast and very simple algorithm. It is initialized by providing a block of memory (the memory partition), then it subdivides it on demand, without fragmentation.

The partition descriptor is small: two 32-bit words for a 32-bit system. This descriptor is allocated at the beginning of the partition itself.

The allocated blocks have the following characteristics:

- They have a variable size, always multiple of 8 bytes
- They are aligned on an 8-byte boundary.
- They do not have an overhead
- An allocated block cannot be released.

It is possible to use several block allocation partitions at the same time, for example in different memory area if the microcontroller supports it.

1.14.3 Configuring memory allocation

The configuration of the dynamic memory allocation is done by setting constants in the file *aacfg.h*.

It is possible to include either or both of the dynamic allocation algorithms in the kernel. This is done with:

```
#define   AA_WITH_TLSF       1   // Include TLSF

#define   AA_WITH_MEMBLOCK   1   // Include memory block allocator
```

The function *aaMalloc()* (as well as the other functions of the family if they are valid) is generic: it adapts to the algorithm assigned to dynamic allocation. For this one need to define only one of the constants:

```
#define   AA_WITH_MALLOC_TLSF   1  // Dynamic memory allocation
                                   // aaMalloc() enabled and use TLSF

#define   AA_WITH_MALLOC_BLOC   1  // Dynamic memory allocation
                                   // aaMalloc() enabled and use bloc
                                   // allocator
```

The declaration of the constant AA_WITH_MALLOC_TLSF causes the declaration of the constant AA_WITH_MALLOC which is used by the kernel to know if the dynamic allocation of memory is usable.

The *aaRealloc()* function is complex and it is possible to save its code size by inhibiting it with the definition of AA_WITH_REALLOC in *aacfg.h*.

When the dynamic memory management is allowed, the kernel automatically creates a memory partition with the heap information provided by the linker script (_heap_hegin and _heap_end), calling *aaMallocInit()*.

If the partition must be placed in a specific place in memory by ignoring the linker information, three constants must be defined:

```
#define   AA_WITH_USERHEAP   1
#define   AA_HEAP_BEGIN      xxx        // Heap address
#define   AA_HEAP_SIZE       yyy        // Heap byte size
```

It is then possible for the application to use the functions *aaMalloc()*, *aaFree()*, and so on.

In the case of TLSF allocation the *aaMemPoolCheck()* function can be used to check the integrity of the dynamic allocation, which is possible thanks to the information contained in the descriptor of the partition and the header of each block.

1.14.4 Memory allocation port

The files *aatlsf.c* and *aatlsf.h* realize the allocation of dynamic memory TLSF.

The files *aamemblock.c* and *aamemblock.h* realize the allocation of dynamic memory by block.

The API (*aaMalloc()*, *aaFfree()* ...) is implemented by the files *aamalloc.c* and *aamalloc.h*.

To change the dynamic allocator just re-implants the file *aamalloc.c*.

1.15 Software Timers

A software timer can be created by any task, it is then used to call a callback function after a predetermined time. The callback function is executed in the context of the system tick interrupt.

A software timer is:

- Created by *aaTimerCreate()*, in the idle state.
- Configured with *aaTimerSet()*, to indicate the callback to use as well as the delay to use in number of tick system.
- Started with *aaTimerStart()*.
- Stopped before its end by *aaTimerStop()*.
- Destroyed by *aaTimerDelete()*.

The prototype of the callback function is:

typedef uint32_t (aaTimerCallback) (uintptr_t arg) ;*

The timer can be used as "one shot": when the callback is called the timer is in the stopped state. If the callback returns 0 it will remain stopped.

The timer can be periodic: If the callback returns a non-zero value the timer is reset with the timeout value specified by *aaTimerSet()* and then restarted.

The callback can reconfigure the timer before returning a non-zero value, which makes it possible to have a timer with a variable duration.

The callback of a timer is called in an interrupt context. The restrictions applicable to interrupt drivers therefore apply to this callback: among other things do not take a semaphore, do not call a blocking API, be as short as possible.

A timer can be used as a watchdog: started at the beginning of a treatment, it can be restarted during processing to avoid the timer expiry, and then stopped once this treatment is finished. If the callback is called, the treatment has lasted too long.

```
uint32_t  cbTimer1 (uintptr_t arg)
{
     aaLogMes ("Watchdog %d\n", arg, 0, 0, 0, 0) ;
}


void fn ()
{
     aaTimerId_t          t1 ;

     // Create a timer with a timeout of 20 ticks
     aaTimerCreate (& t1) ;
     aaTimerSet (t1, cbTimer1, 1, 20) ;
     aaTimerStart (t1) ;

     while (! end)
     {
          aaTimerStart (t1) ;
```

```
            // Treatment
        }
        aaTimerDelete (t1) ;
}
```

The maximum number of timers managed by the kernel is specified during kernel configuration.

1.16 Message queues

Message queues are an inter-task communication tool. They allow:

- A queue has a variable number of messages to be stored in a buffer managed in FIFO. The maximum number of messages and the maximum message size are specified when creating the message queue with *aaQueueCreate()*.
- The messages are stored by copying in the message queue buffers.
- The messages can be of variable length, only the useful part of the messages is recopied, in and out.
- Any task or interrupt can send a message to a message queue. If the queue is full the task can be blocked with timeout, and if the caller is an interrupt the error AA_EWOULDBLOCK is returned.
- Any task can read a message in the message queue, and if it is empty the task will be blocked with timeout. If a read on an empty queue is requested by an interrupt, or if the specified timeout is 0, the return value is AA_EWOULDBLOCK.

When creating the message queue the user can provide the address of the buffer that will be used to store the messages, allowing for static allocation. If the address is NULL and dynamic memory management is allowed, the queue handles buffer allocation and release.

Several tasks can wait to write or read a message in a message queue. The *flag* parameter of the function *aaQueueCreate()* allows to specify the order of the tasks:

AA_QUEUE_PRIORITY	Tasks are handled in order of priority: the highest priority task will get the first available message.
AA_QUEUE_FIFO	The tasks are managed in their order of arrival. This is the default mode when creating the queue.

The maximum number of message queues managed by the kernel is specified at kernel configuration.

A particular case of message queue management occurs when the message consists of only one pointer. In this case only the pointer is copied, which is fast, and the information is stored in an external buffer by the application. The address of the buffer is transferred from the transmitter to the receiver without copying the message body.

To use this pointers management, create the queue with the AA_QUEUE_POINTER flag.

To facilitate this type of management it is possible to use the buffer pools.

1.17 Buffer pools

The buffer pool is a mechanism allowing for example the use of message queues without copying data.

A buffer pool consists of a series of buffers chained in a list, which is the list of free buffers.

When creating the buffer pool (with *aaBufferPoolCreate()*) the user can provide the address of the memory block that will be used to build the buffers, which allows for static allocation. If the address is NULL and dynamic memory management is allowed, the pool manages memory allocation and release.

Example of use:

When a task needs a buffer, it requests it from the pool with aaBufferPoolTake(), the buffer is taken from the list of free buffers. Then the task can use it: fill it with data and then put it in a message queue by supplying the buffer address to the queue.

The task reading the message queue receives the buffer address, uses the contents of the buffer, and then returns it to the pool with *aaBufferPoolGive()*. This will put this buffer in the list of free buffers.

The maximum number of buffer pools that the kernel manages is specified during kernel configuration.

1.18 Board Support Package

The BSP groups the kernel features that depend on the system used. Some features are accessible to the user through generic functions such as *bspGetTickRate()* or *bspOutput()* for example.

The BSP offers access to some often used resources:

- GPIOs associated with LEDs or buttons. Other GPIOs can be added, for debug on the oscilloscope for example.
- A generic UART driver to have a console.
- Configure and query the system tick.

In addition to these user functions, the BSP provides kernel support:

- Initialization of the hardware
- System tick support
- Interrupt support
- Management of the task context switch.

1.19 Debug

The provision of tools to facilitate debugging is an important feature of an embedded system.

AdAstra RTK offers some of these tools: a console, the logMes feature and the use of the SWO signal of ARM Cortex, a trace system, an AA_ASSERT macro, and centralized handling of errors.

1.19.1 Console

A console is the basic tool needed for debugging. It is possible to use any UART / USART available in the processor for this.

The console is configured in *aacfg.h (AA_WITH_CONSOLE)*. In addition it is necessary to define in the *uartbasic.c* file the UART / USART which will have to be managed by the kernel. The console is initialized by the kernel, and the handle of the corresponding UART is available in the global variable *aaConsoleUartHandle*.

If a development board is used, it is often convenient to use as the console the UART routed to the programming / debug probe.

1.19.2 aaLogMes

Showing console traces can be an effective way to debug. However, these traces tend to change the timing of the application and in that sense they are intrusive.

A low-intrusive trace system is based on the *idle* task and the *aaLogMes()* function.

The *aaLogMes()* function has a syntax similar to that of *printf()*, but instead of immediately formatting the message, it places the arguments in a message queue.

This function also has the advantage of being able to be called by an interrupt function.

In a second step, the *idle* task extracts messages from the queue to format these messages and send them. It is possible to choose which device will be sent messages, most often a UART or SWO.

The timing of the application is undisturbed because:

- The formatting of the messages, which can be time-consuming, is not realized by *aaLogMes()* who moreover does not have access to any peripheral, which prevents him from being possibly blocked.
- The *idle* task runs with a lower priority than the application tasks, and thus run during the instants left free by the application.
- The use of SWO is fast and does not involve the use of interrupt as a UART.

The *aaLogMes()* arguments are not used when the function is called, but later by the *idle* task. This means that the arguments must still be valid at this time, which excludes pointers to volatile data such as pointers or data allocated on the stack.

In the following example, the use of *str* is invalid because when *idle* processes the *str* message, it may not exist anymore. The use of *pStr* is valid because the value pointed to by *pStr* will not change.

The use of *arg* is valid because *arg* is not a pointer and its value will still be valid.

```
void myFunc (uint32_t arg)
{
    char        str [32] ;
    const char  * pStr ;

    pStr = "Hello" ;


    aaLogMes ("myFunc %s %s %d\n", str, pStr, arg, 0,0) ;
}
```

The message queue used by the trace system has a limited depth. If the application generates more messages than the *idle* task can handle, the queue may be full and in this case messages will be ignored. This is indicated by the message "logMes lost: xx" where xx is the number of lost messages.

The message queue overflow can for example be caused by a task that generates messages in a permanent loop: which is not suspended from time to time by waiting for an I/O, a semaphore, a mutex, or *aaTaskDelay()*.

In AdAstra-RTK kernels prior to version 1.10, the formatting of logMes messages was handled by a dedicated tLogM task. Now formatting is handled by the idle task, which saves resources.

1.19.3 Stacks monitoring

When developing an application, problems due to stack stacks can be difficult to detect. The kernel and the BSP must facilitate the detection of errors in stack management.

AdAstra-RTK allows to monitor task stacks, i.e. to detect if they are approaching or exceeding their limits. To validate the stack monitoring of a task, you must provide the AA_FLAG_STACKCHECK flag when creating the task.

When the AA_FLAG_STACKCHECK flag is provided, the stack is initialized with words of known value. Monitoring is performed by the kernel during each context change: (when the task is inactivated) by performing 2 tests:

- Check that the current pointer of the stack is not greater than the limit. This makes it possible to detect a proven stack overflow.
- Compares words 7 and 8 of the stack with the padding values of the stack. If the words have a different value, this indicates that the stack usage is very close to the maximum.

When one of the tests is positive, the kernel calls the *aaUserNotify()* function which must be defined by the user (a default implementation is in *userInitTask.c*). The arguments of this function let you know which task is impacted and which test is signaled.

The user can use the *aaTaskCheckStack()* function at any time to determine the use of the stack whose monitoring was requested during the creation of the task.

The system stack used by interrupts is independent of any task. It has a dedicated function to allow its monitoring: *bspMainStackCheck()* which allows to know the number of unused words in this stack.

The memory used by the system stack is systematically initialized to a value known by the BSP, using the limits of the stack provided by the linker.

1.19.4 SWO

The Single Wire Output (SWO) pin is a feature of ARM Cortex. It can be used in many ways. AdAstra RTK uses it as a very fast UART output (several tens of MBits / s), in combination with the ITM mechanism, optimizing the throughput.

The ARM's ITM mechanism allows 8, 16 or 32-bit words to be transmitted through 32 channels (called ARM stimuli) to the SWO pin. Each channel precedes the transmitted word with a header that allows the receiver to identify the sending channel. Channel 0 has the particularity of having headers whose value is less than 0x04, and therefore not displayed by a terminal.

If the messages generated by *aaLogMes()* are sent to SWO by channel 0 of the ITM one has a readable tracing mechanism with high throughput and very little intrusive for the application.

The ITM / SWO mechanism is only available in debug mode. For this reason, by default, the BSP initializes the ITM mechanism and directs the messages generated by *aaLogMes()* to it when DEBUG is defined.

To display the traces, it is necessary to have a suitable receiver. For example, it is possible to use the FTDI C232HM cable, which provides the USB equivalent of a 12 Mbits / s UART.

The BSP provides some functions for:

- Initialize the SWO mechanism: *swoInit()*.
- Emit characters or words: *swoSend8()*, *swoSend16()* and *swoSend32()*.
- Send a string terminated by a 0: *swoSendStr()*.
- Use standard output to send a character to channel 0: *swoSend()*, which is used for example to direct messages from *aaLogMes()* to SWO.

1.19.5 The AA_ASSERT macro

The AA_ASSERT macro is defined in *aacfg.h*, if AA_WITH_DEBUG is set to 1. Its prototype is:

```
void assert(scalar expression);
```

If expression is false (is 0), the *bspAssertFailed()* function defined by the BSP is called. This function has a different behavior depending on the presence or absence of a debugger:

- If the application was launched by a debugger, the application stops in the debugger, which allows inspecting the cause of the shutdown.
- If the application has not been launched by a debugger, it enters an infinite loop after inhibiting interrupts.

Note: The assertions of the HAL arrive at the same place.

The *bspAssertFailed()* function is set with the "weak" attribute by the BSP. This allows the application to redefine it, and to affect it differently. For example, to light an LED in case of error, this allows to be alerted even if there is no active debugger.

1.19.6 Centralized handling of errors

Error management is often a difficult task. In the code of the applications it is necessary to systematically test the returned code of the called functions, and in case of error, often to raise the error of several levels of functions nesting, then to make a decision to handle this error.

This generates a large amount of code, which must be checked and tested. In addition, the code must be read carefully to ensure that all the return values are tested, which is tedious, with no guarantee of completeness, even if compiling options may help.

To lighten and systematize these procedures, the AdAstra kernel uses a centralized management of errors: All the errors recorded by the kernel cause a call to the *bspErrorNotify()* function. This function is defined in the BSP with a default behavior, but it can be redefined by the user to adapt its behavior to particular cases.

The fact that all the errors are directed towards this function guarantee that none can be forgotten and those are all handled, without imposing any constraint on the code of the applications. If the centralized error handling is used, this allows the application to no longer have to test the return values of the kernel functions, since in case of fatal error the error handling function is called. Moreover in debug mode this function calls the debugger: the user does not need to set a breakpoint to trap these errors.

Centralized error handling is configured in *aacfg.h*, by setting AA_WITH_ERRORCHECK to one of the following values:

AA_WITH_ERRORCHECK_NONE
: There is no centralized error handling, errors are not reported to *bspErrorNotify()*, assertion checks are not performed.
This option must be reserved for very specific cases.

AA_WITH_ERRORCHECK_ASSERT
: In case of an error, the application generates a breakpoint and calls the debugger from where the error was detected, which makes it possible to immediately locate the error.
This option must be reserved for the development phase, during which the debugger is in use. In case of error while the debugger is not in use the behavior is not guaranteed (e.g. generation of a "HardFault Exception").

AA_WITH_ERRORHECK_NOTIFY
: This is the most general case: the *bspErrorNotify ()* function is called, and when using the debugger generates a breakpoint. If the debugger is not in use, it manages its return according to the flags associated with the error number. If the return is forbidden it enters an infinite loop after inhibiting interruptions.

To centralize the errors the *aaerror.h* file defines error numbers, flags, and macros.

All errors have a different number, and are associated with flags to configure the processing. These flags are:

AA_ERROR_FATAL_FLAG
: The error is flagged fatal: normally the application is no longer able to continue to run if such an error occurs and the application ignores it.

AA_ERROR_NORETURN_FLAG
: After this error, the *bspErrorNotify()* function should not return.

AA_ERROR_FORCERETURN_FLAG
: After this error the function *bspErrorNotify()* can return, even if the flag AA_ERROR_NORETURN_FLAG is also present. In other words AA_ERROR_FORCERETURN_FLAG has priority over AA_ERROR_NORETURN_FLAG.

  This flag allows after *bspErrorNotify()* reported the error to let the application handle it.

The macros used for the centralized handling of errors allow several operations:

AA_ERRORNOTIFY (errorNumber)

This macro is the basis of the centralization of errors. It handles the report of the error according to the value of AA_WITH_ERRORCHECK: breakpoint of the debugger or call of *bspErrorNotify()*.

This macro always uses AA_ERROR_NORETURN_FLAG, so the *bspErrorNotify()* function should not return.

However if it is necessary that the application continues after the signaling of this error the flag AA_ERROR_FORCERETURN_FLAG can be used.

AA_ERRORASSERT (test, errorNumber)

This macro performs a test and if the test result is false, call *AA_ERRORNOTIFY(errorNumber)*.

In this use case the *bspErrorNotify()* function should not return, so do not use AA_ERROR_FORCERETURN_FLAG.

AA_ERRORCHECK (test, returnValue, errorNumber)

This macro makes it possible to perform a test similar to AA_ERRORASSERT, and the *bspErrorNotify()* function must return. After reporting the error a return to the calling function with the return value *returnValue* is executed. For this it systematically associates AA_ERROR_FORCERETURN_FLAG with errorNumber.

The macro AA_ERRORCHECK has the following pseudo code:

```
if (! (test))
{
    AA_ERRORNOTIFY ((errorNumber) | AA_ERROR_FORCERETURN_FLAG);
    return returnValue ;
}
```

It is sometimes necessary to insert code in the processing of the error, before returning to the calling function. For this we can use the following pseudo code:

```
if (! (test))
{
     // User code

     AA_ERRORNOTIFY ((errorNumber) | AA_ERROR_FORCERETURN_FLAG);

     // User code

     return returnValue ;
}
```

Some error numbers are reserved for the application in the file *aaerrors.h*. The application can thus benefit from the centralization of error handling by using these error numbers and the macros defined in this file.

1.19.7 Traces

The traces are described in a dedicated chapter.

2 Traces

AdAstra provides a built-in kernel trace mechanism. These traces make it possible, among other things, to follow in real time the kernel API used by the tasks and the interrupts. Traces are therefore more suitable for studying the behavior of applications.

2.1 Traces configuration

Traces are allowed globally at the time of kernel compilation by the definition of AA_WITH_TRACE in *aacfg.h*:

AA_WITH_TRACE = 0 No trace is generated, which allows to get the kernel in release mode.

AA_WITH_TRACE = 1 The traces are generated.

When AA_WITH_TRACE is 1 it is possible to choose which traces will be generated by setting the various constants AA_WITH_T_xxx to 0 or 1. For example :

```
#define AA_WITH_T_IOWAIT    1   // Task waiting I/O   task id

#define AA_WITH_T_INTENTER  0   // Interrupt enter    irq num
#define AA_WITH_T_INTEXIT   0   // Interrupt exit     irq num
```

- The tasks that wait for an event from a driver will be traced, and the trace parameter is the identifier of the task.
- The interrupts will not be traced, and the parameter of these traces is the interrupt number.

Traces are numerous, more than 70, and some may have a high frequency. It is recommended to select only the necessary traces.

2.2 Enabling traces

By default the traces are not enabled. To do this, use the *aaTraceEnable(1)* function. The *aaTraceEnable(0)* function disable traces.

This makes it possible to generate the traces exactly when it is necessary.

While enabling traces, information on the tasks already created are emitted, this allows an easier exploitation of these traces. This enabling may take a little longer than other traces.

2.3 Traces implementation

When designing the trace mechanism, compromises must be made:

- The traces must be as extensive as possible, in order to provide the information needed to validate the application or to solve the problems encountered.

- The traces must be the least intrusive possible: change as little as possible the temporal behavior of the kernel for example.
- Use with a debugger, or in release build.
- The traditional means used to record traces are either an array in memory, or real-time broadcast.

The array in memory necessarily has a limited size, but modifies very little the temporal behavior of the kernel.

The real-time broadcast allows long-term recordings, but is more or less intrusive depending on the device used.

There is no perfect solution. Therefore the mechanism is conceived in two parts:

- Traces calls predetermined functions, with parameters.
- These functions do not exist as such: in the *aabase.h* file these functions are defined as macros that must be implemented to perform the trace.

Thus the user can implement a storage or transmission of traces according to the protocol he chooses.

2.4 User traces

The set of macros and trace functions can be extended by the user to transmit specific information.

The following traces are predefined and can be used freely by the application:

```
aaTraceUser1_1x8       (arg)          1 paramètre 8 bits
aaTraceUser1_2x8       (arg1, arg2)   2 paramètres 8 bits
aaTraceUser1_1x8_1x16  (arg1, arg2)   1 paramètre 8 bits et un de 16
aaTraceUser1_1x32      (arg)          1 paramètre 32 bits

aaTraceUser2_1x8       (arg)
aaTraceUser2_2x8       (arg1, arg2)
aaTraceUser2_1x8_1x16  (arg1, arg2)
aaTraceUser2_1x32      (arg)
```

The text and display format of user traces can be configured for viewing by the aaView software.

2.5 Trace example

The *aatrace.c* file is an example of functions associated with macros in the *aabase.h* file. The assumptions that have been retained to design this file are:

- Using the SWO link to issue the traces. It is a fast link (currently 12 Mbits / s is less than a micro second per octet) which has a small FIFO. The disadvantage is that it can only be used if it is configured by a debugger.
- The lowest data stream. For this purpose the identifiers of the objects (tasks, semaphores ...) are limited to 8 bits. Therefore it is not possible to plot more than 255 objects of each type, but this covers the vast majority of applications on the type of processor concerned.

- Trace dating uses the *aaTsGet()* function and uses a 32-bit word. With this function that uses the cycle counter of the CPU (Except on Cortex M0(+)), the dating is very accurate, but loops after a relatively short time: The application that receives the trace must take this into account.

The traces use the stimulus port 1, which is compatible with the SWO implementation of the *aaLogMes()* task that uses the stimulus port 0. Therefore the SWO output may contain a mixture of traces and messages generated by *aaLogMes()*. Traces take precedence over log messages.

It is easy to consider modifying this file to use, for example, a serial link, by emulating the ITM protocol, which would make it possible to use the traces without debugging.

2.6 The aaVieew application

The aaVieew application is a Microsoft Windows console application that can receive traces from a serial link and display them in clear.

The protocol used is ITM, which allows log traces and messages to be displayed.

The traces are displayed in real time, but the speed of display can hinder their analysis. In this case several approaches are possible:

- If the application manages the traces with *aaTraceEnable()* the traces can be stopped and analyzed.
- Otherwise it is possible to select the traces and copy them in a text editor such as Notepad ++ to save them and analyze them in deferred time.

The aaView application uses the aaView.ini configuration file by default. This allows you to specify the COM serial link number, its baud rate, the MCU frequency, and the display formats for user traces.

Example configuration file:

```
[COM]
comPort     = 6
baudrate    = 12000000
; Time stamp frequency in Mhz
tsClock     = 168

[USER1_1x8]
text        = myUSER1_1x8
format      = %u

[USER1_2x8]
text        = myUSER1_2x8
format      = %u  %u

[USER1_1x8_1x16]
text        = myUSER1_1x8_1x16
format      = %u  %u
```

```
[USER1_1x32]
text        = myUSER1_1x32
format      = %u

[USER2_1x8]
text        = myUSER2_1x8
format      = %u

[USER2_2x8]
text        = myUSER2_2x8
format      = %u  %u

[USER2_1x8_1x16]
text        = myUSER2_1x8_1x16
format      = %u  %u

[USER2_1x32]
text        = myUSER2_1x32
format      = %u
```

aaView command line syntaxe :

```
aaView [-com N] [-br B] [-fr MHz] [-c filepath]
```

It allows you to specify:

- The number of the COM serial link
- His baud rate
- The frequency of the MCU
- The path of a configuration file.

The parameters present on the command line take precedence over the parameters of the configuration file.

Trace example:

```
// aaView V1.2
// Configuration file: .\aaView.ini
// COM6 12000000 bauds, MCU 168 MHz
// Hit 'q' then enter to quit

 202139348    0.0us TASKINFO             ta   0  pr   0  tIdle
 202139586    1.4us TASKINFO             ta   1  pr   1  tLogM
 202140782    7.1us TASKINFO             ta   2  pr   2  tInit
 202143162   14.2us TASKINFO             ta   3  pr  15  tLeds
 202146151   17.8us INTENTER             it 38
 202152167   35.8us TASKIOWAIT           ta   2
 202152523    2.1us TSWITCH              ta   2 -> ta   0
 202233642  482.9us INTENTER             it 38
 202321081  520.5us INTENTER             it 38
 202408582  520.8us INTENTER             it 38
 202496043  520.6us INTENTER             it 38
 204595561  520.7us INTENTER             it 38
 204683053  520.8us INTENTER             it 38
 204770521  520.6us INTENTER             it 38
 204858023  520.8us INTENTER             it 38
 204945491  520.6us INTENTER             it 38
 205032961  520.7us INTENTER             it 38
 205120459  520.8us INTENTER             it 38
 205207921  520.6us INTENTER             it 38
 205295401  520.7us INTENTER             it 38
 235177217  177.9ms TASKREADY            ta   3
 235177651    2.6us TSWITCH              ta   0 -> ta   3
 235178107    2.7us TASKDELAYED          ta   3
 235178360    1.5us TSWITCH              ta   3 -> ta   0
 258566909  139.2ms INTENTER             it 38
 258567247    2.0us TASKREADY            ta   2
 258567633    2.3us TSWITCH              ta   0 -> ta   2
 258568323    4.1us MSG                  Trace stop
```

3 Writing an application

This chapter explains the initialization mechanisms of the kernel, and then how to start the execution of an application.

3.1 Kernel configuration

The kernel is configured with the aacfg.h file, which contains definitions, the main ones being:

AA_PRIO_COUNT	The count of priorities that the kernel must handle. This number must be between 3 and 256. The performance is optimal with a number at most equal to the number of bits of a word of the processor (for example 32 if the processor manages 32-bit words).
AA_TASK_MAX	The count of tasks required for the application. This number is independent of the number of priorities handled by the kernel.
AA_MUTEX_MAX	The count of mutexes needed by the application.
AA_SEM_MAX	The count of semaphores needed by the application.
AA_TIMER_MAX	The count of timers needed by the application.
AA_QUEUE_MAX	The count of tails required by the application.
AA_BUFPOOL_MAX	The count of buffer pools needed by the application.
AA_WITH_LOGMES et AA_LOGMES_MAXBUF	*aaLogMes()* configuration.
AA_INIT_xxx	The parameters needed to create the first task of the application.
AA_WITH_CONSOLE	Set the characteristics of the standard output as *aaPrintf()*.

The following definitions set the measure and debug functions:

AA_WITH_ARGCHECK	Inserts tests on the validity of the parameters of the main functions: mutex, semaphore, queue, pool buffer...
AA_WITH_DEBUG	Validates AA_ASSERT controls in the kernel and in the application.
AA_WITH_TASKSTAT	Calculates the CPU usage time for each task.
AA_WITH_CRITICALSTAT	Calculates the maximum duration of a critical section (not implemented)

The use of these very useful features is intrusive: they can disrupt the timing of tasks and degrade to a certain extent the performance of the kernel.

The following definitions set the dynamic memory management. See also Dynamic memory management.

AA_WITH_TLSF If set to 1 indicates that TLSF algorithm is implemented.

AA_WITH_MALLOC_TLSF If set to 1 indicates that dynamic memory management uses the TLSF algorithm.

AA_WITH_REALLOC If set to 1 indicates that the realloc feature is implemented. This feature is optional because it takes up a lot of code and is rarely useful in an embedded system.

AA_WITH_MEMBLOCK If set to 1 indicates block allocation is implemented.

AA_WITH_MALLOC_BLOCIf set to 1 indicates that dynamic memory management uses the block allocation algorithm

AA_WITH_MALLOC_BLOC_ERRORFREE If set to 1 indicates a fatal error should be triggered if *aaFree()* is used, if set to 0 indicates that *aaFree()* does nothing.

There are other definitions concerning for example the printing of floats, the use of newlib… Refer to the *aacfg.h* file.

Centralized management of fatal errors: see §1.19.6 "Centralized handling of errors".

The configuration of the heap:

AA_WITH_USERHEAP Set to 0 indicates that the heap occupies all available memory between the BSS section and the system stack. This is defined by the linker's script.

Set to 1 indicates that the user wants to set the location and size of the heap with the definitions AA_HEAP_BEGIN and AA_HEAP_SIZE.

Configuration of traces: see Traces.

3.2 System initialization

The system initialization mechanism is explained here for an ARM Cortex-M system.

Several files are used:

vectors_xxx.c This file contains the CPU interrupt and exception vector table. In particular the 2 vectors used for startup:

Vector 0 contains the address of the MSP stack (main stack)

Vector 1 contains the address of the first function to call: *_start()*.

This information is provided by the linker script.

startup.c	Contains the *_start()* function that is the first called when starting the processor. After initialization, it calls *bspMain()*.
bsp.c	Contains functions called from *_start()* for minimal hardware initialization: clock, FPU ...
system_stm32xxx	File provided by the processor producer with the *SystemInit()* function.

The system initialization process is as follows:

- At the start of the processor the control is transferred to the function whose address is in the interrupt vector 1, which is *_start()* which:
- Calls *bspSystemInit_()*: Calls *SystemInit()*, and sets the vector table in the right place (in RAM if necessary). Performs any operations to be performed at the earliest after the start of the processor
- Uses linker information to initialize initialized and uninitialized data sections (BSS) in different RAM areas.
- Call *bspHardwareInit_()*: configure FPU, system clock, NVIC_SetPriorityGrouping.
- Call *bspMain()* which which has no return.
- *bspMain()* performs some initializations of the hardware and the BSP. And then Calls *aaMain()* which is the entry point of the kernel and which has no return.

Hardware FPU configuration:

In the Eclipse C / C ++ Build / Settings / Target Processor configuration select:

- Float ABI : FP instructions (hard)
- FPU type : fpv4-sp-d16 (for Cortex M4)

This allows the core_cm4.h file to generate the constant __FPU_USED which is itself used by *bsp.c* to manage the FPU.

Software FPU configuration, when you do not want to use a real number:

In the Eclipse C / C ++ Build / Settings / Target Processor configuration select:

- Float ABI : Toolchain Default
- To print floating numbers with nanolib *printf()*, add "-u _printf_float" to the linker command line. aaPrinf() can also print float values.

3.3 Kernel initialization

The initialization of the kernel is done by *aaMain()* which is in *aaMain.c*.

The sequence of initialization operations is as follows:

- If AA_USART_CONSOLE is set in aacfg.h the UART console link is initialized, the outputs of *aaPrintf()* and *aaLogMes()* are directed to this console.

Note: If DEBUG is set, the BSP directs the outputs of *aaLogMes()* to the SWO output.

- Initialize AdAstra RTK by calling *aaInit()*.
- Displays the banner defined in the BSP with *bspBanner()()*, if BSP_WITH_BANNER is defined to 1 in *bspcfg.h*
- Initializes the dynamic memory allocation according to what is configured
- Creates the "idle" task of priority 0.
- Initializes the additional components: queues, timers, ...
- Created the first task with the configuration defined in *aacfg.h* by constants AA_INIT_xxx.
- Start the kernel by calling *aaStart()*.

The first task continues kernel initializations that require the kernel to be started, and then calls the *userInitTask()* function that must be defined by the user.

3.4 Application initialization

The first task performed by the kernel calls the *userInitTask()* function, written by the user, which is also the entry point of the first user task. Its configuration is defined in *aacfg.h* by constants AA_INIT_xxx.

This function is called after the complete initialization of the system, the set of resources is therefore usable.

In most cases this function realizes:

- A change of priority of the task so that it corresponds to the need of the application (if the priority is not set in *aacfg.h*).
- Initialization of the application environment.
- Create the other tasks of the application.

The *userInitTask()* function has the prototype of a task function, i.e.:

```
void userInitTask (uintptr_t arg) ;
```

The stack size of the first task is defined by AA_INIT_STACK_SIZE. This stack is allocated dynamically if AA_WITH_MALLOC is set and is equal to 1. Otherwise a static array is automatically allocated.

3.5 STMicroelectronics Hardware Abstraction Layer

The HAL supplied by STMicroelectronics is not designed to coexist with an RTOS.

The main problem is that the HAL configures the systick timer, while this is handled by the kernel in due time. STM recommends using another timer dedicated to HAL. But you do not have to use two timers to do the same thing twice.

To work around this problem and use HAL, the user must:

- Add the USE_HAL_DRIVER preprocessor symbol. This includes the initialization functions of HAL and informs the BSP to call HAL_IncTick ().
- Do not use HAL_Init () or HAL_SYSTICK_Config (). The kernel performs the appropriate initializations.
- Call HAL_MspInit () in the first task of the user, before calling any other function of the HAL. As long as the kernel is not started the systick timer is stopped. Initializing the HAL once the kernel has started allows for proper operation of the timeout loops used by the HAL.
- Do not use HAL_SuspendTick () / HAL_ResumeTick (). This disrupts the operation of the real-time kernel. These functions are redefined by the kernel with an AA_ASSERT (0) to warn the user when used.

If the HAL is not used, remove the USE_HAL_DRIVER symbol from the preprocessor.

4 Reference Manual

4.1 Miscellaneous

aaVersion

```
uint32_t      aaVersion      (void)
```

Description TOC §↑

This function returns the kernel version as 2 values in an integer 0xVVVVRRRR:

- VVVV Kernel version.
- RRRR Kernel revision.

Example

- 0x00010000 is version 1.0.
- 0x00020008 is version 2.8

Return value

The kernel version

4.2 Task management

aaTaskCreate

```
aaError_t   aaTaskCreate    (uint8_t           prio,
                             const char        * pName,
                             aaTaskFunction    pEntry,
                             uintptr_t         arg,
                             bspStackType_t    * pStack,
                             uint16_t          stackSize,
                             uint8_t           flags
                             aaTaskId_t        * pTaskId)
```

Description TOC §↑

This function allows you to create a task.

prio	The priority of the task is between 0 and AA_PRIO_COUNT-1. Priority 0 is reserved for the task idle, and AA_PRIO_COUNT is the number of priority levels managed by the kernel..
pName	A pointer to a string that is the name of the task. The maximum size is AA_TASK_NAME_SIZE, including the final 0.
pEntry	A pointer to the function the task should execute.
arg	The pEntry function argument.
pStack	Pointer to the memory area that will serve as stack for the task. If this pointer is non-NULL the user takes care of the allocation of the stack at the time of the creation of the task, and the release of the stack during the destruction of the task. This allows having a completely static allocation. If the stack pointer is NULL and dynamic memory allocation is allowed, then the kernel takes care of dynamically managing the stack. The stack must be aligned to a multiple of 8 bytes (ARM).
stackSize	The size of the memory area pointed to by *pStack*, in number of words of type *bspStackType_t*.
flags	Some flags : AA_FLAG_STACKCHECK — Enable stack monitoring for this task AA_FLAG_SUSPENDED — The task is created in the suspended state. You have to use *aaTaskResume()* to start its execution.

pTaskId A pointer to a variable that will contain the identifier on the task created when the function returns.

The created task is immediately active: if it has a higher priority than the current task, the current task will immediately be preempted by the new task.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument: priority or stack.
AA_EDEPLETED	There is no more job descriptor available.
AA_EMEMORY	Stack memory allocation error.

aaTaskDelete

```
aaError_t   aaTaskDelete      (aaTaskId_t     taskId)
```

Description TOC §↑

This function allows you to delete a task: release its resources, and place its descriptor in the queue of free task descriptors.

taskId The identifier of the task to complete. If the identifier is AA_SELFTASKID, then the calling task is deleted.

If a task exits the specified function when created by return, or has reached the end of the function then *aaTaskDelete (AA_SELFTASKID)* is implicitly called.

If a task to delete uses a static stack allocated by the user, then the *aaUserReleaseStack()* callback is called. This gives the user the opportunity to know that a memory block is free and manage it accordingly.

If a task is deleted while it has a mutex or semaphore, these remain in the state, which can cause unpredictable behavior of the application. The same thing can happen if the deleted task was waiting in a driver (state *aaIoWaitingState*).

If a task is destroyed by another task, the task stack and descriptor are released immediately. If the task destroys itself, this is not possible, and the releases will be done later by the idle task.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument: priority or stack.
AA_ESTATE	The task is in an unknown state that does not allow the completion.

aaTaskIsId

```
aaError_t   aaTaskIsId        (aaTaskId_t     taskId)
```

Description TOC §↑

This function checks that the task identifier provided is valid:

- The identifier corresponds to a task
- The task exists (has been created and has not been destroyed).

taskId The task identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaTaskGetBasePriority

```
aaError_t  aaTaskGetBasePriority  (aaTaskId_t taskId,
                                   uint8_t    * pBasePriority)
```

Description TOC §↑

This function allows you to know the basic priority of the task. The basic priority is the one that is specified when creating the task, and that is used when the priority inheritance mechanism is not active.

taskId	The identifier of the task that must be prioritized. If the identifier is AA_SELFTASKID, then this is the calling task.
pBasePriority	A pointer to a variable in which the value of the priority will be placed.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.

aaTaskGetRealPriority

```
aaError_t  aaTaskGetRealPriority  (aaTaskId_t taskId,
                                   uint8_t    * pRealPriority)
```

Description TOC §↑

This function allows you to know the current priority of the task. The current priority can be the basic priority or inherited priority.

taskId	The identifier of the task that must be prioritized. If the identifier is AA_SELFTASKID, then this is the calling task.
pRealPriority	A pointer to a variable in which the value of the priority will be placed.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.

aaTaskSetPriority

```
aaError_t    aaTaskSetPriority  (aaTaskId_t   taskId,
                                 uint8_t      newBasePriority)
```

Description TOC §↑

This function is used to change the basic priority of the task. The basic priority is the one that is specified when creating the task, and is used when the priority inheritance mechanism is not active.

tasked	The identifier of the task whose priority must be changed. If the identifier is AA_SELFTASKID, then this is the calling task.
newBasePriority	The new task priority between 1 and AA_PRIO_COUNT-1.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.

aaTaskSuspend

```
aaError_t     aaTaskSuspend  (aaTaskId_t      taskId)
```

Description TOC §↑

The specified task is placed in the suspended state: even if it is ready to execute, it does not. It cannot resume its activity until it has been reactivated by *aaTaskResume()*.

If a task is suspended while waiting for a resource (mutex, semaphore ...) it will not be suspended until after obtaining the resource, which then becomes unavailable for other tasks during the whole suspension of the task having got the resource.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.

aaTaskResume

```
aaError_t      aaTaskResume    (aaTaskId_t      taskId)
```

Description TOC §↑

The specified task is reactivated if it is in the suspended state.

Return value

AA_ENONE	No error
AA_EARG	Invalid argument

aaTaskDelay

```
void           aaTaskDelay     (uint32_t delay)
```

Description TOC §↑

The current task is put in wait state for the specified duration in number of system ticks. This causes the activation of the highest priority active task, if there is one.

The pending task does not consume CPUs.

Delay	The wait time in ticks, from 0 to 0xFFFFFFFE. The value AA_INFINITE specifies an infinite duration.

Return value

None

aaTaskWaikeUp

```
void          aaTaskWakeUp   (aaTaskId_t      taskId)
```

Description TOC §↑

Stops the waiting for a task that called *aaTaskDelay()*. The specified task is immediately placed in the ready state, and cannot know that it has been woken up earlier than expected.

This function cannot stop the waiting for a task waiting for an event: signal, semaphore, mutex, queue, etc.

Return value

None

aaTaskSelfId

```
aaTaskId_t    aaTaskSelfId   (void)
```

Description TOC §↑

Get the identifier of the calling task.

Called from an interrupt function, it returns the identifier of the interrupted task (the "current" task).

Return value

The identifier value of the calling task.

aaTaskYield

void aaTaskYield (void)

Description TOC §↑

Allows the current task to grant its execute right to another task, which has the same priority. The current task is placed at the end of the list of tasks ready with this priority.

If there is no other task with the same priority, the current task continues executing.

This function makes it possible to manage the execution time in a cooperative manner between tasks of the same priority.

Return value

None

aaTaskGetName

```
const char *  aaTaskGetName  (aaTaskId_t      taskId,
                              const char      ** ppName)
```

Description TOC §↑

Get a pointer to the name of the specified task.

Example, to get the name of the current task:

```
const char  * pStr ;
(void) aaTaskGetName (AA_SELFTASKID, & pStr) ;
```

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.

aaTaskCheckStack

```
aaError_t aaTaskCheckStack   (aaTaskId_t      taskId,
                              uint32_t        * pFreeSpace)
```

Description TOC §↑

Allows you to get the unused space in the stack of the specified task, in number of words of type *bspStackType_t*.

The AA_FLAG_STACKCHECK flag must have been specified when the task was created, which caused the task's stack to be initialized with a marker.

This function, whose execution time may be important, does not use a critical section, so the specified task must not be destroyed during the execution of this function.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument.
AA_ESTATE	AA_FLAG_STACKCHECK was not specified for this task.

aaTaskInfo

```
aaError_t   aaTaskInfo    (aaTaskInfo_t     * pInfo,
                           uint32_t         size,
                           uint32_t         * pReturnSize,
                           uint32_t         * pCpuTotal,
                           uint32_t         * pCriticalUsage,
                           uint32_t         flags)
```

Description TOC §↑

Provides information and statistics on tasks.

pInfo	Address of an array of structures that will be filled by the function
size	Number of elements in the *pInfo* structure. It is useful if this number is at least equal to the number of tasks created.
pReturnSize	Number of *pInfo* elements used by the function.
pCpuTotal	Address of a variable in which the time in micro seconds used by the tasks will be placed since the last call to *aaTaskStatClear()*.
pCriticalUsage	The address of a variable in which will be placed the time spent in the longest critical section since the last call to *aaTaskStatClear()*.
flags	not used.

The information structure is as follows:

```
typedef struct
{
    aaTaskId_t      taskId ;        // Id of this task
    aaTaskState_t   state ;
    uint8_t         priority ;      // Effective priority
    uint8_t         basePriority ;
    uint32_t        cpuUsage ;      // Count of CPU usage
    uint32_t        stackFree ;     // Count of unused words
                                    // in the task stack

} aaTaskInfo_t ;
```

The names of the task states are available in the table *aaTaskStateName[]*. Example:

```
aaTaskInfo_t    info ;
aaPrintf ("%u %s\n", info.state, aaTaskStateName[info.state]);
```

This function uses a critical section so that the status of all tasks is consistent. If the number of tasks is important, the duration of the critical section may be important.

Return value

AA_ENONE	No error.

aaTaskStatClear

```
void      aaTaskStatClear    (void)
```

Description

TOC §↑

Initializes to 0 the statistics of the tasks that can be obtained with *aaTaskInfo()*.

Return value

None.

4.3 Mutex

aaMutexCreate

```
aaError_t  aaMutexCreate     (aaMutexId_t     * pMutexId)
```

Description TOC §↑

Create a mutex. This mutex provides exclusive access to a resource such as a device or data structure.

It has special characteristics:

- It can be acquired recursively: the same task can acquire several times the same mutex, and then release it as many times as it has acquired.
- The mutex uses a task priority inheritance algorithm, to avoid the priority inversion phenomenon, which causes a low priority task to prevent another higher priority task from executing.
- If a task is destroyed while holding a mutex, it will not be rendered, and the resource whose access is protected remains locked.
- If a task is suspended while holding a mutex, the mutex remains acquired by the task.

pMutexId Pointer to a variable that will receive the mutex identifier.

Return value

AA_ENONE	No error, the mutex is created.
AA_EDEPLETED	There is no more mutex descriptor available.
AA_ENOTALLOWED	Not allowed from an interrupt function.

aaMutexDelete

```
aaError_t   aaMutexDelete    (aaMutexId_t     mutexId)
```

Description TOC §↑

Delete a mutex, and place its descriptor in the list of free mutex descriptors.

The mutex must be free to be deleted.

Return value

AA_ENONE	No error, the mutex is deleted.
AA_ESTATE	The mutex is not in a state of being destroyed (acquired by a task).
AA_ENOTALLOWED	Not allowed from an interrupt function.

AA_EARG	Invalid mutex identifier.

aaMutexIsId

```
aaError_t   aaMutexIsId       (aaMutexId_t    mutexId)
```

Description TOC §↑

This function checks that the mutex identifier provided is valid:

- The identifier corresponds to a mutex
- The mutex exists (has been created and has not been destroyed).

`mutexId` The mutex identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaMutexTake

```
aaError_t   aaMutexTake      (aaMutexId_t     mutexId,
                              uint32_t        timeOut)
```

Description TOC §↑

This function is used to acquire the mutex. If the mutex is already acquired by another task, the calling task is suspended until the mutex is acquired or the timeout expires.

mutexId	The identifier of the mutex
timeout	The value of the system tick timeout, from 0 (no wait) to 0xFFFFFFFE or AA_INFINITE.

This function cannot be used by an interrupt handler.

Return value

AA_ENONE	No error, the mutex is acquired
AA_EARG	Invalid argument.
AA_ENOTALLOWED	Unauthorized operation, e.g. from an interruption.
AA_EWOULDBLOCK	Mutex not acquired, specified timeout equal to 0.
AA_ETIMEOUT	Mutex not acquired, return with timeout.

AA_EFAIL	Mutex counter overflow.

aaMutexTryVTake

```
aaError_t   aaMutexTryTake    (aaMutexId_t    mutexId)
```

Description TOC §↑

Try to acquire the mutex, with a timeout of 0.

This is a macro equivalent to: *aaMutexTake (mutexId, 0)*.

This macro cannot be used by an interrupt handler.

Return value

The same as *aaMutexTake()*.

aaMutexGive

```
aaError_t   aaMutexGive       (aaMutexId_t    mutexId)
```

Description TOC §↑

This function allows to releases the specified mutex. If the recursion counter drops to 0, the mutex is actually released, and if another task is waiting for that mutex it acquire the mutex.

If another task acquire the released mutex and this task has a higher priority than the base priority of the calling task, the other task is immediately activated.

This function cannot be used by an interrupt handler.

Return value

AA_ENONE	No error, the mutex is acquired
AA_EARG	Invalid argument.
AA_ENOTALLOWED	Unauthorized operation, e.g. from an interrupt routine.
AA_ESTATE	The mutex is not able to be released: it is acquired by another task, already released.

4.4 Semaphore

aaSemCreate

```
aaError_t   aaSemCreate       (int32_t        count,
                              aaSemId_t       * pSemId)
```

Description TOC §↑

Allows you to create a counter semaphore, and initialize its value. A semaphore always handles the pending tasks in descending order of priority.

If a task is destroyed while holding a semaphore, the semaphore will not be rendered, and the resource whose access is protected remains locked.

If a task is suspended while holding a semaphore, the semaphore remains acquired by the task.

count	The initial value of the semaphore counter (from -32768 to +32767).
pSemId	A pointer to the variable that will contain the identifier of the created semaphore.

Return value

AA_ENONE	No error, the semaphore is created.
AA_EDEPLETED	There is no more semaphore descriptor available.

aaSemDelete

```
aaError_t   aaSemDelete       (aaSemId_t      semId)
```

Description TOC §↑

Lets you delete a semaphore, and release its resources.

If any tasks are pending for the semaphore, they are all released by a call to *aaSemFlush()*.

semId	The identifier of the semaphore to delete.

Return value

AA_ENONE	No error, the semaphore is destroyed.
AA_EARG	The identifier is not a valid identifier.

aaSemIsId

```
aaError_t   aaSemIsId         (aaSemId_t      semId)
```

Description TOC §↑

This function checks that the semaphore identifier provided is valid:

- The identifier corresponds to a semaphore
- The semaphore exists (has been created and has not been destroyed).

`semId` The semaphore identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaSemTake

```
aaError_t   aaSemTake         (aaSemId_t      semId,
                               uint32_t       timeOut)
```

Description TOC §↑

Acquires the semaphore whose identifier is specified. If the semaphore counter is less than or equal to 0 the task is put on blocked state until the counter becomes positive and no other higher priority task is waiting for the semaphore.

This function cannot be used by an ISR.

semId The identifier of the semaphore to acquire.

Return value

AA_ENONE	No error, the semaphore is acquired.
AA_EARG	The identifier is not a valid identifier.
AA_ENOTALLOWED	Call from an interrupt service routine
AA_EWOULDBLOCK	Not acquired and timeout 0
AA_EFLUSH	Not acquired, released by *aaSemFlush()*.
AA_ETIMEOUT	Not acquired, timeout expired.

aaSemTryTake

```
aaError_t   aaSemTryTake      (aaSemId_t      semId)
```

Description TOC §↑

Try to get the semaphore with a timeout of 0.

It's a macro equivalent to: *aaSemTake (semId, 0)*.

This macro cannot be used by an ISR.

Return value

The same as *aaSemTake()*.

aaSemGive

```
aaError_t   aaSemGive         (aaSemId_t      semId)
```

Description TOC §↑

Increments the counter of a semaphore, and optionally assigns it to a pending task if it is positive.

The function is allowed from an ISR.

Return value

AA_ENONE	No error, the semaphore is released.
AA_EARG	The identifier is not a valid identifier.

aaSemFlush

```
aaError_t   aaSemFlush        (aaSemId_t      semId)
```

Description TOC §↑

Release all pending tasks atomically. All tasks are released before any of them can be activated.

The state of the semaphore is unchanged.

This is useful for performing some sort of broadcast to synchronize tasks.

The function is allowed from an ISR.

Return value

AA_ENONE	No error, all pending tasks are released.
AA_EARG	The identifier is not a valid identifier.

aaSemReset

```
aaError_t   aaSemReset      (aaSemId_t      semId,
                             int16_t        count)
```

Description

TOC §↑

Initialize the counter of a semaphore. This is only allowed if there are no tasks waiting to acquire the semaphore.

The function is allowed from an ISR.

Return value

AA_ENONE	No error, the semaphore is initialized.
AA_EARG	The identifier is not a valid identifier.
AA_ESTATE	Not done, there are pending tasks.

4.5 Inter-task signals

aaSignalWait

```
aaError_t   aaSignalWait     (aaSignal_t      sigsIn,
                              aaSignal_t      * pSigsOut,
                              uint32_t        mode,
                              uint32_t        timeOut)
```

Description TOC §↑

Allow a task to wait for one or more of its signals to be reported. If the signals are already present during the call, the task returns immediately. Otherwise it is blocked until the signals are reported or the timeout is complete.

sigsIn	A bit mask that tells what signals are expected.
pSigsOut	A pointer to a variable that contains on the return of the function a bit mask that corresponds to the *sigsIn* signals that caused the return. This bit mask can be different from *sigsIn* if the mode AA_SIGNAL_OR was used. *pSigsOut* can be NULL if the output signal mask is useless.
mode	The signal processing to be used to cause the return of the function.
timeout	The timeout in system ticks.

The two available modes are:

AA_SIGNAL_AND	The task waits until all requested signals are reported.
AA_SIGNAL_OR	The task waits until at least one of the requested signals is reported.

This function is not allowed from an ISR.

Return value

AA_ENONE	Expected signals are positioned.
AA_ENOTALLOWED	Not allowed from an ISR.
AA_EWOULDBLOCK	The expected signals are not present, and the timeout is 0.
AA_ETIMEOUT	Signals have not been set before the timeout expires.

aaSignalSend

```
aaError_t   aaSignalSend     (aaTaskId_t      taskId,
                              aaSignal_t      sigs)
```

Description TOC §↑

Set the *sigs* signals of the specified task.

If the specified task is waiting for these signals, it is immediately activated.

This function can be used by an interrupt function.

tasked	The identifier of the reportable task.
sigs	The mask of the signals to be positioned.

Return value

AA_ENONE	No error.
AA_EARG	The task identifier is not a valid identifier.

aaSignalPulse

```
aaError_t   aaSignalPulse    (aaTaskId_t      taskId,
                              aaSignal_t      sigs)
```

Description TOC §↑

Set the signals *sigs* and reset signals immediately. In other words, this function is equivalent to *aaSignalSend()*, but the *sigs* signals are not stored.

If the task is waiting and the combination of the signals already present and *sigs* is what the task is waiting for it is awake, and the expected signals are set to 0.

In all cases the *sigs* signals are set to 0.

This function can be used by an interrupt function.

taskId	The ID of the task waiting for the signals.
sigs	The mask of signals to report.

Return value

AA_ENONE	No error.
AA_EARG	The ID is not a valid ID.

aaSignalClear

```
aaError_t aaSignalClear   (aaTaskId_t taskId, aaSignal_t sigs)
```

Description

TOC §↑

Allow you to set the *sigs* signals of the task *taskNd* to 0.

AA_SGNAL_ALL can be used as a *sigs* value to set all signals to 0.

Return value

None.

4.6 Dynamic memory allocation

aaMalloc

```
void *    aaMalloc            (uint32_t       size)
```

Description TOC §↑

Allocate a block of dynamic memory. The allocation algorithm depends on the kernel configuration.

size The size of the block in bytes.

Forbidden for an ISR.

Return value

A pointer to the allocated block on success, NULL on failure.

aaCalloc

```
void *    aaCalloc            (uint32_t       nmemb,
                               uint32_t       size)
```

Description TOC §↑

Allow you to allocate a dynamic memory block for an array of *nmemb* elements, each of size *size*. The size of the allocated block is *nmemb* * *size* bytes.

The allocation algorithm depends on the kernel configuration.

nmemb The number of elements to allocate.

size The size of an element in bytes.

Forbidden for an ISR.

Return value

A pointer to the allocated block on success, NULL on failure.

aaRealloc

```
void *    aaRealloc          (void            * pMem,
                              uint32_t        size)
```

Description TOC §↑

Change the size of a previously allocated dynamic memory block.

Forbidden for an ISR.

Return value

A pointer to the allocated block on success, NULL on failure.

aaFree

```
void      aaFree             (void            * pMem)
```

Description TOC §↑

Release a previously allocated block of dynamic memory.

Forbidden for an ISR.

Return value

None

aaTryFree

```
aaError_t aaTryFree          (void            * pMem)
```

Description TOC §↑

Attempt to free a previously allocated block of dynamic memory. If this is not possible the calling task is not blocked.

Release is not possible if the dynamic memory allocation protection mutex is already acquired by another task.

Return value

AA_ENONE	No error.
AA_ENOTALLOWED	Unauthorized operation, from an ISR for example.
AA_EWOULDBLOCK	Can't free.

aaMemPoolCheck

```
uint32_t  aaMemPoolCheck      (uint32_t       bVerbose)
```

Description

TOC §↑

Check the integrity of the links between the dynamic memory blocks. This can detect writings beyond block size.

Uses the verification algorithm associated with dynamic memory allocation, for example *tlsfCheck()*.

Forbidden for an ISR.

Return value

Those of the verification algorithm associated with the dynamic memory allocation.

4.7 TLSF Memory Partitioning

tlsfInit

```
hTlsf_t   tlsfInit          (void           * pMem,
                             uint32_t       size)
```

Description TOC §↑

Initializes a memory partition managed by the TLSF algorithm

pMem The address of the partition.

size The size of the partition in bytes.

Return value

If successful: The partition handle to use with other TLSF partition management functions.

In case of failure: NULL.

tlsfMalloc

```
void *    tlsfMalloc        (hTlsf_t        hTlsf,
                             uint32_t       size)
```

Description TOC §↑

Allocate a memory block.

hTlsf the handle of the partition.

size The size of the block to allocate in bytes.

Return value

If successful: the address of the allocated block.

In case of failure: NULL.

tlsfCalloc

```
void *    tlsfCalloc          (hTlsf_t          hTlsf,
                               uint32_t         nmemb,
                               uint32_t         size)
```

Description TOC §↑

Allow you to allocate a memory block of size: *nmemb* * *size*.

hTlsf	the handle of the partition.
nmemb	The number of elements.
size	The size of an element.

Return value

If successful: the address of the allocated block.

In case of failure: NULL.

tlsfFree

```
void      tlsfFree            (hTlsf_t          hTlsf,
                               void             * ptr)
```

Description TOC §↑

Allow you to return a memory block to the partition. This block must have been allocated with *tlsfMalloc()*.

hTlsf	The handle of the partition.
ptr	The address of the block to release.

Return value

None.

tlsfRealloc

```
void *    tlsfRealloc        (hTlsf_t         hTlsf,
                              void             * ptr,
                              uint32_t         size) ;
```

Description TOC §↑

Change the size of a block allocated by *tlsfMalloc()*. Or *tlsfRealloc()*.

hTlsf	The handle of the partition.
ptr	The address of the block.
size	The new size in bytes.

Return value

If successful: the address of the allocated block.

In case of failure: NULL.

tlsfCheck

```
aaError_t tlsfCheck          (hTlsf_t         hTlsf,
                              uint32_t         bVerbose)
```

Description TOC §↑

Allow verification of the integrity of the memory partition.

hTlsf the	The handle of the partition.
bVerbose	If this parameter is 0, the Return value is used to evaluate the result of the test. If the parameter is 1 then additional information is sent with *aaPrintf()*.

Return value

AA_ENONE	If successful.
AA_FAIL	In case of failure.

4.8 Block Memory Partition

aaInitMallocBloc

```
aaError_t aaInitMallocBloc   (uint8_t           * pBloc,
                              uint32_t          size,
                              aaMallocBlocId_t  * pId) ;
```

Description TOC §↑

Allow you to initialize a block allocation partition.

pBloc	The address of the partition.
size	The size of the partition in bytes.
pId	A pointer to a variable that contains the handle of the partition when the function returns.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument, the partition is not created.

aaMallocBloc

```
aaError_t aaMallocBloc       (aaMallocBlocId_t  blockId,
                              uint32_t          size,
                              void              ** ppBloc) ;
```

Description TOC §↑

Allocate a block in the block partition.

blockId	The handle of the block partition.
size	The size of the block to allocate in bytes.
ppBloc	A pointer to a variable that contains the address of the allocated block on return of the function.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument, the block is not allocated.

aaMallocBlocFreeSize

```
aaError_t aaMallocBlocFreeSize  (aaMallocBlocId_t  blockId,
                                 uint32_t          * pSize) ;
```

Description

TOC §↑

Lets you know the free space in the partition.

blockId	The handle of the partition.
pSize	A pointer to a variable that on returning the function contains the number of free bytes in the partition.

Return value

AA_ENONE	No error.
AA_EARG	Invalid argument the size is not filled.

4.9 Log and console

aaLogMes

```
void      aaLogMes            (const char      * fmt,
                               uintptr_t       a1,
                               uintptr_t       a2,
                               uintptr_t       a3,
                               uintptr_t       a4,
                               uintptr_t       a5)
```

Description TOC §↑

Send a trace request to the *tLogM* task.

The trace is not performed immediately but placed in a message queue and later processed by the task *tLogM*, the function can be called by an interrupt function.

The arguments must not be pointers to volatile entities (allocated on the stack of the calling task, for example).

Return value

None.

aaLogMesSetPutChar

```
void      aaLogMesSetPutChar (void (* pPutChar) (char cc))
```

Description TOC §↑

Specify the function to use to transmit each character of trace messages. This allows choosing to send the messages to the console, a UART or SWO for example.

Return value

None.

aaPrintf

```
uint32_t  aaPrintf            (const char      * fmt,
                                 ...)
```

Description TOC §↑

This function has the same syntax as the *printf()* function of the 'C' standard. It allows for simplified message formatting, to occupy little space and be fast.

The supported formats are :

c	A character
d i	Decimal number signed
u	Unsigned decimal number
x X	X Hexadecimal number
o	Octal number
b	Binary number
f	Float number (with restrictions)
s	Character string
p	Pointer to void (output similar to %08X)
f	Float or double

g format is not supported.

Optional fields can appear between % and the format character, in the following order :

« - »	Left aligned.
« 0 »	Adding 0 in front to complete the length if the "width" field is specified.
« width »	Minimum number of characters to generate, can be replaced by *.
« l »	Size specification, accepted but ignored.

The %f format was introduced so as not to have to use the standard library which uses a lot of FLASH and RAM space. However this display must be considered as indicative and does not support all the functionalities of the standard library (not NaN for example, and lower precision).

The %f format is enabled and configured in aacfg.h by the constants AA_WITH_FLOAT_PRINT, AA_FLOAT_T and AA_FLOAT_SEP.

Example: with the format %7.2f the value -3.128 is displayed as '-3.13'

Return value

The number of characters issued.

aaPrintfEx

```
uint32_t  aaPrintfEx          (void (* fnPutc) (char),
                               const char      * fmt, ...)
```

Description

TOC §↑

Allows as *aaPrintf()* to format a message, but specifying the function to use to emit characters.

Return value

As *aaPrintf()*.

aaSnPrintf

```
uint32_t  aaSnPrintf          (char            * pBuffer,
                               uint32_t        size
                               const char      * fmt,
                               ...)
```

Description

TOC §↑

Allows as *aaPrintf()* to format a message, but by copying it to the *pBuffer* buffer. It is equivalent to the *snprintf()* function of C99, but without dynamic memory allocation, and a moderate use of the stack.

pBuffer	The string that receives the characters
size	The maximum size to use in *pBuffer*, including the final NUL character
fmt	The format string
...	The arguments used by the format

The string contained in *pBuffer* is always terminated with '\ 0', even if there has been an overflow.

Return value

The count of characters that are generated, not counting the final '\ 0', assuming that size is sufficient. If this number is greater than or equal to size, there has been truncation.

aaGets

```
uint32_t  aaGets              (char            * pBuffer,
                               uint32_t        size)
```

Description

TOC §↑

Reads at most size - 1 character from the console and places them in the buffer pointed by *pBuffer.* Reading stops after a carriage return, which is not placed in the buffer. A null character is placed at the end of the line.

Some special characters and ANSI sequences are handled: backspaces, arrows, del, home, end.

pBuffer	The string that receives the characters
size	The maximum size to use in pBuffer, including the final draw

Return value

The count of characters actually returned, not counted the final draw.

aaSetStdOut

```
void      aaSetStdOut         (void (* fnPutc) (char))
```

Description

TOC §↑

Specify the function to use by *aaPrintf()* and *aaPutChar()* to emit a character.

In general, the function *fnPut()* emits a character to the console.

Return value

None.

aaSetStdIn

```
void      aaSetStdIn          (char (* fnGetc) (void))
```

Description

TOC §↑

Specify the function to use by *aaGetChar()* to acquire a character.

In general, the *fnGetc ()* function is used to acquire a character from the console.

Return value

None.

aaPutChar

```
void      aaPutChar          (char cc)
```

Description

TOC §↑

Macro that allows to send a character with the function configured with *aaSetStdOut()*. Using this macro allows you to write applications independent of the device you are using.

Return value

None.

aaGetChar

```
char      aaGetChar          (void)
```

Description

TOC §↑

Macro that allows you to acquire a character with the function configured with *aaSetStdIn()*. Using this macro allows you to write applications independent of the device you are using.

Return value

None.

4.10 Software Timers

aaTimerCreate

```
aaError_t aaTimerCreate      (aaTimerId_t     * pTimerId)
```

Description TOC §↑

Create a timer and get its identifier.

pTimerId	A pointer to the variable that will contain the identifier of the created timer.

Return value

AA_ENONE	No error, the timer is created.
AA_EARG	*pTimerId* is not valid.
AA_EDEPLETED	There is no more timer descriptor available.

aaTimerDelete

```
aaError_t aaTimerDelete      (aaTimerId_t     timerId)
```

Description TOC §↑

Disables the timer, and places it in the list of free timers. It cannot be used anymore.

Return value

AA_ENONE	No error, the timer is destroyed.
AA_EARG	*timerId* is not a valid flag.
AA_ESTATE	This timer is already destroyed.

aaTimerIsId

```
aaError_t   aaTimerIsId       (aaTimerId_t     timerId)
```

Description TOC §↑

This function checks that the timer identifier provided is valid:

- The identifier corresponds to a timer
- The timer exists (has been created and has not been destroyed).

`timerId` The timer identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaTimerSet

```
aaError_t aaTimerSet        (aaTimerId_t      timerId,
                             aaTimerCallback  callback,
                             uintptr_t        arg,
                             uint32_t         timeout)
```

Description TOC §↑

Configure the software timer.

timerId	The identifier of the timer to use.
callback	A pointer to the function that will be called when the timer expires. This pointer must not be NULL. The prototype of the function is: *typedef uint32_t (* aaTimerCallback) (uintptr_t arg);*
arg	The argument passed to the callback.
timeout	The timer delay in system tick. This time must be between 1 and AA_INFINITE-1.

Return value

AA_ENONE	No error, the timer is configured.
AA_EARG	An argument is invalid.
AA_ESTATE	This timer is not usable (not created ...).

aaTimerStart

```
aaError_t aaTimerStart       (aaTimerId_t     timerId)
```

Description TOC §↑

Start the timer. The timer must have been configured beforehand with *aaTimerSet()*.

If the timer is already started, it is restarted with its initial duration.

Return value

AA_ENONE	No error, the timer is started.
AA_EARG	*TimerId* is invalid.
AA_ESTATE	This timer is not usable (not created ...).

aaTimerStop

```
aaError_t aaTimerStop        (aaTimerId_t     timerId)
```

Description TOC §↑

Stop the timer before the timeout expires. It is not a mistake to stop a timer already stopped.

Return value

AA_ENONE	No error, the timer is stopped.
AA_EARG	*TimerId* is invalid.
AA_ESTATE	This timer is not usable (not created ...).

4.11 Message queues

aaQueueCreate

```
aaError_t aaQueueCreate      (aaQueueId_t     * pQueueId,
                              uint32_t        msgSize,
                              uint32_t        msgCount,
                              uint8_t         * pBuffer,
                              uint32_t        flags)
```

Description TOC §↑

Create a message queue.

pQueueId	A pointer to the variable that will contain the identifier of the queue created.
msgSize	The maximum size of messages in bytes between 1 and 65535.
msgCount	The maximum number of messages that the queue can hold, between 1 and 65535.
pBuffer	If the message buffer is provided by the user, pBuffer is a pointer to a space of at least *msgSize * msgCount* bytes. If the buffer needs to be allocated by the kernel, then pBuffer is NULL.
Flags	A ccombination of indicators:

AA_QUEUE_PRIORITY	Pending tasks are processed by priority order (exclusive of AA_QUEUE_FIFO).
AA_QUEUE_FIFO	Pending tasks are processed in FIFO order (exclusive of AA_QUEUE_PRIORITY).
AA_QUEUE_POINTER	Messages are pointers. In this case the value of the *msgSize* parameter is ignored. The size of the messages is implicitly the size of a pointer.

For the kernel to allocate and free the buffer, dynamic memory allocation must be allowed.

Return value

AA_ENONE	No error, the queue is created.
AA_EARG	*pQueueId* is not valid.
AA_EMEMORY	The message buffer could not be allocated.
AA_EDEPLETED	There are no more queue descriptors available.

aaQueueDelete

```
aaError_t aaQueueDelete      (aaQueueId_t     queueId)
```

Description TOC §↑

Allow to delete the queue and place it in the list of free queues. It cannot be used anymore.

The message buffer is released if it has been allocated by the kernel, otherwise the user must take care of it.

Return value

AA_ENONE	No error.
AA_EARG	*pQueueId* is not valid.

aaQueueIsId

```
aaError_t   aaQueueId         (aaQueueId_t    queueId)
```

Description TOC §↑

This function checks that the provided queue identifier is valid:

- The identifier corresponds to a queue
- The queue exists (has been created and has not been destroyed).

`queueId` The queue identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaQueueGive

```
aaError_t aaQueueGive        (aaQueueId_t     queueId,
                              void            * pData,
                              uint32_t        size,
                              uint32_t        timeout)
```

Description TOC §↑

Add a message to the queue.

pQueueId The identifier of the queue.

pData	The address of the message to be copied to the queue.
size	The size of the message. If size is 0, then the size specified when creating the queue is used.
timeout	The timeout if the queue is full. If the timeout is 0, the function returns immediately.

The value of the *pData* parameter of *aaQueueGive()* depends on the use of AA_QUEUE_POINTER when the pool is created.

Without AA_QUEUE_POINTER	pData **is a pointer to** the information to put in the queue,
With AA_QUEUE_POINTER	pData **is** the information to put in the queue (therefore it is not a pointer to a pointer).

If the timeout parameter is 0, this is equivalent to using a function that could be called "*aaQueueTryGive()*". If the message could not be placed in the queue, the function returns immediately with the value AA_EWOULDBLOCK.

If this function is called by an interrupt handler, the timeout is ignored and considered to be 0.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.
AA_EWOULDBLOCK	The queue is full and the timeout is 0, or the caller is an interrupt.
AA_ETIMEOUT	The queue is full and the timeout has expired.

aaQueueTake

```
aaError_t aaQueueTake        (aaQueueId_t     queueId,
                              void            * pData,
                              uint32_t        size,
                              uint32_t        timeout)
```

Description TOC §↑

Extract a message from the queue.

pQueueId	The identifier of the queue.
pData	The address where to copy the message extracted from the queue.
size	The size of the message to copy. If size is 0, then the size specified when creating the queue is used. If AA_QUEUE_POINTER is used then size is ignored.

timeout	The timeout if the queue is empty. If the timeout is 0, the function returns immediately.

If the timeout parameter is 0, this is equivalent to using a function that could be called "*aaQueueTryTake()*".

If this function is called by an interrupt handler, the timeout is ignored and considered at 0.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.
AA_EWOULDBLOCK	The queue is empty and the timeout is 0, or the caller is an interrupt.
AA_ETIMEOUT	The queue is empty and the timeout has expired.

aaQueuePeek

```
aaError_t aaQueuePeek        (aaQueueId_t     queueId,
                              void            ** ppData,
                              uint32_t        timeout)
```

Description TOC §↑

Get the address of the next message in the queue without removing the message from the queue. This allows the message to be inspected before removing it from the queue.

This function should be used with care if there is more than one reader in the queue: while one task is inspecting a message, another task can remove it from the queue.

pQueueId	The identifier of the queue.
pData	The address where to copy the message address.
timeout	The timeout if the queue is full. If the timeout is 0, the function returns immediately.

If this function is called by an interrupt, the timeout is ignored and considered at 0.

Return value

AA_ENONE	No error
AA_EARG	An argument is not valid.
AA_EWOULDBLOCK	The queue is full and the timeout is 0, or the caller is an interrupt.

AA_ETIMEOUT	The queue is full and the timeout has expired.

aaQueuePurge

```
aaError_t aaQueuePurge      (aaQueueId_t     queueId)
```

Description TOC §↑

Removes the first message from the queue without reading it.

This can be used in conjunction with *aaQueuePeek()*, if it is not necessary to read the message.

pQueueId The identifier of the queue.

Return value

AA_ENONE	No error
AA_EARG	An argument is not valid.

aaQueueGetCount

```
aaError_t aaQueueGetCount    (aaQueueId_t     queueId,
                              uint32_t        * pCount)
```

Description TOC §↑

Copy the count of messages present in the queue in the variable pointed to by *pCount*.

Return value

AA_ENONE	No error.
AA_EARG	An argument is invalid.

4.12 Buffer Pool

aaBufferPoolCreate

```
aaError_t aaBufferPoolCreate (aaBufPoolId_t   * pPoolId,
                              uint32_t        bufCount,
                              uint32_t        bufSize,
                              void            * pBuffer)
```

Description TOC §↑

Create a descriptor for a buffer pool

pPoold	A pointer to the variable that will contain the identifier of the created pool.
bufCount	The number of buffers in the pool.
bufSize	The size of a buffer in bytes.
pBuffer	If the buffer pool is provided by the user, pBuffer is a pointer to a space of at least bufSize * bufCount bytes. If the buffer needs to be allocated by the kernel, then pBuffer is NULL.

For the kernel to allocate and free the pool, dynamic memory allocation must be allowed.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.
AA_EMEMORY	The buffer pool could not be allocated.
AA_EDEPLETED	There is no more pool descriptor available.

aaBufferPoolDelete

```
aaError_t aaBufferPoolDelete (aaBufPoolId_t   bufPoolId,
                              uint32_t        bForce)
```

Description TOC §↑

Delete a pool of buffers, and places its descriptor in the list of free descriptors.

bufPoolId	The pool identifier
bForce	If *bForce* is 0, the pool is destroyed only if all buffers in the pool are free (returned to the pool). If *bForce* is 1, the buffer is destroyed unconditionally.

The buffer pool is released if it has been allocated by the kernel, otherwise the user must take care of it.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.

aaBufferPoolIsId

```
aaError_t   aaBufferPoolIsId  (aaBufPoolId_t  bufPoolId)
```

Description TOC §↑

This function checks that the provided pool identifier is valid:

- The identifier corresponds to a pool
- The pool exists (has been created and has not been destroyed).

`bufPoolId`	The pool identifier to check.

Return value

AA_ENONE	The identifier is valid.
AA_EFAIL	The identifier is not valid.

aaBufferPoolTake

```
aaError_t aaBufferPoolTake   (aaBufPoolId_t   bufPoolId,
                              void            ** ppBuffer)
```

Description TOC §↑

Get a buffer from the pool.

bufPoolId	The pool identifier
ppBuffer	A pointer to a pointer that contains the buffer address at the return of the function.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.
AA_EDEPLETED	There is no buffer available.

aaBufferPoolGive

```
aaError_t aaBufferPoolGive   (aaBufPoolId_t   bufPoolId,
                              void            * pBuffer)
```

Description TOC §↑

Return the buffer to the pool.

bufPoolId	The pool identifier
pBuffer	The address of the buffer to return to the pool.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.

aaBufferPoolGetCount

```
aaError_t aaBufferPoolGetCount (aaBufPoolId_t bufPoolId,
                                uint32_t      * pCount)
```

Description TOC §↑

Get the number of buffers available from the pool.

bufPoolId	The pool identifier.
pCount	The address of a variable that will contain the count of available buffers.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.

aaBufferPoolReset

```
aaError_t aaBufferPoolReset  (aaBufPoolId_t   bufPoolId)
```

Description TOC §↑

Allows reset the pool to its state when it was created.

Return value

AA_ENONE	No error.
AA_EARG	An argument is not valid.

4.13 User Functions

This chapter lists the application functions known by the kernel.

userInitTask

```
void      userInitTask       (uintptr_t arg)
```

Description TOC §↑

This function is called by kernel initialization and this is the first task of the application.

When calling this function the kernel is completely started, all resources are available.

The application must define this function.

Return value

None

aaUserReleaseStack

```
aaError_t aaUserReleaseStack   (uint8_t      * pStack,
                                uint32_t     size)
```

Description TOC §↑

If a task uses a static stack allocated by the user, then during the destruction of this task the *aaUserReleaseStack()* callback is called. This gives the user the opportunity to know that a memory block is free and manage it accordingly.

pStack A pointer to the memory block to release.

Size The size of the memory block.

If the memory block is released, the function must return AA_ENONE.

If the memory block is not released, the function must return AA_EFAIL. The callback will be recalled later for a new release attempt.

A weak version of this function is defined by the kernel, it always returns AA_ENONE. If the application does not have the use of this function it does not need to define it.

Return value

AA_ENONE The memory block is released.

AA_EFAIL The memory block is not released, try again.

aaUserNotify

```
void      aaUserNotify          (uint32_t     event,
                                 uintptr_t    arg)
```

Description

TOC §↑

This function allows the kernel to warn the user of certain events.

These events are identified by the value of the event parameter:

AA_NOTIFY_STACKOVFL	The stack of the task overflowed. arg is the task identifier.
AA_NOTIFY_STACKTHR	The stack monitoring threshold has been reached. *arg* is the task identifier.

In order for these events to be detected and transmitted to the user, the task must be created with the AA_FLAG_STACKCHECK flag.

These events are only notified once.

A weak version of this function is defined by the kernel. If the application does not have the use of this function it does not need to define it.

Return value

None.

4.14 Board Support Package

bspGetTickRate

```
uint32_t  bspGetTickRate     (void)
```

Description TOC §↑

Lets you know the frequency of the system clock (tick).

Return value

The frequency of the system clock in hertz.

bspSetTickRate

```
aaError_t bspSetTickRate      (uint32_t       tickHz)
```

Description TOC §↑

Allow you to specify the frequency of the system clock (tick). Cannot be used if stretched tick mode is chosen.

tickHz	The frequency of the system clock in hertz.

Return value

AA_ENONE	The system clock is configured.
AA_FAIL	The system clock is not configured.

bspGetSysClock

```
uint32_t  bspGetSysClock     (void)
```

Description TOC §↑

Lets you know the clock frequency of the processor.

Return value

The clock frequency of the processor

bspResetHardware

```
void      bspResetHardware   (void)
```

Description

TOC §↑

Allows a software reset of the processor.

Return value

None.

bspOutput

```
void      bspOutput          (uint32_t num,
                              uint32_t state)
```

Description

TOC §↑

Used to set the status of a GPIO output configured by the BSP.

In general, the LEDs of an evaluation board are defined by the BSP, and constants are used to access these outputs.

num	The GPIO number that can be defined by the BSP, for example BSP_LED0.
state	The state to assign to the GPIO output: 0 or 1.

This function is declared "inline" and if the parameters are constants the generated code is reduced to a single assembly instruction, which makes this function very little intrusive.

Return value

None.

bspInput

```
uint32_t  bspInput           (uint32_t num)
```

Description

TOC §↑

Read the status of a GPIO input configured by the BSP.

In general the buttons of an evaluation board are defined by the BSP, and constants are used to access these inputs.

num	The GPIO number that can be set by the BSP, for example BSP_BUTTON0.

Return value

The value of the GPIO input: 0 or 1.

bspMainStackCheck

```
uint32_t  bspMainStackCheck  (void)
```

Description

TOC §↑

Provides information about system stack usage.

Return value

Returns the number of unused words in the system stack.

The "time stamp" functions allows to measure periods of time using the processor cycle counter. The counter uses 32 bits, and the resolution corresponds to the frequency of the system clock.

For example, if the system clock is 168 MHz, the counter resolution is 5.95 ns. Under these conditions, the 32-bit counter loops back after about 25.5 seconds, which corresponds to the maximum measurable time period, according to the formula: *Tmax = 2^{32} / bspGetSysClock()*.

CPU Frequency	Maximum measurable time
64 MHz	67.1 s
168 MHz	25.5 s
400 MHz	10.7 s

High resolution measurements can be made using the *bspTsGet()* and *bspRawDelta()* functions.

It is sometimes necessary to accumulate durations for a time longer than that allowed by the 32 bits and the resolution of the cycle counter. For this we can use *bspTsDelta()* which provides a duration in µs. It is thus possible to accumulate durations for about 1h: 11mn. Pay attention to losses due to rounding accumulation.

Warning: each delta remains subject to the loopback constraint of the cycle counter, and must therefore be less than this loopback time.

Remark: An MCU with Cortex-M0 or Cortex M0+ core does not have a cycle counter. For these MCU the counter is implemented using a timer. The timer and its resolution must be specified in the bsp.h file. Most timers have a 16 bits counter, so the resolution is lower than with the cycle counter, to get a useful measurement period.

bspTsGet

```
uint32_t  bspTsGet          (void)
```

Description

TOC §↑

Return the current value of the processor cycle counter. The resolution is that of the frequency of the system clock, which can be obtained with *bspGetSysClock()*.

This function is declared “inline”.

Return value

The current value of the cycle counter.

bspRawTsDelta

```
uint32_t  bspRawTsDelta      (uint32_t        * pTs)
```

Description TOC §↑

Acquires the current value of the cycle counter and subtracts the value pointed to by *pTs*, the result of the subtraction is the value returned. It corresponds to the number of cycles elapsed between the previous call to *bspTsGet()* or *bspTsDelta()* which provided the value pointed to by *pTs*, and the instant of the call of this function.

After the calculation, the current value of the counter is placed in the variable pointed by *pTs*.

This function is declared "inline".

Return value

The difference between the current value of the counter and the value pointed by *pTs*.

bspTsDelta

```
uint32_t  bspTsDelta         (uint32_t * pTs)
```

Description TOC §↑

Performs the same operations as *bspRawTsGet()*, but the returned value is converted to µs.

This allows for example to accumulate delays up to more than one hour in an unsigned 32-bit variable.

This function is declared "inline".

Example to check the duration of a second of the kernel:

```
uint32_t    ts, delta ;

aaTaskDelay (1) ;
ts = bspTsGet () ;
aaTaskDelay (bspGetTickRate ()) ;
delta = bspTsDelta (& ts) ;
aaLogMes ("Delay:%u us  Now:%u\n", delta, ts, 0, 0, 0) ;
```

Return value

The difference between the current counter value and the value pointed to by *pTs* converted to µs.

bspDelayUs

```
void      bspDelayUs          (uint32_t us)
```

Description

TOC §↑

Execute an active wait of the duration in micro second passed in parameter. The calling task is not suspended for the duration of the delay and therefore consumes CPU time.

However, the task can be preempted by a higher priority task that becomes ready. The waiting time as a parameter is therefore a minimum time.

The accuracy of the delay is of the order of 1% beyond 10µs.

swoInit

```
void      swoInit            (uint32_t        portBits,
                              uint32_t        cpuCoreFreqHz,
                              uint32_t        baudrate)
```

Description TOC §↑

Initialize the ITM mechanism that makes it possible to emit traces by the SWO pin in DEBUG mode.

portBits	A mask that allows you to specify the stimuli to configure.
cpuCoreFreqHz	The clock frequency of the processor in Hz.
Baudrate	The bit frequency of the SWO output

The values of *cpuCoreFreqHz* and *baudrate* are used to calculate the divisor needed to generate the SWO bit rate frequency.

To find out if the SWO mechanism is operational, use *swoIsEnabled()*.

Return value

None.

swoIsEnabled

```
void      swoIsEnabled       (void)
```

Description TOC §↑

Lets you know if SWO mechanism is initialized and operational.

If this function is called while the application is handled by a debugger, then the SWO mechanic is operational, and the *swoIsEnabled()* function will return a non-zero value.

If this function is called while the application is not managed by the debugger (application launched by a physical reset for example), then the SWO mechanic is not operational, and the *swoIsEnabled()* function will return 0. In this case the SWO management functions can be called, but will not do anything

swoSendXx

```
void      swoSend8          (uint8_t value, uint8_t portNo)
void      swoSend16         (uint16_t value, uint8_t portNo)
void      swoSend32         (uint32_t value, uint8_t portNo)
```

Description TOC §↑

These functions make it possible to issue words of different lengths on the stimulus *portNo*, from 0 to 31, of the ITM.

Return value

None

swoPutChar, swoPutStr

```
void      swoPutChar        (char     value)
void      swoPutStr         (char *   pStr)
```

Description TOC §↑

swoPutChar() transmits a character on stimulus 0 of the ITM. This function is used by the BSP to specify the redirection of the *logMes()* outputs to the SWO.

The function *swoPutStr()* allows to emit a string terminated by 0 on the stimulus 0 of the ITM.

Return value

None

4.15 Intrinsics

The kernel uses intrinsic functions of the compiler if they exist, otherwise the BSP must provide them. The functions used are therefore available for applications. Advantageously refer to the documentation of the compiler for the meaning and use of these functions.

Correspondence for GCC :

aaVA_LIST	__builtin_va_list
aaVA_START	__builtin_va_start
aaVA_END	__builtin_va_end
aaVA_ARG	__builtin_va_arg
aaISDIGIT	__builtin_isdigit
aaSTRLEN	__builtin_strlen
aaSTRCMP	__builtin_strcmp
aaSTRNCMP	__builtin_strncmp
aaSTRCPY	__builtin_strcpy
aaMEMCPY	__builtin_memcpy
aaMEMSET	__builtin_memset

The use of intrinsic functions allows in numerous uses to take place of the "C" library, as newlib or nanolib

5 License

GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a

section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept

compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.

- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- H. Include an unaltered copy of this License.

- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the

Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.