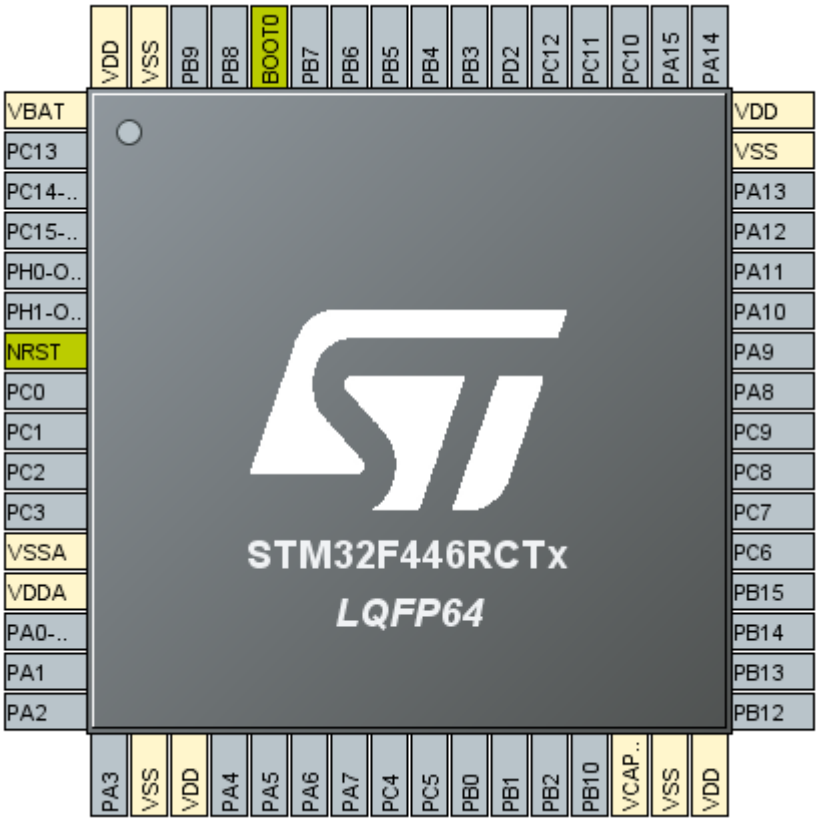
AdAstra RTK

Real Time Kernel

STM32 edition 2022-03



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

Limitation de responsabilité

DANS AUCUNE CIRCONSTANCE AUTRE QUE CELLES REQUISES PAR LA LOI APPLICABLE OU CONSENTIES PAR UN ACCORD ÉCRIT, LES TITULAIRES DE DROITS, OU TOUT AUTRE PARTIE QUI MODIFIE ET/OU TRANSFÈRE LE PROGRAMME AINSI QU'AUTORISÉ PRÉCÉDEMMENT, NE PEUVENT ÊTRE TENU POUR RESPONSABLE ENVERS VOUS POUR LES DOMMAGES, INCLUANT TOUT DOMMAGE GÉNÉRAL, SPÉCIAL, ACCIDENTEL OU INDIRECTS CONSECUTIFS A L'UTILISATION OU A L'INCAPACITÉ D'UTILISER LE PROGRAMME (NOTAMMENT LA PERTE DE DONNÉES OU L'INEXACTITUDE DES DONNÉES RETOURNÉES OU LES PERTES SUBIES PAR VOUS OU DES PARTIES TIERCES OU L'INCAPACITÉ DU PROGRAMME À FONCTIONNER AVEC TOUT AUTRE PROGRAMME), MÊME SI UN TEL TITULAIRE OU TOUTE AUTRE PARTIE A ÉTÉ INFORMÉE DE LA POSSIBILITÉ DE TELS DOMMAGES.

Annonce légale

Cet ouvrage contient des références à plusieurs produits ou technologies dont le copyright appartient à leurs propriétaires respectifs. Entre autre:

STM32, ST-LINK, STM32CubeMX are copyright © ST Microelectronics NV.

ARM, Cortex, Cortex-M, CoreSight, CoreLink, Thumb, Thumb-2, AMBA, AHB, APB, are registered trademarks of ARM Holdings.

GCC, GDB and other tools of GNU Collection Compiler are copyright © Free Software Foundation.

MISRA and MISRA C are registered trademarks of MIRA Limited.

Eclipse is copyright of the Eclipse community and all its contributors.

AdAstra-RTK is copyright © AdAstra-Soft.

Si une mention de copyright a été oubliée dans cet ouvrage et doit être mentionnée faites m'en part : contact sur AdAstra-Soft.com.

Sommaire

1 Guide de l'utilisateur

1.1 Introduction aux RTOS

Pourquoi utiliser un noyau temps réel, ou RTOS ?

Les quelques lignes qui suivent présentent les arguments les plus souvent avancés sur la question.

Une application traditionnelle sans RTOS sur un microcontrôleur utilise en général deux principes :

- Une gestion d'événements par interruptions
- Une « super boucle » qui appelle périodiquement et consécutivement les traitements à réaliser, qui doivent être écrits sous la forme d'automates d'état pour permettre une sorte de multitâche « round robin » coopératif.

Ces applications quand elles se complexifient au cours du temps, et que les mainteneurs se succèdent révèlent leurs défauts :

- Les timings deviennent difficiles à maitriser dans la super boucle.
- Les interactions entre les différentes parties de l'application deviennent inextricables.
- Pour résoudre les difficultés ci-dessus on développe des fonctionnalités d'exclusion, et de gestion de temps, Ces fonctionnalités sont en général non génériques et demandent à être augmenté à chaque nouvelle utilisation.
- La réalisation de ces fonctionnalités utilise des notions d'architecture de CPU et est coûteuse en temps.

Finalement on remarque qu'une bonne partie des fonctionnalités d'un RTOS ont été développées : on a ré inventé la roue. Mais sans avoir les avantages que l'on est en droit d'attendre d'un noyau temps réel

Ce qu'apporte un noyau temps réel :

- La gestion efficace de multiples tâches permet de décomposer l'application en tâches plus simples à développer, avec un ordonnancement automatique. Vous pouvez écrire vos tâches et laisser au noyau le soin de les faire fonctionner ensemble.
- Un RTOS fournit des services tout prêts et éprouvés : gestion des interruptions, timers, communication inter tâches (sémaphores, mutex, queues de message, mailbox)… Des exemples permettent de les mettre en œuvre dans votre application. Souvent les drivers adaptés au noyau pour périphériques les plus utilisés ou les plus complexes sont disponibles.
- Un RTOS fournit des outils de diagnostic sophistiqués, qui réduisent encore le temps de développement : débugger, traces configurables à plusieurs niveaux, surveillance des piles. Le diagnostic est souvent le service qui a le plus de valeur pour le développeur.

- Le RTOS fournit un BSP qui vous dispense d'écrire les couches bas niveau et de devoir maitriser l'architecture de plus en plus complexe des processeurs. Si un BSP spécifique n'existe pas, des exemples sont fournis et peuvent être adaptés. Votre application est portable (en ce qui concerne le noyau) sur tous les processeurs supportés par le noyau.

En conclusion, ajouter un RTOS à votre projet revient à y ajouter un ou plusieurs ingénieurs hautement compétents, pour une fraction du coût. Un bon RTOS a déjà résolu de nombreux problèmes cachés qu'il faut beaucoup de temps pour résoudre. De plus le fournisseur peut apporter un support. L'utilisation de solutions éprouvées réduit les risques et donne à votre projet les meilleures chances de succès.Il faut cependant admettre que la super boucle est adaptée à un certains nombre de projets, en particulier si le MCU a mettre en œuvre a peu de mémoire (quelques K octets) ou que le projet est très simple et n'évoluera pas. Dans tous les cas il faut bien réfléchir pour adopter la meilleure solution.

1.2 Présentation générale du noyau

AdAstra RTK est un noyau temps réel particulièrement bien adapté aux processeurs 32 bits de milieu de gamme : de quelque dizaines de Ko à environ 1 Mo de mémoire RAM. En général les applications utilisant des processeurs avec moins de ressources ne nécessitent pas de mettre en œuvre un noyau temps réel, et les applications qui nécessitent d'importantes ressources nécessitent aussi des fonctions plus évoluées (partitionnement, mémoire virtuelle, …).

AdAstra RTK a été développé avec des objectifs précis :

- Adapté nativement aux processeurs 32 bits de type ARM Cortex-M. Cela permet d'avoir un code bien structuré et non pollué par d'innombrables sections de compilation conditionnelle.
- Les concepts d'architecture sont volontairement simples pour faciliter la compréhension de tous les utilisateurs : étudiants, utilisateurs finaux, mainteneurs…
- Le standard de codage adopté a pour but de rendre le code maintenable, en renforçant sa lisibilité et l'obligation de commentaires judicieux. L'adoption de règles reconnues (MISRA) va dans le même sens.

Les principales caractéristiques de AdAstra RTK sont :

- Strictement préemptif et déterministe : La tâche de plus haute priorité est assurée de s'exécuter et de ne pas être interrompue jusqu'à ce qu'elle cède son droit d'exécution, ou qu'une tâche de plus haute priorité devienne prête à s'exécuter.
- Nombre de tâches quasi illimité (mais cependant dépendant de la mémoire disponible).
- Jusqu'à 256 niveaux de priorité. La gestion des priorités est optimisée si le nombre de niveau est inférieur ou égal à la taille du mot natif (32 niveaux sur un système 32 bits par exemple).

- Mutex récursif à héritage de priorité, pour éviter le phénomène d'inversion de priorité. Ce mutex dispose d'une implémentation complète de l'algorithme.
- Panoplie complète de communication inter tâche : sémaphore à compteur, mutex, signaux inter tâche, queues de messages, gestion de pools de messages.
- Gestion de timeout pour toutes les API potentiellement bloquantes. Un timeout nul ou illimité est autorisé.
- Timer logiciel rapide, sans tâche supplémentaire.
- Il est possible de paramétrer le noyau pour optimiser les ticks système, et économiser l'énergie (méthode parfois appelé tickless, mais qui ne l'est pas).
- Allocation statique de toutes les structures des objets du noyau, configurable par l'utilisateur. L'allocation des piles des tâches peut aussi être statique, et l'utilisateur peut choisir de se passer complètement d'allocation dynamique de mémoire. Cela se fait en utilisant une seule API : il n'y a pas de fonctions spécifiques à l'allocation statique et d'autres pour l'allocation dynamique.
- Ajustable aux besoins : Des définitions permettent de spécifier la quantité de chaque type d'objets du noyau (tâches, sémaphores, queues, etc.).
- Allocation dynamique de mémoire par l'algorithme TLSF qui est flexible, efficace et surtout déterministe. Le noyau n'utilise l'allocation dynamique de mémoire que sur demande de l'application. Cette allocation dynamique peut être inhibée lors de la configuration du noyau.
- Facilités de mise au point : analyse d'occupation des piles, tâche de log différé, statistiques d'utilisation de ressources, utilisation des caractéristiques spécifique du matériel (SWO). Utilisation intensive de vérifications par ASSERT, ce qui permet éventuellement de supprimer ces tests lors de la génération du code final. Gestion centralisée des erreurs fatales ou non.
- Une API claire et cohérente (orthogonale : https://en.wikipedia.org/wiki/Orthogonality_(programming)). Une fois familiarisé avec les conventions de codage et de nommage l'utilisateur peut deviner les noms des fonctions et les paramètres à fournir.
  Pour le développement des drivers il n'y a pas d'API dédiée aux fonctions d'interruption, qui utilisent l'API standard. S'il y a une violation des restrictions à l'utilisation par un driver d'interruption, cela est signalé.
- Le codage est en ANSI-C, avec le moins d'assembleur possible : Un portage typique utilise moins de 50 lignes d'assembleur. Cette caractéristique facilite, outre la lecture et donc la maintenance, le portage vers d'autres familles de processeurs.

1.3 Le Board Support Package

Le noyau AdAstra RTK est strictement indépendant des processeurs. Pour porter le noyau sur une famille de processeur il faut créer un Board Support Package (BSP) spécifique de cette famille. Ce BSP fournit :

- un jeu de fonctions prédéfini à adapter lors du portage
- Suivant les besoins de l'utilisateur des fonctions supplémentaire.

Le BSP repose souvent sur des logiciels fournis par les constructeurs : ARM CMSIS, STM32 Low Level Drivers, etc.

Un BSP standard est en général adapté à un circuit d'évaluation et propose :

- Une console sur UART (UART à sélectionner parmi tous ceux du processeur),
- Accès aux LEDs, et autres sorties GPIO,
- Accès aux boutons, et autres entrées GPIO,
- Une application de démonstration standard utilisant les éléments ci dessus.

Le BSP offre d'autres API génériques telles que :

- Configuration du tick système, et fonctions pour l'interroger ou le modifier
- Fonctions de mesure de temps (time stamp)
- Gestion des interruptions
- Configuration de debug

1.4 Statique ou dynamique ?

La configuration du noyau est statique : la quantité de chaque objet à rendre disponible dans le noyau (tâche, mutex, queue, timer, …) est spécifiée lors de la configuration, et ils sont alloués lors de la compilation. Ainsi la taille du noyau est connue dès la compilation, et il est certain que ces objets seront disponibles lors de l'exécution.

Un cas particulier concerne les piles des tâches. Plusieurs choix sont disponibles :

- La gestion de mémoire dynamique n'est pas implémentée dans le noyau. L'utilisateur doit alors prévoir lui-même les espaces mémoire à utiliser comme pile des tâches.
- La gestion de mémoire dynamique est implémentée, mais il est possible de ne pas l'utiliser au cas par cas, ce qui revient au cas précédent.
- La gestion de mémoire dynamique est implémentée et utilisée automatiquement par le noyau, l'utilisateur ne communique que la taille de chaque pile.

C'est donc la configuration du noyau associée à l'utilisation qui en est faite qui permet d'avoir une allocation entièrement statique, ou dynamique.

1.5 Les identificateurs

La plupart des objets fournissent lors de leur création un identificateur (handle) qui permet de les référencer lors des utilisations ultérieures. Ces identificateurs doivent être considérés comme des références opaques sans signification en dehors des API du noyau.

1.6 Librairie standard

Le noyau n'utilise pas de bibliothèque standard telle que « newlib » ou « newlib-nano », pour plusieurs raisons:

- Le noyau ne nécessite aucune de ces librairies.
- Une partie de ces librairies et des fichiers d'inclusion qui les accompagnent sont interdite par des standards tels que MISRA.
- Les ressources mémoire occupées par ces librairies peuvent être incompatibles avec certains processeurs.

Certaines fonctions de ces librairies peuvent être remplacées par celles fournies par le noyau, telles que celle-ci:

- *aaGetChar* Obtient un caractère de la console
- *aaPutChar* Envoie un caractère vers la console
- *aaPrintf* Equivalent, avec quelques restrictions, au printf() standard
- *aaPrintfEx* Printf sur un périphérique quelconque
- *aaSnPrintf* Equivalent à snprintf().
- *aaGets* Equivalent à gets() mais avec la sécurité de fgets()

Cependant le noyau utilise quelques fonctions intrinsèques des compilateurs, qui doivent être fournies par le BSP. La liste de ces fonctions est donnée en annexe.

La librairie standard « stdlib.h » fournit les fonctions de gestion dynamique de la mémoire de la famille *malloc()*. Ces fonctions reposent sur l'appel système *sbrk()*, qui gère le tas. Mais le tas est déjà géré par le noyau.

Par conséquent pour permettre l'utilisation des fonctions de la famille *malloc()* le fichier _syscall.c possède des fonctions qui déroutent la gestion de la mémoire dynamique de la librairie standard vers la gestion assurée par le noyau.

Il est à remarquer que l'utilisation de *printf()* entraîne l'allocation dynamique de plusieurs blocs, et une utilisation intensive de la pile. La plupart du temps il est plus avantageux d'utiliser *aaPrintf()* ou *aaSnPrintf()*.

1.6.1 Remarque sur l'utilisation de newlib.

Newlib pose deux problèmes principaux lors de son utilisation avec un noyau temps réel :

- Le réentrance de certaines fonctions : newlib a des fonctions qui ne sont pas « thread-safe ».
- La libération de la mémoire allouée par newlib lorsque la tâche qui a provoqué cette allocation se termine.

Le problème de la ré-entrance des fonctions est facilement résolu, car prévu lors de la conception de newlib : Il suffit de configurer AA_WITH_NEWLIB_REENT dans

aacfg.h. Après cela une structure de type *_reent* est ajouté au bloc de contrôle de chaque tâche, et le noyau assure la mise à jour de la variable *_impure_ptr* de newlib à chaque commutation de contexte.

Newlib n'a rien prévu pour libérer certains blocs de mémoire qu'elle a alloués. Il faut donc faire très attention lorsqu'une tâche utilise des fonctions de newlib qui allouent de la mémoire et qu'ensuite cette tâche est détruite : il peut y avoir des fuites de mémoire !

Cela est probablement du au fait que newlib est conçue pour des systèmes basés sur la notion de process et de threads. La mémoire allouée par newlib n'est libérée automatiquement qu'à la fin du process, par le système et non par newlib.

Le modèle d'un RT peut être comparé à un système avec un seul process, dont les threads sont les tâches. Comme le process n'est jamais terminé, la mémoire n'est pas libérée.

Si vous voulez réellement utilisée toutes les fonctionnalités de newlib, vous devrez vous même surveiller les allocations de mémoire et gérer sa libération.

1.7 MISRA compatible ?

Pour faire court non.

Principalement parce que le cycle de développement appliqué est différent sur certains points de celui décrit dans la version 2012 de ce standard.

Mais les règles de développement conformes à l'état de l'art ont été appliquées, en particulier sur les règles de codage. Les règles de codage appliquées sont destinées à :

- Assurer la performance du code
- Assurer la facilité de lecture du code ce qui assure sa fiabilité et sa maintenabilité.

Pour cela quelques règles ont été sciemment assouplies :

- Plusieurs points de sortie par fonction sont autorisés (rule 15.5). Cela permet d'assurer le contrôle des paramètres des fonctions en limitant la profondeur d'indentation. Une grande profondeur d'indentation est préjudiciable à la clarté du code et favorise les erreurs en cas de modification du code.
- Plusieurs « break » (rule 15.4) par structure « for » ou « while » sont autorisés. Cela assure une plus grande clarté du code.
- L'usage d'union (rule 19.2) est aussi limité que possible : un seul endroit. Mais son emploi est nécessaire pour des raisons de performance du code. L'usage de cet union est toujours cohérent : l'écriture puis la relecture se font toujours sur le même élément.
- Pour respecter l'encapsulation des données et l'opacité des identificateurs, la conversion d'un type pointeur sur void vers un autre type de pointeur est autorisée (rule 11.5).

1.8 Gestion des tâches

1.8.1 Creation

Une tâche est créée avec la fonction *aaTaskCreate()*. A cette fonction sont fournis entre autre :

- Le nom de la tâche.
- Le point d'entrée de la tâche : le nom de la fonction qu'elle doit exécuter.
- Un argument qui sera fourni à la fonction de la tâche.
- La taille de la pile qui doit lui être attribuée.
- Un pointeur sur la pile. Cela permet d'avoir une allocation complètement statique : l'utilisateur se charge de l'allocation de la pile au moment de la création de la tâche, et de la libération de la pile lors de la destruction de la tâche. Si le pointeur de pile vaut NULL et que l'allocation de mémoire dynamique est autorisée, alors le noyau se charge de la gestion dynamique de la pile.

En retour de la création de la tâche un identificateur de tâche est fourni à l'utilisateur, il doit être utilisé avec l'API de gestion des tâches. Un identificateur spécial *AA_SELFTASKID* identifie la tâche courante, l'identificateur réel de la tâche courante peut être obtenu avec *aaTaskSelfId()*.

Le nombre maximal de tâches que gère le noyau est spécifié lors de la configuration du noyau (fichier aacfh.h).

1.8.2 Destruction

La destruction d'une tâche peut se faire par plusieurs moyens :

- La tâche sort de la fonction spécifiée lors de la création par return, ou en ayant atteint la fin de la fonction.
- En exécutant *aaTaskDelete()* avec AA_SELFTASKID.
- Une autre tâche exécute *aaTaskDelete()* avec l'identificateur de la tâche.

Une tâche doit être détruite avec grande prudence, à cause de la difficulté de récupération des ressources qu'elle détient. En particulier si la tâche détruite détient des mutex ou des sémaphores.

Si une tâche utilise une pile statique allouée par l'utilisateur, lors de la destruction de la tâche la callback *aaUserReleaseStack()* est appelée. Cela donne l'opportunité à l'utilisateur de savoir qu'un bloc mémoire se libère et de le gérer en conséquence.

1.8.3 Suspension

Une tâche peut être suspendue par *aaTaskSuspend()*. Dans cet état, même si elle est prête à s'exécuter elle ne le fait pas.

On peut lui faire reprendre son activité avec *aaTaskResume()*.

Lorsqu'une tâche est suspendue alors qu'elle est dans un état d'attente (*aaTaskDelay()* par exemple), elle ne passe pas immédiatement en état suspendu. Elle ne passe dans l'état suspendu qu'a la fin de l'état d'attente : fin du délai, échéance du timeout, obtention de la ressource…

1.8.4 Gestion de priorité

AdAstra permet de gérer jusqu'à 256 niveaux de priorités, 0 étant la priorité la plus faible, 255 étant la plus élevée.

AdAstra est strictement préemptif et déterministe : La tâche prête de plus haute priorité est assurée de s'exécuter et de ne pas être suspendue jusqu'à ce qu'elle cède son droit d'exécution ou qu'une tâche de plus haute priorité devienne prête. Cela veut dire qu'une tâche peut empêcher toutes les tâches de priorité inférieure de s'exécuter si elle ne s'endort pas ou ne se bloque pas en attente de ressource par exemple.

La seule exception concerne les interruptions qui peuvent, quand elles ne sont pas inhibées, interrompre temporairement la tâche active de plus haute priorité.

Si plusieurs tâches de même priorité sont prêtes celle qui se trouve en tête de liste pour cette priorité s'exécute. Elle s'exécutera jusqu'à ce qu'elle cède son droit d'exécution : *aaTaskDelay()*, *aaTaskYield()* ou attente de ressource par exemple. A ce moment-là elle est insérée en fin de liste, et la tâche en tête de liste s'exécute. Ce mécanisme permet une de gestion en « round-robin » coopératif. Cela respecte le principe de gestion strictement préemptive et déterministe exprimé plus haut.

Le nombre de niveaux de priorité qui doit être géré est configuré par *AA_PRIO_COUNT* dans *aacfg.h*. L'ordonnanceur est plus efficace si le nombre de priorités est au plus égal au nombre de bits dans un mot natif du processeur (32 bits sur un processeur 32 bits). L'algorithme général s'applique au-delà de 2 mots (64 niveaux de priorité sur un processeur 32 bits). Limiter le nombre de priorité permet d'économiser de la mémoire car le noyau maintient une liste par priorité.

La priorité 0 est réservée à la tâche « idle » qui s'exécute quand aucune autre tâche n'est prête à s'exécuter, autrement dit il n'est pas possible de créer une autre tâche de priorité 0. La tâche « idle » est créée automatiquement à l'initialisation du système, et ne peut pas être détruite.

Une tâche dispose d'une priorité de base, qui est celle qui lui est attribuée à sa création ou par *aaTaskSetPriority()* et qui peut être obtenue par *aaTaskGetBasePriority()*. Lors de l'utilisation des mutex, la priorité d'une tâche peut changer à cause du mécanisme d'héritage de priorité. La priorité réelle à laquelle s'exécute la tâche peut être obtenue avec *aaTaskGetRealPriority()*.

1.8.5 Les états des tâches

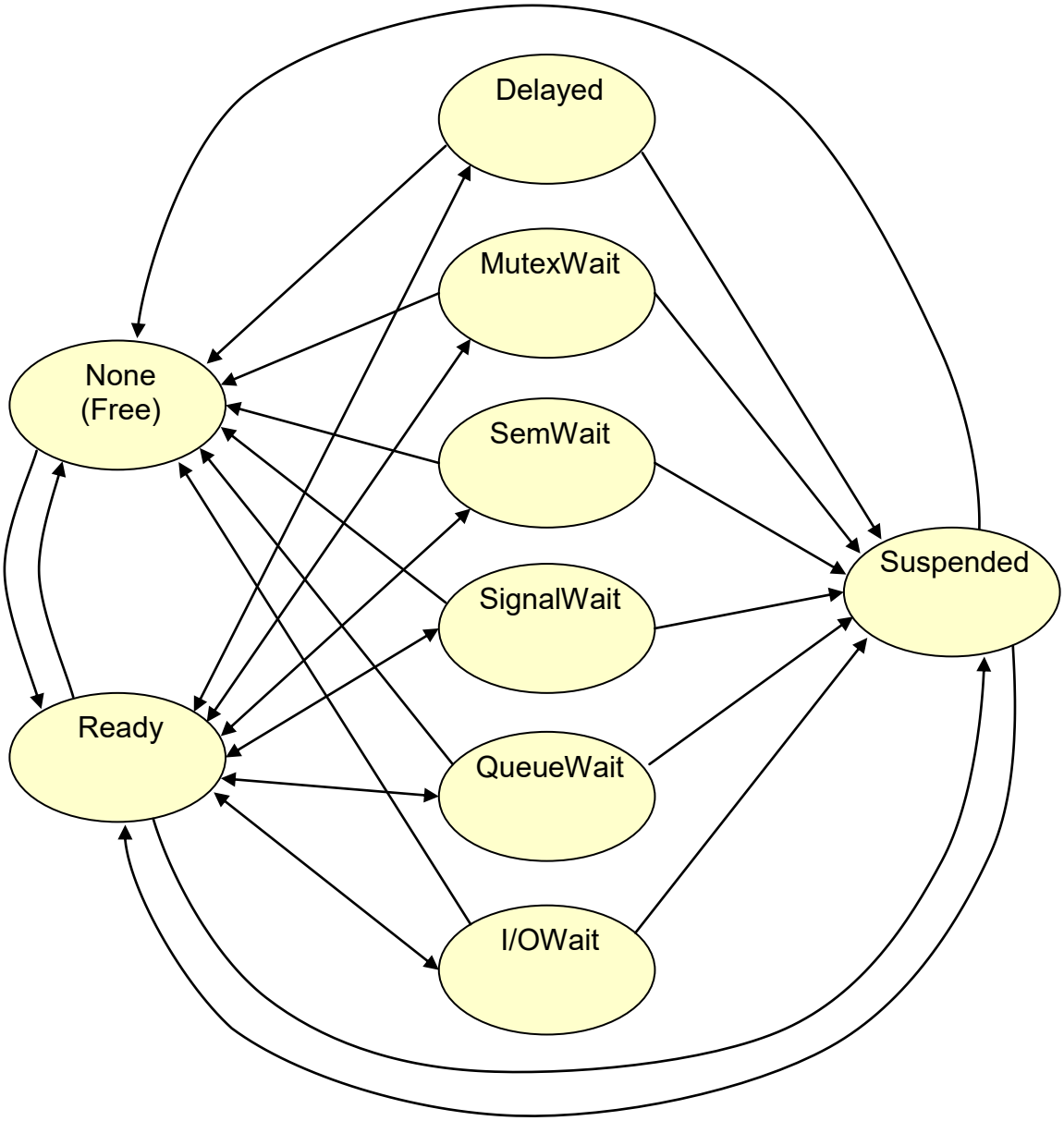
Les tâches sont dans un des états suivants :

aaNoneState	Ces tâches sont libres, c'est à dire non créées.
aaReadyState	Ces tâches prêtes à s'exécuter. Une seule tâche s'exécute : une de celle qui a la plus grande priorité, et qui est dans cet état.
aaDelayedState	Ces tâches sont en attente pour un certain délai, qui peut être infini.
aaMutexWaitingState	Ces tâches attendent d'obtenir un mutex qui est détenu par une autre tâche.
aaSemWaitingState	Ces tâches attendent que le compteur d'un sémaphore devienne positif et l'obtenir.
aaSignalWaitingState	Ces tâches attendent qu'une combinaison de leurs signaux soient positionnés par d'autres tâches.
aaQueueWaitingState	Ces tâches attendent de pouvoir écrire ou lire un message dans une queue de messages.
aaIoWaitingState	Ces tâches attendent un événement dans un driver.
aaSuspendedState	Ces tâches sont suspendues : même si elles réunissent les conditions pour s'exécuter elles restent inactives.

La suspension d'une tâche est un mécanisme particulier :

Si une tâche est dans l'état aaReadyState lorsqu'elle est suspendue, elle passe immédiatement dans l'état suspendu.

Si une tâche est dans un état d'attente lorsqu'elle est suspendue, elle continue d'attendre, et dès que la condition de sortie de l'état d'attente est remplie elle entre dans l'état suspendu. Même si la suspension de la tâche s'effectue en deux étapes, la tâche est effectivement suspendue dès la demande de suspension.



Ce schéma montre toutes les transitions d'états licites pour une tâche.

1.9 Les interruptions

La gestion des interruptions est réservée au noyau et aux drivers, qui doivent inclure *aakernel.h*, qui lui-même inclut *bspcfg.h* qui déclare les fonctions de gestion des interruptions :

bspEnableIrq()	Autorise les interruptions
bspDisableIrq()	Inhibe les interruptions
aaCriticalEnter ()	Entre dans une section critique
aaCriticalExit()	Quitte une section critique
bspSaveAndDisableIrq	Mémorise le mot d'état d'interruption et inhibe les interruptions
bspRestoreIrq	Restaure le mot d'état d'interruption
aaIntEnter()	A appeler à l'entrée d'une fonction d'interruption
aaIntExit()	A appeler à la sortie d'une fonction d'interruption

1.9.1 Types et priorités des interruptions

Le noyau autorise deux types d'interruptions.

- Les interruptions « zéro latence ». Ces interruptions ont les priorités les plus élevées, supérieures à BSP_MAX_INT_PRIO, et ne sont jamais masquées par le noyau. La latence de ces interruptions ne dépend donc que du matériel.
  Ces interruptions ne doivent jamais appeler l'API du noyau.
- Les interruptions gérées par le noyau, qui sont en général celles utilisées par les drivers de périphériques. Elles ont une priorité entre BSP_MAX_INT_PRIO et BSP_MIN_INT_PRIO. Ces interruptions sont masquées par le noyau dans les sections critiques.

La priorité BSP_MIN_INT_PRIO est définie dans *bsp.h*. Elle est imposée par le nombre de bits du masque de priorité du gestionnaire d'interruption du processeur. C'est la priorité la plus basse, ou la moins prioritaire.

La priorité BSP_MAX_INT_PRIO est définie dans *bsp.h*, et peut être modifiée suivant les besoins. C'est la priorité la plus haute, c'est à dire la plus prioritaire (ou la plus urgente)

1.9.2 Sections critiques

1.9.2.1 Section critique standard

Le moyen recommandé pour inhiber temporairement les interruptions est d'utiliser une section critique standard: Une section critique inhibe les interruptions de priorité inférieures à BSP_MAX_INT_PRIO. Les sections critiques sont réentrantes : il faut appeler autant de fois *aaCriticalExit()* qu'il y a eu d'appel à *aaCriticalEnter ()*.

1.9.2.2 Section critique rapide

Les fonctions *bspSaveAndDisableIrq()* et *bspRestoreIrq()* permettent de créer des sections critiques rapides (une à 2 instructions assembleur). Cependant les sections critiques de ce type ne sont pas prises en compte par le mécanisme de calcul du temps d'inhibition maximal des interruptions si celui-ci est validé.

Ces sections critiques rapides doivent être réservées à de très courtes sections de code. En particulier il est possible d'inclure une section critique rapide dans une section critique standard, mais l'inverse est interdit.

Les sections critiques rapides sont réentrantes.

Exemple d'utilisation :

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

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

1.9.3 Validation et inhibition des interruptions

Toutes les interruption peuvent être validées et inhibé globalement avec *bspEnableIrqAll()* et *bspDisableIrqAll()*. Cela agit sur le flag de validation d'interruption général du CPU et donc concerne toutes les interruptions, même celles dites « zéro latence ». Il est donc recommandé de ne pas utiliser ces fonctions.

Les interruptions des niveaux de priorité gérés par le noyau peuvent être validées et inhibées avec *bspEnableIrq() et bspDisableIrq()*. Cela agit au niveau du masque de priorité des interruptions du processeur.

Ces deux jeux de fonctions sont indépendants.

Ces fonctions ne sont pas réentrantes : L'utilisation de *ces fonctions* doit être faite avec prudence et doit être réservée aux suspensions très courtes qui ne font pas appel à des fonctions susceptibles d'utiliser elles même ces fonctions.

Il ne faut pas entremêler une section critique (standard ou rapide) et ces fonctions : leur usage est exclusif.

1.9.4 Handler d'interruption

Lors de l'écriture d'un handler d'interruption il est absolument nécessaire d'appeler *aaIntEnter()* au tout début du handler, puis d'appeler *aaIntExit()* à la fin du handler.

Exemple :

```
//  SysTick interrupt handler

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

Sur ARM CORTEX-M les interruptions sont réentrantes : plusieurs interruptions de priorité croissante peuvent s'imbriquer.

1.9.4.1 Définition de la priorité d'une interruption

La gestion des priorités des interruptions peut être troublante au premier abord. En effet la relation entre les priorités et leur valeur numérique n'est pas toujours intuitive. D'autre par les priorités MAX et MIN sont ajustables, il est donc difficile d'utiliser des constantes prédéfinies pour définir des priorités intermédiaires.

Pour unifier et simplifier ces notions un mode d'abstraction des priorités des interruptions est utilisé par AdAstra.

Le niveau le plus prioritaire est défini par BSP_MAX_INT_PRIO.

Le niveau le moins prioritaire est défini par BSP_MIN_INT_PRIO.

Il est convenu qu'en ajoutant un offset à la priorité la plus basse on progresse vers la priorité la plus haute. Et inversement en soustrayant un offset à la priorité la plus haute on progresse vers la priorité la plus basse.

Pour mettre en œuvre cette notion deux fonctions sont utilisées :

bspIrqPrioMaxMinus(x)

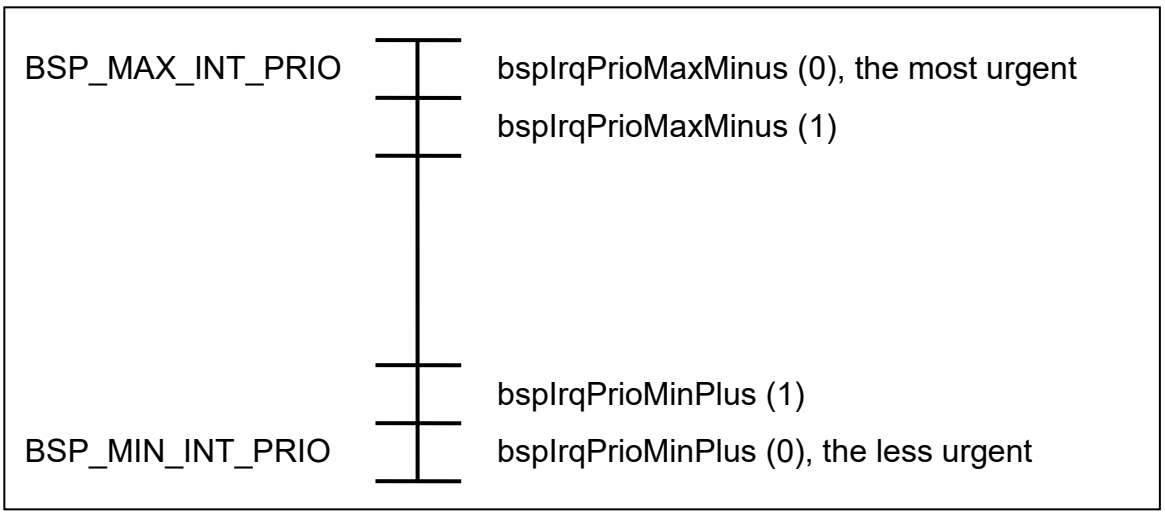
> Le paramètre x de cette fonction indique que nous voulons obtenir le niveau de priorité qui est x plus bas que le niveau maximum.
> *bspIrqPrioMaxMinus(0)* correspond à BSP_MAX_INT_PRIO,
> *bspIrqPrioMaxMinus(1)* correspond au niveau immédiatement inférieur à BSP_MAX_INT_PRIO, etc.
>
> Cette fonction est pratique pour définir les niveaux d'interruption les plus élevés.

bspIrqPrioMinPlus(x)

> Le paramètre x de cette fonction indique que nous voulons obtenir le niveau de priorité qui est x plus élevé que le niveau minimum.
> bspIrqPrioMinPlus(0) correspond à BSP_MIN_INT_PRIO,
> bspIrqPrioMinPlus(1) correspond au niveau immédiatement supérieur à BSP_MIN_INT_PRIO, etc.

Cette fonction est pratique pour définir les niveaux d'interruption les plus bas.

Les deux fonctions assurent que le niveau de priorité est valide, dans l'intervalle BSP_MAX_INT_PRIO - BSP_MIN_INT_PRIO.



Ces fonctions ne permettent pas d'initialiser des constantes. Pour cela deux macros équivalentes sont définies :

BSP_IRQPRIOMAX_MINUS
BSP_IRQPRIOMIN_PLUS.

Exemple :

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

1.10 Timer système et basse consommation

AdAstra est un système qui nécessite un timer pour assurer ses services. Ce timer génère une interruption, nécessaire à certaines fonctions telles que la gestion des timeout.

Ce timer doit être fourni par l'architecture matérielle : SysTick (sur les architectures ARM Cortex-M), timer standard ou basse énergie (low-power timer). Il est configuré par le BSP.

Dans les applications usuelles des microcontrôleurs le CPU exécute en permanence une tâche. Quand l'application n'a rien à faire, c'est la tâche « idfle » du RTOS qui s'exécute : Le timer est périodique, et la consommation quasiment constante à un niveau élevé.

Certaines applications nécessitent de diminuer la consommation du système quand sa charge de travail diminue. Il y a plusieurs approches possibles.

Mettre le processeur en sommeil entre les tick système

La première approche qui est facile consiste à demander à la tâche « idle » d'endormir le CPU, Le CPU sera réveillé à chaque interruption du timer système ou d'un périphérique.

Cela représente un gain de consommation non négligeable. Cependant si la fréquence du timer système est élevée (couramment 1 kHz), le CPU se réveille fréquemment pour presque rien.

Etirer le timer système.

L'idée est de mettre le CPU en sommeil le plus longtemps possible, donc de ne pas le réveiller pour les ticks inutiles. Cette technique est appelé « Etirement du tick » (tick stretching), car les interruptions du timer ne sont plus périodiques.

Lorsque la tâche « idle » doit mettre en sommeil le CPU, elle demande au noyau de dire dans combien de ticks il souhaite être réveillé. Cela correspond au timeout ou au timer logiciel en cours le plus court.

Le timer est alors programmé pour cette durée, et le CPU mis en sommeil.

Si le timer arrive à échéance, le BSP indique au noyau que ce nombre de ticks se sont écoulés, et le noyau met à jour son état. Puis le timer est reconfiguré pour qu'il reprenne le rythme périodique (à 1 ms par exemple).

Mais le CPU peut être réveillé n'importe quand par une autre interruption que celle du timer. Dans ce cas il faut :

- Calculer combien de ticks se sont écoulée depuis la mise en sommeil du CPU, et avertir celui-ci afin qu'il mette à jour son état.
- Reconfigurer le timer pour qu'il reprenne le rythme périodique (à 1 ms par exemple).
- Libérer les interruptions afin que l'interruption qui a réveillé le CPU soit traitée.

Tout cela avec un certain nombre de contraintes :

- La mesure du temps doit rester précise. Il n'est pas question de perdre une fraction de tick quand le timer est reconfiguré, à la mise en sommeil ou au réveil du CPU. Le décompte du temps est donc le même, que les ticks soient périodiques ou étirés.
- AdAstra supporte les ZLI (Zero Latency Interrupts). Il ne faut donc pas inhiber ces interruptions afin de préserver cette fonctionnalité. Cependant sur ARM Cortex-M, il est demandé que les interruptions soient inhibées lors de la mise en sommeil par l'instruction WFI. L'implantation des ticks étirés ne peut pas faire autrement que d'inhiber les interruptions, mais la conception utilisée réduit au minimum le temps d'inhibition des interruptions.

  Rien n'est gratuit ni facile : Pour bénéficier de l'économie d'énergie procurée par l'étirement des ticks, il faut accepter une latence supérieure de quelques dizaines de nano secondes pour les interruptions dites ZLI.
- L'idéal est de pouvoir mettre en veille le CPU pendant la plus grande période possible. Cependant il y a un compromis à faire en considérant la granularité du tick qui doit être fine si on a besoin d'une mesure de temps précise, et le nombre de bits du compteur du timer dont dépend la durée maximale de mise en sommeil. Par exemple si le compteur a 16 bits, et que la granularité du tick est de 1ms, l'étirement du tick pourra mettre le CPU en sommeil 32 secondes au maximum.
- La fréquence nominale du tick ne peut plus être modifiée avec *bspSetTickRate()*. Pour des raisons de performance cette fréquence est fixée par codage dans la fonction *bspTickStretch()* du BSP, et BSP_TICK_RATE doit avoir la valeur correspondante.

Le choix de configurer le noyau pour utiliser l'étirement des ticks est simplement réalisée en définissant la valeur de AA_TICK_STRETCH dans le ficher aacfg.h :

0 Tick périodique.

1 Tick étiré..

L'étirement des ticks est géré par le BSP dans la fonction *bspTickStretch_()* qui est appelée par la tâche « idle ». Cette fonction fait partie du BSP car elle est dépendante du timer utilisé.

Cette fonction peut utiliser l'une ou l'autre des méthodes indiquée plus haut.

1.11 Exclusion mutuelle : Le mutex

Un mutex est un objet qui permet d'assurer un accès exclusif à une ressource telle qu'un périphérique ou une structure de données. Il est créé avec *aaMutexCreate()*.

Un mutex est une variante du sémaphore mais avec les restrictions suivantes :

- Il ne peut être utilisé que pour assurer une exclusion mutuelle
- Il ne peut être donné que par la tâche qui l'a obtenu
- Il ne peut pas être acquis ou donné par une interruption
- On ne peut pas faire de « flush » sur un mutex.

Par contre il a des caractéristiques particulières :

- Il peut être acquis de façon récursive : la même tâche peut acquérir plusieurs fois le même mutex, puis le libérer autant de fois qu'elle l'a acquis.
- Le mutex utilise un algorithme d'héritage de priorité des tâches, afin d'éviter le phénomène d'inversion de priorité illimité, qui a pour effet qu'une tâche de basse priorité empêche une autre tâche de plus haute priorité de s'exécuter.

Le mécanisme d'héritage de priorité est coûteux en ressources. Il doit tenir compte de nombreux cas, tels que les tâches qui possèdent plusieurs mutex, d'une tâche qui change de priorité, ou d'une tâche en attente de mutex qui abandonne sur timeout. Tous ces cas demandent de recalculer la priorité des tâches qui possèdent des mutex qui sont aussi attendus par d'autres tâches.

Si des tâches doivent acquérir plusieurs mutex, il y a un risque d'inter blocage (deadlock). L'utilisateur doit mettre en place une stratégie pour éviter cela, par exemple en s'assurant que toutes les tâches acquièrent les mutex dans le même ordre, et les libèrent en ordre inverse.

Le nombre maximal de mutex que gère le noyau est spécifié lors de la configuration du noyau.

1.12 Sémaphores

AdAstra fournit un sémaphore à compteur, ce qui est utile pour garder une ressource à instances multiples, par exemple un tableau de plusieurs éléments. Il est créé par *aaSemCreate()*.

Un sémaphore peut être vu comme un compteur. Quand une tâche acquiert un sémaphore en utilisant *aaSemTake()* :

- Si le compteur est supérieur à 0, le compteur est décrémenté, et la tâche peut continuer son exécution.
- Si le compteur est inférieur ou égal à 0, la tâche est suspendue jusqu'à ce que le compteur devienne positif ou que le timeout optionnel arrive à échéance.

En sortie de *aaSemTake()* un code d'erreur indique le résultat de l'opération.

Un nombre quelconque de tâches peuvent être mises en attente d'acquérir un sémaphore.

Quand une tâche ou une fonction d'interruption libère un sémaphore avec *aaSemGive()*:

- Si aucune tâche n'est en attente du sémaphore, le compteur du sémaphore est incrémenté.
- Si une tâche est suspendue en attente du sémaphore, cette tâche est activée, et retourne avec un code qui indique le succès de l'opération. Le compteur est inchangé.

Si plusieurs tâches sont en attente, la tâche de plus haute priorité est sélectionnée pour être activée. Si la tâche activée a une priorité supérieure à celle de la tâche courante qui a libéré le sémaphore, celle-ci est immédiatement préemptée par la tâche activée.

Un sémaphore peut être libéré par une fonction d'interruption, mais ne peut pas être acquis par une fonction d'interruption.

Si une tâche est détruite alors qu'elle possède un sémaphore, celui-ci reste en l'état, ce qui a généralement un effet imprévisible sur l'application.

Le nombre maximal de sémaphores que gère le noyau est spécifié lors de la configuration du noyau.

Dans les versions de AdAstra-RTK antérieures à 1.10 il existait une variante du sémaphore appelée mutex simple : un mutex sans héritage de priorité. Ce mutex simple a été supprimé.

1.13 Signaux inter tâches

Les signaux inter tâche sont un moyen de communication inter tâche, ce qui est très différent des signaux POSIX qui s'apparentent plus à des interruptions pour traiter des erreurs et exceptions.

Chaque tâche dispose d'un certain nombre de signaux, dont le nombre est défini par la taille du type *aaSignal_t* défini dans *aabase.h*. (16 si *aaSignal_t* est défini en tant que *uint16_t* par exemple).

Une tâche peut attendre qu'un ou plusieurs de ses propre signaux soient positionnés. N'importe quelle tâche ou interruption peut positionner un signal dans une tâche. Ce mécanisme est rapide et permet par exemple à une tâche d'être activée à chaque occurrence d'une interruption, sans avoir à utiliser un sémaphore plus coûteux en ressources.

Deux modes sont utilisés pour attendre un signal :

AA_SIGNAL_AND	La tâche attend jusqu'à ce que tous les signaux demandés soient signalés.
AA_SIGNAL_OR	La tâche attend jusqu'à ce que au moins un des signaux demandés soient signalé.

Lorsque la tâche revient de *aaSignalWait()* les signaux qui ont servi à déclencher le retour sont transmis, et effacés dans le descripteur de tâche.

Exemple : une tâche dont l'identificateur est *mainTaskId* attend pendant 300 ticks que deux autres tâches positionnent chacune avec *aaSignalSend()* un signal: (0x80 et 0x40)

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

Si la fonction retourne AA_TIMEOUT, il est possible d'examiner *sigsOut* pour déterminer quelle tâche n'a pas positionné son signal.

L'une des tâches attendue exécute:

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

L'autre tâche attendue exécute:

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

1.14 Gestion de mémoire dynamique

Il est souvent pratique de pouvoir utiliser l'allocation de mémoire dynamique : les fonctions *malloc()*, *free()*, etc.

Cependant certaines règles de programmation l'interdisent. Pour répondre à tous ces besoins, trois options sont disponibles :

- Ne pas utiliser d'allocation dynamique : dans ce cas les fonctions aaM*alloc()*, *aaFree()*, etc ,ne sont pas disponibles.

- Utiliser une allocation de blocs : chaque bloc ne peut être alloué qu'une fois car il ne peut pas être libéré. Cela donne la souplesse de l'allocation dynamique avec *aaMalloc()*, mais évite les problèmes de fragmentation ou de réutilisation de blocs de mémoire avec *aaFree()* et *aaRealloc()*. C'est un cas très fréquent dans la conception de systèmes embarqués par exemple, où les tâches et les tampons sont tous créés une seule fois au démarrage du système.
- L'allocation de mémoire dynamique complète, en utilisant l'algorithme TLSF.

Pour l'allocation dynamique de mémoire le noyau propose les fonctions :

aaMalloc()

aaFree() TLSF seulement

aaRealloc() TLSF seulement

aaCalloc() TLSF seulement

Ces fonctions sont « thread safe ».

1.14.1 L'allocation dynamique TLSF

L'algorithme utilisé est dérivé du TLSF (Two-Level Segregate Fit) dont la description peut être trouvée sur http://www.gii.upv.es/tlsf/.

Cet algorithme a les avantages suivants dans un système temps réèl :

- Déterministe : TLSF a un coût constant en O(1)
- Rapide
- Efficace, il limite la fragmentation de la mémoire

L'algorithme a été adapté aux processeurs ayant peu de mémoire. Dans une implémentation standard chaque bloc alloué à un entête qui permet de contenir les informations de gestion du bloc (taille, chaînage, …), et occupe 8 octets dans le cas général d'un processeur 32 bits. Dans cette implémentation entête a été réduit à 4 octets, mais en contrepartie la taille du pool de mémoire gérable est réduite.

Les caractéristiques de l'allocateur TLSF sont :

- Le bloc fourni est toujours aligné sur 8 octets, ce qui le rend compatible avec l'alignement requis pour une pile ARM par exemple.
- L'entête de chaque bloc est de 4 octets seulement.
- La taille maximale de la partition mémoire est de 262143 octets. Plusieurs partitions de la même taille peuvent être utilisées.
- Le bloc descripteur de la partition est prélevé sur la partition elle-même lors de l'initialisation de celle-ci.

L'espace occupé par le descripteur de la partition mémoire dépend de la taille de celle-ci. Dans le fichier *aatlsf.c* il est possible d'indiquer la taille maximale de

l'espace mémoire géré avec *FLI_MAX_INDEX*, afin d'optimiser la taille du descripteur :

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%

Lorsque l'utilisateur crée une partition TLSF qu'il gère lui-même (sans utiliser l'API *aaMalloc()*, *aaFree()* etc), un jeu de fonction spécifique permet d'y accéder.

1.14.2 L'allocation de blocs

Cette allocation utilise un algorithme très rapide et très simple. Il est initialisé en lui fournissant un bloc de mémoire (la partition mémoire), ensuite il le subdivise à la demande, sans fragmentation.

Le descripteur de partition est petit : 2 mots de 32 bits pour un système 32 bits. Ce descripteur est alloué au début de la partition elle-même.

Les blocs alloués ont les caractéristiques suivantes :

- Ils ont une taille variable, toujours multiple de 8 octets
- Ils sont alignés sur une frontière d 8 octets.
- Ils n'ont pas d'entête (overhead)
- Une fois alloué un bloc ne peut pas être libéré.

Il est possible d'utiliser en même temps plusieurs partitions d'allocation par bloc, dans des zones mémoire différentes par exemple, si le microcontrôleur le supporte.

1.14.3 Configuration de l'allocation de mémoire

La configuration de l'allocation de mémoire dynamique se fait en définissant des constants dans le fichier *aacfg.h*.

Il est possible d'inclure au noyau l'un ou l'autre des algorithmes d'allocation dynamique, ou même les deux ou aucun. Cela se fait avec :

```
#define   AA_WITH_TLSF       1   // Include TLSF

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

La fonction *aaMalloc()* (ainsi que les autres fonctions de la famille si elles sont valides) est générique : elle s'adapte à l'algorithme affecté à l'allocation dynamique. Pour cela il faut définir une seule des constantes :

```
#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
```

La déclaration de la constante *AA_WITH_MALLOC_TLSF* provoque la déclaration de la constante *AA_WITH_MALLOC* qui est utilisée par le noyau pour savoir si l'allocation dynamique de mémoire est utilisable.

La fonction *aaRealloc()* est complexe et il est possible d'économiser son code en l'inhibant avec la définition de *AA_WITH_REALLOC* dans *aacfg.h*.

Quand la gestion dynamique de mémoire est autorisée, le noyau créée automatiquement une partition mémoire avec les informations sur le tas fournies par le script du linker (*_heap_begin_* et *_heap_end_*), en appelant *aaMallocInit_()*.

Si la partition doit être placée à un endroit précis en mémoire en ignorant les informations du linker, il faut définir les constantes :

```
#define   AA_HEAP_BEGIN   xxx    // Heap address

#define   AA_HEAP_SIZE    yyy    // Heap byte size
```

Il est ensuite possible à l'application d'utiliser les fonctions *aaMalloc()*, *aaFree()*, etc.

Dans le cas de l'allocation TLSF la fonction *aaMemPoolCheck()* peut être utilisée pour vérifier l'intégrité de l'allocation dynamique, ce qui est possible grâce aux informations contenues dans le descripteur de la partition et l(entête de chaque bloc.

1.14.4 Portage

Les fichiers *aatlsf.c* et *aatlsf.h* réalisent l'allocation de mémoire dynamique TLSF.

Les fichiers *aamemblock.c* et *aamemblock.h* réalisent l'allocation de mémoire dynamique par bloc.

L'API (*aaMalloc()*, *aaFfree()*, …) est implémentée par les fichier *aamalloc.c* et *aamalloc.h*.

Pour changer l'allocateur dynamique il suffit de réimplanter le fichier *aamalloc.c*.

1.15 Timers logiciels

Un timer logiciel peut être créé par n'importe quelle tâche, il est ensuite utilisé pour appeler une fonction callback après un temps prédéterminé. La fonction callback est exécutée dans le contexte de l'interruption du tick système.

Un timer logiciel est :

- Créé par *aaTimerCreate()*, dans l'état inactif.
- Configuré avec *aaTimerSet()*, pour indiquer la callback à utiliser ainsi que le délai à utiliser en nombre de tick système.
- Démarré avec *aaTimerStart()*.
- Arrêté avant son terme par *aaTimerStop()*.
- Détruit par *aaTimerDelete()*.

Le prototype de la fonction callback est :

typedef uint32_t (aaTimerCallback) (uintptr_t arg) ;*

Le timer peut être utilisé en « one shot » : quand la callback est appelée le timer se trouve dans l'état arrêté. Si la callback retourne 0 il restera arrêté.

Le timer peut être périodique : Si la callback retourne une valeur non nulle le timer est réinitialisé avec la valeur de timeout spécifiée par *aaTimerSet()* puis redémarré.

La callback peut reconfigurer le timer avant de retourner une valeur non nulle, ce qui permet d'avoir un timer avec une durée variable.

La callback d'un timer est appelée dans un contexte d'interruption. Les restrictions applicables aux drivers d'interruption s'appliquent donc à cette callback : entre autre ne pas prendre de sémaphore, ne pas appeler une API bloquante, être aussi courte que possible.

Un timer peut être utilisé comme un watchdog : démarré au début d'un traitement, il peut être redémarré en cours de traitement pour éviter l'échéance du timer, puis arrêté une fois ce traitement terminé. Si la callback est appelée c'est que le traitement a duré trop longtemps.

```
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) ;
}
```

Le nombre maximal de timers que gère le noyau est spécifié lors de la configuration du noyau.

1.16 Queue de messages

Les queues de messages sont un outil de communication inter tâche. Elles permettent :

- A un nombre variable de messages d'être stockés dans un tampon géré en FIFO. Le nombre maximal de messages et la taille maximale des messages sont spécifiés lors de la création de la queue de messages avec *aaQueueCreate()*.
- Les messages sont stockés par recopie dans les tampons de la queue de messages
- Les messages peuvent être de longueur variable, seule la partie utile des messages est recopiée, en entrée et en sortie.
- N'importe quelle tâche ou interruption peut envoyer un message à une queue de message. Si la queue est pleine la tâche peut être bloquée avec timeout, et si l'appelant est une interruption l'erreur AA_EWOULDBLOCK est retournée.
- Toute tâche peut lire un message dans la queue de messages, et si elle est vide la tâche sera bloquée avec timeout. Si la lecture sur une queue vide est demandée par une interruption, ou si le timeout spécifié est 0, la valeur de retour est AA_EWOULDBLOCK.

Lors de la création de la queue de messages l'utilisateur peut fournir l'adresse du tampon qui sera utilisé pour stocker les messages, ce qui permet une allocation statique. Si l'adresse vaut NULL et que la gestion dynamique de mémoire est autorisée, la queue gère l'allocation du tampon ainsi que sa libération.

Plusieurs tâches peuvent attendre d'écrire ou de lire un message dans une queue de messages. Le paramètre *flags* de la fonction *aaQueueCreate()* permet de spécifier l'ordre des tâches :

AA_QUEUE_PRIORITY	Les tâches sont gérées par ordre de priorité : la tâche de priorité la plus élevée obtiendra le 1er message disponible.
AA_QUEUE_FIFO	Les tâches sont gérées dans leur ordre d'arrivée. C'est le mode par défaut à la création de la queue.

Le nombre maximal de queues de messages que gère le noyau est spécifié lors de la configuration du noyau.

Un cas particulier de gestion de queue de message arrive quand le message n'est constitué que d'un pointeur. Dans ce cas seul le pointeur est copié, ce qui est rapide, et l'information est stockée dans un tampon externe par l'application. L'adresse de ce tampon est transféré de l'émetteur au récepteur sans recopie du tampon lui même.

Pour utiliser cette gestion de pointeurs faut créer la queue avec l'indicateur AA_QUEUE_POINTER.

Pour faciliter ce type de gestion il est possible d'utiliser les pools de tampons.

1.17 Pools de tampons

Le pool de tampons est un mécanisme qui permet par exemple d'utiliser les queues de messages sans recopie des données.

Un pool de tampons est constitué d'une suite de tampons chaînés dans une liste, ce qui constitue la liste des tampons libres.

Lors de la création du pool de tampons (avec *aaBufferPoolCreate()*) l'utilisateur peut fournir l'adresse du bloc mémoire qui sera utilisé pour constituer les tampons, ce qui permet une allocation statique. Si l'adresse vaut NULL et que la gestion dynamique de mémoire est autorisée, le pool gère l'allocation de la mémoire ainsi que sa libération.

Exemple d'utilisation :

Quand une tâche a besoin d'un tampon, elle le demande avec *aaBufferPoolTake ()* au pool qui le retire de la liste des tampons libres. Ensuite la tâche peut l'utiliser : le remplir avec des données puis le placer dans une queue de messages en fournissant l'adresse du tampon à la queue.

La tâche qui lit la queue de messages reçoit l'adresse du tampon, utilise le contenu du tampon, puis le rend au pool avec *aaBufferPoolGive()*, qui va le mettre dans la liste des tampons libres.

Le nombre maximal de Pools de tampons que gère le noyau est spécifié lors de la configuration du noyau.

1.18 Board Support Package

Le BSP regroupe les fonctionnalités du noyau qui dépendent du système utilisé. Certaines fonctionnalités sont accessibles à l'utilisateur à travers des fonctions génériques telles que *bspGetTickRate()* ou *bspOutput()* par exemple.

Le BSP propose un accès à certaines ressources souvent utilisées :

- Les GPIO associées à des LEDs ou à des boutons. D'autres GPIO peuvent être ajoutées, pour le debug à l'oscilloscope par exemple.
- Un driver générique d'UART pour disposer d'une console.
- Configurer et interroger le tick système.

En plus de ces fonctions utilisateur, le BSP offre un support au noyau :

- Initialisation du matériel
- Support du tick système
- Support des interruptions
- Gestion du changement de contexte des tâches.

1.19 Debug

La mise à disposition d'outils pour faciliter le debug est une fonctionnalité importante d'un système embarqué.

AdAstra RTK propose quelque un de ces outils : une console, la tâche *logMes* et l'utilisation du signal SWO des ARM Cortex, un système de traces, une macro AA_ASSERT, et une gestion centralisée des erreurs.

1.19.1 Console

Une console est l'outil de base indispensable pour le debug. Il est possible d'utiliser n'importe quel UART/USART disponible dans le processeur pour cela.

La console est configurée dans *aacfg.h (AA_WITH_CONSOLE)*. En complément il est nécessaire de définir dans le fichier *uartbasic.c* du de la librairie des périphériques les UART/USART qui doivent être gérés par le noyau. La console est initialisée par le noyau, et le handle de l'UART correspondant est disponible dans la variable globale *aaConsoleUartHandle*.

Si une carte de développement est utilisée, il est souvent pratique d'utiliser comme console l'UART qui est utilisé par la sonde de programmation/debug.

1.19.2 aaLogMes

L'affichage de traces peut être un moyen efficace de debug. Cependant les traces ont tendance à modifier le timing de l'application et en ce sens elles sont intrusives.

Un système de trace peut intrusif est basé sur la tâche *idle* et la fonction *aaLogMes()*.

La fonction *aaLogMes()* a une syntaxe similaire à celle de *printf()*, mais au lieu de réaliser immédiatement la mise en forme du message, elle place les arguments dans une queue de messages. Cette fonction a aussi l'avantage de pouvoir être appelée par une fonction d'interruption.

Dans un second temps la tâche *idle* extrait les messages de la queue pour mettre en forme ces messages et les envoyer. Il est possible de choisir à quel périphérique seront envoyés les messages, le plus souvent un UART ou SWO.

Le timing de l'application est peu perturbé parce que :

- Le formatage des messages, qui peut être coûteux en temps, n'est pas réalisé par *aaLogMes()* qui de plus n'accède a aucun périphérique, ce qui lui évite d'être éventuellement bloquée.
- La tâche *idle* s'exécute avec une priorité plus basse que les tâches de l'application, et donc s'exécute pendant les instants laissés libres par l'application.
- L'utilisation de SWO est rapide et n'implique pas l'utilisation d'interruption comme un UART.

Les arguments de *aaLogMes()* ne sont pas utilisés au moment de l'appel de la fonction, mais plus tard par la tâche *idle*. Cela veut dire que les arguments doivent toujours être valides à ce moment là, ce qui exclu des pointeurs sur des données volatiles tels que des pointeurs ou des données alloués sur la pile.

Dans l'exemple suivant l'utilisation de *str* est invalide, car quand *idle* traitera le message *str* n'existera peut être plus. L'utilisation de *pStr* est valide car la valeur pointée par *pStr* ne changera pas.
L'utilisation de *arg* est valide parce que *arg* n'est pas un pointeur et la valeur passée sera toujours valide.

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

    pStr = "Hello" ;


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

La queue de messages utilisée par le système de trace a une profondeur limitée. Si l'application génère plus de messages que la tâche *idle* ne peut en traiter, la queue peut être pleine et dans ce cas des messages seront ignorés. Cela est signalé par le message « *logMes lost:xx* » où xx est le nombre de messages perdus.

Le débordement de la queue de messages peut par exemple être provoqué par une tâche qui génère des messages dans une boucle permanente : qui n'est pas suspendue de temps en temps par l'attente d'une I/O, d'un sémaphore ou d'un mutex, ou par *aaTaskDelay()*.

Dans les noyaux AdAstra-RTK antérieurs à la version 1.10, le formatage des messages logMes était assuré par une tache dédiée tLogM. Maintenant le formatage est assuré par la tâche idle, ce qui économise des ressources.

1.19.3 La surveillance des piles

Lors de la mise au point d'une application, les problèmes dus au piles des tâches peuvent être ardus à détecter. Le noyau et le BSP doivent faciliter la détection d'erreurs dans la gestion des piles.

AdAstra-RTK permet de surveiller les piles des tâches, c'est à dire de détecter si elles s'approchent ou dépassent leurs limites. Pour valider la surveillance de la pile d'une tâche il faut fournir l'indicateur AA_FLAG_STACKCHECK lors de la création de la tâche.

Lorsque l'indicateur AA_FLAG_STACKCHECK est fourni, la pile est initialisée avec des mots de valeur connue. La surveillance est réalisée par le noyau lors de chaque changement de contexte : (lorsque la tâche est inactivée) en réalisant 2 tests :

- Vérifier que le pointeur courant dans la pile n'est pas supérieur à la limite. Cela permet de détecter un dépassement avéré de la pile.
- Compare les mots 7 et 8 de la pile avec les valeurs de remplissage de la pile. Si les mots ont une valeur différente, cela indique que l'utilisation de la pile est très proche du maximum.

Quand un des tests est positif, le noyau appelle la fonction *aaUserNotify()* qui doit être définie par l'utilisateur (une implémentation par défaut se trouve dans *userInitTask.c*). Les arguments de cette fonction permettent de savoir quelle tâche est impactée et quel test est signalé.

L'utilisateur peut utiliser à tout moment la fonction *aaTaskCheckStack()* afin de déterminer l'usage de la pile dont la surveillance a été demandée lors de la création de la tâche.

La pile système utilisée par les interruptions est indépendante de toute tâche. Elle a une fonction dédiée pour permettre sa surveillance : *bspMainStackCheck()* qui permet de connaître le nombre de mots inutilisés dans cette pile.

La mémoire utilisée par la pile système est systématiquement initialisée à une valeur connue par le BSP, en utilisant les limites de la pile fournies par le linker.

1.19.4 SWO

La broche Single Wire Output (SWO) est une caractéristique des ARM Cortex. Elle peut être utilisé de plusieurs façons. AdAstra RTK l'utilise comme une sortie de type UART très rapide (plusieurs dizaines de MBits/s), en association avec le mécanisme ITM, en optimisant le débit utile.

Le mécanisme ITM de l'ARM permet de transmettre des mots de 8, 16 ou 32 bits par l'intermédiaire de 32 canaux (appelés stimulus par ARM) vers la broche SWO. Chaque canal fait précéder le mot émis d'un entête qui permet au récepteur

d'identifier le canal émetteur. Le canal 0 a la particularité d'avoir des entêtes dont la valeur est inférieure à 0x04, et donc non affichés par un terminal.

Si les messages générés par *aaLogMes()* sont envoyés vers SWO par le canal 0 de l'ITM on dispose d'un mécanisme de trace lisibles, à fort débit et très peu intrusif pour l'application.

Le mécanisme ITM/SWO n'est disponible qu'en mode debug. C'est pour cela que par défaut le BSP initialise le mécanisme ITM et y dirige les messages générés par *aaLogMes()* lorsque DEBUG est défini.

Pour afficher les traces, il faut disposer d'un récepteur adapté. Il est possible par exemple d'utiliser le câble C232HM de FTDI, qui permet d'obtenir l'équivalent sur USB d'un UART à 8 ou 12 Mbits/s.

Le BSP fournit quelques fonctions pour :

- Initialiser le mécanisme SWO : *swoInit()*.
- Emettre des caractères ou des mots : *swoSend8()*, *swoSend16()* et *swoSend32()*.
- Emettre une chaîne de caractères terminée par un 0 : *swoPutStr()*.
- Utiliser la sortie standard pour émettre un caractère vers le canal 0: *swoSend()*, qui est utilisée par exemple pour diriger les messages de *aaLogMes()* vers SWO.

1.19.5 La macro *AA_ASSERT*

La macro *AA_ASSERT* est définie dans *aacfg.h*, si *AA_WITH_DEBUG* est défini à 1. Son prototype est :

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

Si expression est fausse (vaut 0), la fonction *bspAssertFailed()* définie par le BSP est appelée. Cette fonction a un comportement différent suivant la présence ou non d'un débugger :

- Si l'application a été lancée par un debugger, l'application s'arrête dans le débugger, ce qui permet d'inspecter la cause de l'arrêt.
- Si l'application n'a pas été lancée par un débugger, elle entre dans une boucle infinie après avoir inhibé les interruptions.

Remarque : les assertions de la HAL arrivent au même endroit.

La fonction *bspAssertFailed()* est définie avec l'attribut « weak » par le BSP. Cela permet à l'application de la redéfinir, et de lui affecter un comportement différent. Par exemple allumer une LED en cas d'erreur, ce qui permet d'être alerté même s'il n'y a pas de débugger actif.

1.19.6 La centralisation des erreurs

La gestion des erreurs est souvent une tâche difficile. Dans le code des applications il faut tester systématiquement le code de retour des fonctions appelées, et en cas d'erreur, souvent faire remonter l'erreur de plusieurs niveaux d'imbrication de fonctions, puis prendre une décision pour gérer cette erreur.

Cela génère une quantité de code importante, qu'il faut vérifier et tester. De plus il faut relire soigneusement le code pour s'assurer que toutes les valeurs de retour sont testées, ce qui est fastidieux, sans garantie d'exhaustivité, même si des options de compilation peuvent aider.

Pour alléger et systématiser ces procédures, le noyau AdAstra utilise une gestion centralisée des erreurs : Toutes les erreurs relevées par le noyau provoquent un appel à la fonction *bspErrorNotify()*. Cette fonction est définie dans le BSP avec un comportement par défaut, mais elle peut être redéfinie par l'utilisateur pour adapter son comportement à des cas particuliers.

Le fait que toutes les erreurs soient dirigées vers cette fonction garanti qu'aucune ne peut être oubliée, et qu'elles seront toutes gérées, sans imposer de contrainte au code des applications. Si la gestion centralisée est utilisée, cela permet à l'application de ne plus à avoir à tester les valeurs de retour des fonctions du noyau, puisqu'en cas d'erreur fatale la fonction de gestion des erreurs est appelée. De plus en mode debug cette fonction appelle le débugger : l'utilisateur n'a pas besoin de placer de point d'arrêt pour trapper ces erreurs.

La gestion centralisée des erreurs est configurée dans *aacfg.h*, en définissant AA_WITH_ERRORCHECK avec une des valeurs suivantes :

AA_WITH_ERRORCHECK_NONE

Il n'y a pas de gestion centralisée : les erreurs ne sont pas rapportées à *bspErrorNotify*(), les contrôles d'assertion ne sont pas réalisés.
Cette option est à réserver à des cas très particuliers.

AA_WITH_ERRORCHECK_ASSERT

En cas d'erreur l'application génère un point d'arrêt et appelle le débugger à partir de l'endroit où l'erreur a été détectée, ce qui permet de localiser immédiatement l'erreur.
Cette option est à réserver à la phase de développement, pendant laquelle le débugger et utilisé. En cas d'erreur alors que le débugger n'est pas utilisé le comportement n'est pas garanti (par exemple génération d'une « HardFault Exception »).

AA_WITH_ERRORHECK_NOTIFY

C'est le cas le plus général : la fonction *bspErrorNotify*() est appelée, et en cas d'utilisation du débugger génère un point d'arrêt. Si le débugger n'est pas utilisé elle gère son retour en fonction des flags associés au numéro d'erreur. Si le retour est interdit elle entre dans une boucle infinie après avoir inhibé les interruptions.

Pour centraliser les erreurs le fichier *aaerror.h* définit les numéros d'erreurs, des flags, et des macros.

Toutes les erreurs ont un numéro différent, et sont associées à des flags pour en configurer le traitement. Ces flags sont :

AA_ERROR_FATAL_FLAG

L'erreur est dite fatale : normalement l'application n'est plus en mesure de continuer à s'exécuter si une telle erreur survient et que l'application n'en tient pas compte.

AA_ERROR_NORETURN_FLAG

Après cette erreur le fonction *bspErrorNotify*() ne doit pas revenir.

AA_ERROR_FORCERETURN_FLAG

Après cette erreur le fonction *bspErrorNotify*() peut revenir, même si le flag AA_ERROR_NORETURN_FLAG est aussi présent. Autrement dit AA_ERROR_FORCERETURN_FLAG est prioritaire sur AA_ERROR_NORETURN_FLAG

Ce flag permet après que *bspErrorNotify*() ait signalé l'erreur de laisser l'application la gérer.

Les macros utilisées pour la gestion centralisée des erreurs permettent plusieurs opérations :

AA_ERRORNOTIFY(errorNumber)

Cette macro est la base de la centralisation des erreurs. Elle gère le signalement de l'erreur en fonction de la valeur d'AA_WITH_ERRORCHECK : point d'arrêt du débugger ou appel de *bspErrorNotify*().

Cette macro utilise systématiquement AA_ERROR_NORETURN_FLAG, par conséquent la fonction *bspErrorNotify*() ne doit pas revenir.

Cependant s'il est nécessaire que l'application continue après le signalement de cette erreur l'indicateur AA_ERROR_FORCERETURN_FLAG peut être utilisé.

AA_ERRORASSERT(test, errorNumber)

Cette macro réalise un test et si le résultat du test est faux, appelle *AA_ERRORNOTIFY(errorNumber)*.

Dans ce cas d'utilisation la fonction *bspErrorNotify*() ne doit pas revenir, il ne faut donc pas utiliser AA_ERROR_FORCERETURN_FLAG.

AA_ERRORCHECK(test, returnValue, errorNumber)

Cette macro permet de réaliser un test similaire à *AA_ERRORASSERT*, et la fonction *bspErrorNotify*() doit revenir. Après le signalement de l'erreur un retour à la fonction appelante avec la valeur de retour *returnValue* est exécuté. Pour cela elle associe systématiquement AA_ERROR_FORCERETURN_FLAG à *errorNumber*.

La macro AA_ERRORCHECK a le pseudo code suivant :

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

Il est parfois nécessaire d'insérer du code dans le traitement de l'erreur, avant le retour à la fonction appelante. Pour cela on peut utiliser le pseudo code suivant :

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

     AA_ERRORNOTIFY ((errorNumber) | AA_ERROR_FORCERETURN_FLAG);

     // User code

     return returnValue ;
}
```

Des numéros d'erreurs sont réservés pour l'application dans le fichier *aaerrors.h*. L'application peut donc bénéficier de la centralisation des erreurs en utilisant ces numéros d'erreurs et les macros définis dans ce fichier.

1.19.7 Les traces.

Les traces sont décrites dans un chapitre dédié.

2 Traces

AdAstra propose un mécanisme de trace intégré au noyau. Ces traces permettent entre autre de suivre en temps réel les API du noyau utilisées par les tâches et les interruptions. Les traces sont donc plus adaptées à l'étude du comportement des applications.

2.1 Configuration des traces

Les traces sont autorisées globalement au moment de la compilation du noyau par la définition de *AA_WITH_TRACE* dans *aacfg.h* :

AA_WITH_TRACE = 0 Aucune trace n'est générée, ce qui permet d'obtenir le noyau en mode release.

AA_WITH_TRACE = 1 Les traces sont générées.

Quand *AA_WITH_TRACE* vaut 1 il est possible de choisir quelles traces seront générées en définissant à 0 ou à 1 les diverses constantes *AA_WITH_T_xxx*. Par exemple :

```
#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
```

- les tâches qui entrent en attente d'un événement provenant d'un driver seront tracées, et le paramètre de la trace est l'identificateur de la tâche.
- Les interruptions ne seront pas tracées, et le paramètre de ces traces est le numéro d'interruption.

Les traces sont nombreuses, plus de 70, et certaines peuvent arriver avec une fréquence élevée. Il est recommandé de ne sélectionner que les traces nécessaires.

2.2 Validation des traces.

Par défaut les traces ne sont pas validées. Pour le faire il faut utiliser la fonction *aaTraceEnable(1)*. La fonction *aaTraceEnable(0)* inhibe les traces.

Cela permet de générer les traces exactement lorsque c'est nécessaire.

Lors de la validation des traces, des informations sur les tâches déjà créées sont émises, ce qui permet une l'exploitation plus facile de ces traces. Cette validation peut donc prendre un peu plus de temps que les autres traces.

2.3 Implémentation des traces.

Lors de la conception du mécanisme de trace, il faut faire des compromis :

- Les traces doivent être le plus extensives possibles, afin de fournir les informations nécessaires à la validation de l'application ou à la résolution des problèmes rencontrés.
- Les traces doivent être le moins intrusives possible : changer le moins possible le comportement temporel du noyau par exemple.
- Utilisation avec un débugger, ou en opérationnel.
- Les moyens traditionnels employés pour enregistrer les traces sont soit un tableau en mémoire, soit l'émission en temps réel.

Le tableau en mémoire a forcément une taille limitée, mais modifie très peu le comportement temporel du noyau.

L'émission en temps réel permet des enregistrements de longue durée, mais est plus ou moins intrusif suivant le périphérique utilisé.

Il n'y a donc pas de solution parfaite. Par conséquent le mécanisme est conçu en deux parties :

- Les traces appellent des fonctions prédéterminées, avec des paramètres.
- Ces fonctions n'existe pas en tant que telles : dans le fichier *aabase.h* ces fonction sont définies comme des macros qui doivent être implémentées pour réaliser la trace.

Ainsi l'utilisateur peut implémenter une mémorisation ou une transmission des traces suivant le protocole qu'il choisira.

2.4 Traces utilisateur.

Le jeu de macros et fonctions des traces peut être étendu par l'utilisateur pour transmettre des informations spécifiques.

Les traces suivantes sont prédéfinies et peuvent être utilisées librement par l'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)
```

Le libellé et le format d'affichage des traces utilisateur est configurable pour leur visualisation par le logiciel aaView.

2.5 Exemple.

Le fichier *aatrace.c* est un exemple de fonctions associées aux macros du fichier *aabase.h*. Les hypothèses qui ont été retenues pour concevoir ce fichier sont :

- Utilisation de la liaison SWO pour émettre les traces. C'est une liaison rapide (couramment 12 Mbits/s soit moins d'une micro seconde par octet) qui dispose d'une petite FIFO. L'inconvénient est qu'elle n'est utilisable que si elle est configurée par un débugger.
- Limitation au maximum du flot de données. Pour cela les identifiants des objets (tâches, sémaphores, …) sont limités à 8 bits. Par conséquent il n'est pas possible de tracer plus de 255 objets de chaque type, mais cela couvre l'immense majorité des applications sur le type de processeur visé.
- La datation des traces utilise la fonction *aaTsGet()* et utilise un mot de 32 bits. Avec cette fonction qui utilise le compteur de cycle de la CPU (sauf sur Cortex M0(+)), la datation est très précise, mais reboucle au bout d'un temps relativement court : L'application qui reçoit les trace doit en tenir compte.

Les traces utilisent le port de stimulus 1, ce qui est compatible avec l'implémentation SWO de la tâche *aaLogMes()* qui utilise le port de stimulus 0. Par conséquent la sortie SWO peut contenir un mélange de traces et de messages générés par *aaLogMes()*. Les traces sont prioritaires sur les messages de log.

Il est facile d'envisager de modifier ce fichier pour utiliser par exemple une liaison série, en émulant le protocole ITM, ce qui permettrait de pouvoir utiliser les traces sans débugger.

2.6 L'application aaVieew.

C'est une application console Microsoft Windows qui permet de recevoir les traces à partir d'une liaison séries et de les afficher en clair.

Le protocole utilisé est ITM, ce qui permet d'afficher les traces et les messages de log.

Les traces sont affichées en temps réel, mais la rapidité d'affichage peut gêner leur analyse. Dans ce cas plusieurs approches sont possibles :

- Si l'application gère les traces avec *aaTraceEnable()* les traces peuvent être arrêtées et analysées.
- Sinon il est possible de sélectionner les traces et de les copier dans un éditeur de texte tel que Notepad++ pour les sauver et les analyser en temps différé.

L'application aaView utilise par défaut le fichier de configuration aaView.ini. Celui ci permet de spécifier le numéro de la liaison série COM, son baud rate, la fréquence du MCU, et les formats d'affichage des traces utilisateur.

Exemple de fichier de configuration:

```
[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
```

La ligne de commande a la syntaxe suivante :

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

Elle permet de spécifier :

- le numéro de la liaison série COM,
- son baud rate
- la fréquence du MCU
- le chemin d'un fichier de configuration.

Les paramètres présents sur la ligne de commande ont priorité sur les paramètres du fichier de configuration.

Exemple de traces :

```
// 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 Ecrire une application

Ce chapitre explique les mécanismes d'initialisation du noyau, puis comment démarre l'exécution d'une application.

3.1 Configuration du noyau.

Le noyau est configuré avec le fichier *aacfg.h*, qui contient des définitions, dont les principales sont les suivantes:

AA_PRIO_COUNT	Le nombre de priorités que doit gérer le noyau. Ce nombre doit être compris entre 3 et 256. La performance est optimale avec un nombre au plus égal au nombre de bits d'un mot du processeur (par exemple 32 si le processeur gère des mots de 32 bits).
AA_TASK_MAX	Le nombre de tâches nécessaires à l'application. Ce nombre est indépendant du nombre de priorités gérées par le noyau.
AA_MUTEX_MAX	Le nombre de mutex nécessaires à l'application.
AA_SEM_MAX	Le nombre de sémaphores nécessaires à l'application.
AA_TIMER_MAX	Le nombre de timers nécessaires à l'application.
AA_QUEUE_MAX	Le nombre de queues nécessaires à l'application.
AA_BUFPOOL_MAX	Le nombre de pool de tampons nécessaire à l'application.
AA_WITH_LOGMES et AA_LOGMES_MAXBUF	La configuration de *aaLogMes()*.
AA_INIT_xxx	Les paramètres nécessaires pour créer la 1ère tâche de l'application.
AA_WITH_CONSOLE	Permet de définir les caractéristiques de la sortie standard de aaPrintf().

Il y a d'autres définitions concernant par exemple la gestion dynamique du tas, l'impression de flottants, l'utilisation de newlib… Se reporter au ficher *aacfg.h*.

Les définitions suivantes paramètrent les fonctions de mesure et de debug :

AA_WITH_ARGCHECK	Insère des tests sur la validité des paramètres des principales fonctions : mutex, sémaphore, queue, pool buffer…
AA_WITH_DEBUG	Valide les contrôles AA_ASSERT dans le noyau et dans l'application.
AA_WITH_TASKSTAT	Calcule le temps d'utilisation du CPU par chaque tâche.
AA_WITH_CRITICALSTAT	Calcule la durée maximale d'une section critique (non implémenté)

L'utilisation de ces fonctionnalités très utiles est intrusive : elles peuvent perturber le timing des tâches et dégrader dans une certaine mesure la performance du noyau.

Les définitions suivantes paramètrent la gestion dynamique de la mémoire. Voir aussi Gestion de mémoire dynamique.

AA_WITH_TLSF	Si défini à 1 indique que l'algorithme TLSF est implémenté.
AA_WITH_MALLOC_TLSF	Si défini à 1 indique que la gestion dynamique de la mémoire utilise l'algorithme TLSF.
AA_WITH_REALLOC	Si défini à 1 indique que la fonctionnalité realloc est implémentée. Cette fonctionnalité est mis en option car elle occupe beaucoup de code et qu'elle est rarement utile dans un système embarqué.
AA_WITH_MEMBLOCK	Si défini à 1 indique que l'allocation par bloc est implémentée.
AA_WITH_MALLOC_BLOC	Si défini à 1 indique que la gestion dynamique de la mémoire utilise l'algorithme d'allocation par bloc
AA_WITH_MALLOC_BLOC_ERRORFREE	Si défini à 1 indique qu'une erreur fatale doit être déclenchée si *aaFree()* est utilisée, si défini à 0 indique que *aaFree()* ne fait rien.

Il y a d'autres définitions concernant par exemple l'impression de flottants, l'utilisation de newlib… Se reporter au ficher *aacfg.h*.

Gestion centralisée des erreurs fatales : voir §1.19.6 « La centralisation des erreurs ».

La configuration du tas.

AA_WITH_USERHEAP	Défini à 0 indique que le tas occupe toute la mémoire disponible entre la section BSS et la pile système. Cela est défini par le script du linker. Défini à 1 indique que l'utilisateur veut définir l'emplacement et la taille du tas avec les définitions AA_HEAP_BEGIN et AA_HEAP_SIZE.

Configuration des traces : voir Traces

3.2 Initialisation du système.

Le mécanisme d'initialisation du système est ici expliqué pour un système ARM Cortex-M.

Plusieurs fichiers sont utilisés :

vectors_xxx.c	Ce fichier contient la table des vecteurs d'interruption et d'exception du processeur. En particulier les 2 vecteurs utilisés pour le démarrage : Le vecteur 0 contient l'adresse de la pile MSP (main stack) Le vecteur 1 contient l'adresse de la première fonction à appeler : *_start()*. Ces informations sont fournies par le script du linker.
startup.c	Contient la fonction *_start()* qui est la première appelée lors du démarrage du processeur. Après l'initialisation elle appelle *bspMain()*.
bsp.c	Contient les fonctions appelées depuis *_start()* pour l'initialisation minimale du matériel : clock, FPU…
system_stm32xxx	Fichier fourni par le producteur du processeur avec la fonction *SystemInit()*.

Le processus d'initialisation du système est le suivant :

- Au démarrage du processeur le contrôle est transféré à la fonction dont l'adresse est dans le vecteur d'interruption 1, qui est *_start()* qui :
  - Appelle *bspSystemInit_()* : Appelle *SystemInit()*, et positionne la table des vecteurs au bon endroit (en RAM si nécessaire). Réalise les éventuelles opérations à réaliser au plus tôt après le démarrage du processeur
  - Utilise les informations du linker pour initialiser les sections de données initialisées et non initialisées (BSS) dans les différentes zones de RAM.
  - Appelle *bspHardwareInit_()* : configure FPU, horloge système, NVIC_SetPriorityGrouping.
  - Appelle *bspMain()* qui sera sans retour.
- *bspMain()*.
  - Réalise quelques initialisations du matériel et du BSP.
  - Appelle *aaMain()* qui est le point d'entrée du noyau et qui est sans retour.

Configuration FPU matérielle :

Dans la configuration d'Eclipse C/C++ Build / Settings / Target Processor sélectionner :

- Float ABI : FP instructions (hard)
- FPU type : fpv4-sp-d16 (Si Cortx-M4)

Cela permet au fichier *core_cm4.h* de générer la constant __FPU_USED qui est elle même utilisée par bsp.c pour gérer le FPU.

Configuration FPU logicielle, quand on ne veut pas utiliser de nombre réel :

Dans la configuration d'Eclipse C/C++ Build / Settings / Target Processor sélectionner :

- Float ABI : Toolchain Default
- Pour imprimer des nombres flottants avec *printf()* de nanolib, ajouter « -u _printf_float » à la ligne de commande du linker. aaPrintf() peut aussi afficher des valeurs flottantes.

3.3 Initialisation du noyau.

L'initialisation du noyau est réalisée par *aaMain()* qui se trouve dans *aaMain.c*.

La séquence des opérations d'initialisation est la suivante :

- Si *AA_USART_CONSOLE* est défini dans *aacfg.h* la liaison console UART est initialisée, les sorties de *aaPrintf()* et de *aaLogMes()* sont dirigées vers cette console.
  Remarque : si DEBUG est défini, le BSP dirige les sorties de *aaLogMes()* vers la sortie SWO.
- Initialise AdAstra RTK en appelant *aaInit()*.
- Affiche la bannière définie dans le BSP avec *bspBanner()*, si BSP_WITH_BANNER est défini à 1 dans *bspcfg.h*.
- Initialise l'allocation de mémoire dynamique suivant ce qui est configuré
- Crée la tâche « idle » de priorité 0.
- Initialise les composants annexes : queues, timers, …
- Créée la première tâche avec la configuration définie dans *aacfg.h* par les constantes AA_INIT_xxx.
- Démarre le noyau en appelant *aaStart()*.

La première tâche continue les initialisations du noyau qui nécessitent que le noyau soit démarré, puis appelle la fonction *userInitTask()* qui doit être définie par l'utilisateur.

3.4 Initialisation de l'application.

La première tâche exécutée par le noyau appelle la fonction *userInitTask()*, écrite par l'utilisateur, qui se trouve donc être aussi la première tâche de l'utilisateur. Sa configuration est définie dans *aacfg.h* par les constantes *AA_INIT_xxx*.

Cette fonction est appelée après l'initialisation complète du système, l'ensemble des ressources est donc utilisable.

Dans la plupart des cas cette fonction réalise :

- Un changement de priorité de la tâche pour qu'elle corresponde au besoin de l'application (si la priorité *aacfg.h*).
- L'initialisation de l'environnement de l'application.
- La création des autres tâches de l'application.

La fonction *userInitTask()* a le prototype d'une fonction de tâche, autrement dit :

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

La taille de la pile de la première tâche est définie par *AA_INIT_STACK_SIZE*. Cette pile est allouée dynamiquement si *AA_WITH_MALLOC* est défini à 1 et AA_INIT_STACK_STATIC est défini à 0. Sinon un tableau statique est automatiquement alloué.

3.5 STMicroelectronics Hardware Abstraction Layer

La HAL fournie par STMicroelectronics n'est pas conçue pour coexister avec un RTOS.

Le principal problème est que la HAL configure le timer systick, alors que cela est géré par le noyau en temps voulu. STM recommande d'utiliser un autre timer dédié à la HAL. Mais il est inutile d'utiliser deux timers pour faire la même chose deux fois.

Pour contourner ce problème et utiliser la HAL, l'utilisateur doit:

- Ajoutez le symbole de préprocesseur USE_HAL_DRIVER. Cela inclut les fonctions d'initialisation de HAL et informe le BSP d'appeler *HAL_IncTick()*.
- Ne pas utiliser *HAL_Init*() ni *HAL_SYSTICK_Config()*. Le noyau effectue les initialisations appropriées.
- Appeler *HAL_MspInit()* dans la première tâche de l'utilisateur, avant d'appeler toute autre fonction de la HAL. Tant que le noyau n'est pas démarré le timer systick est arrêté. Le fait d'initialiser la HAL une fois que le noyau a démarré permet d'obtenir un fonctionnement correct des boucles d'attentes des timeout utilisées par la HAL.
- Ne pas utiliser *HAL_SuspendTick()* / *HAL_ResumeTick()*. Cela perturbe le fonctionnement du noyau temps réel. Ces fonctions sont redéfinies par le noyau avec un AA_ASSERT (0) pour avertir l'utilisateur en cas d'utilisation.

Si la HAL n'est pas utilisée il faut supprimer le symbole USE_HAL_DRIVER du préprocesseur.

4 Manuel de référence

4.1 Divers

aaVersion

```
uint32_t      aaVersion      (void)
```

Description TOC §↑

Cette fonction retourne la version du noyau sous la forme de 2 valeurs dans un entier 0xVVVVRRRR :

- VVVV Version du noyau.
- RRRR Révision du noyau.

Par exemple

- 0x00010000 indique la version 1.00x00020008 indique 2.8

Valeur de retour

La version du noyau

4.2 Gestion des tâches

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 §↑

Cette fonction permet de créer une tâche

prio	La priorité de la tâche entre 0 et AA_PRIO_COUNT-1.La priorité 0 est réservée à la tâche idle, et AA_PRIO_COUNT es le nombre de niveaux de priorité géré par le noyau.
pName	Un pointeur sur une chaîne de caractère qui est le nom de la tâche. La taille maximale de la chaîne est AA_TASK_NAME_SIZE, y compris le 0 final.
pEntry	Un pointeur sur la fonction que doit exécuter la tâche.
arg	L'argument de la fonction pointée par pEntry.
pStack	Pointeur sur la zone mémoire qui servira de pile à la tâche. Si ce pointeur est non NULL l'utilisateur se charge de l'allocation de la pile au moment de la création de la tâche, et de la libération de la pile lors de la destruction de la tâche. Cela permet d'avoir une allocation complètement statique Si le pointeur de pile vaut NULL et que l'allocation de mémoire dynamique est autorisée, alors le noyau se charge de la gestion dynamique de la pile. La pile doit être alignée sur un multiple de 8 octets (ARM).
stackSize	La taille de la zone mémoire pointée par pStack, en nombre de mots de type *bspStackType_t*.
flags	Des indicateurs parmi : AA_FLAG_STACKCHECK — Valide la surveillance de la pile pour cette tâche. AA_FLAG_SUSPENDED — La tâche est créée dans l'état suspendu. Il faut utiliser *aaTaskResume()* pour lancer son exécution.
pTaskId	Un pointeur sur une variable qui contiendra l'identificateur sur la tâche créée au retour de la fonction.

La tâche créée est immédiatement active : si elle a une priorité supérieure à la tâche courante, celle-ci sera immédiatement préemptée par la nouvelle tâche.

Surveillance de la pile

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide : priorité ou pile.
AA_EDEPLETED	Il n'y a plus de descripteur de tâche disponible.
AA_EMEMORY	Erreur d'allocation mémoire de la pile.

aaTaskDelete

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

Description TOC §↑

Cette fonction permet de terminer une tâche, de libérer ses ressources, et de placer son descripteur dans la file des descripteurs de tâches libres.

taskId	L'identificateur de la tâche à terminer. Si l'identificateur est AA_SELFTASKID, alors la tâche appelante est terminée.

Si une tâche sort de la fonction spécifiée lors de la création par return, ou en ayant atteint la fin de la fonction alors *aaTaskDelete(AA_SELFTASKID)* est implicitement appelée.

Si une tâche terminée utilise une pile statique allouée par l'utilisateur, alors la callback *aaUserReleaseStack()* est appelée. Cela donne l'opportunité à l'utilisateur de savoir qu'un bloc mémoire se libère et de le gérer en conséquence.

Si une tâche est terminée alors qu'elle possède un mutex ou un sémaphore, ceux-ci restent en l'état, ce qui peut provoquer un comportement imprévisible de l'application. La même chose peut arriver si la tâche terminée était en attente dans un driver (état *aaIoWaitingState*).

Si une tâche est détruite par une autre tâche, la pile et le descripteur de tâche sont libérés immédiatement. Si la tâche se détruit elle même, cela n'est pas possible, et les libérations seront faites ultérieurement par la tâche idle.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide : priorité ou pile.

AA_ESTATE	La tâche est dans un état inconnu qui ne permet pas de la terminer.

aaTaskIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de tâche fourni est valide :

- Il correspond à une tâche
- La tâche existe (a été créée et n'a pas été détruite).

taskId	L'identificateur de tâche à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaTaskGetBasePriority

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

Description TOC §↑

Cette fonction permet de connaître la priorité de base de la tâche. La priorité de base est celle qui est spécifiée lors de la création de la tâche, et qui est utilisée quand le mécanisme d'héritage de priorité n'est pas actif.

taskId	L'identificateur de la tâche dont il faut obtenir la priorité. Si l'identificateur est AA_SELFTASKID, alors il s'agit de la tâche appelante.
pBasePriority	Un pointeur sur une variable dans laquelle la valeur de la priorité sera placée.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.

aaTaskGetRealPriority

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

Description TOC §↑

Cette fonction permet de connaître la priorité actuelle de la tâche. La priorité actuelle peur être la priorité de base ou une priorité héritée.

taskId	L'identificateur de la tâche dont il faut obtenir la priorité. Si l'identificateur est AA_SELFTASKID, alors il s'agit de la tâche appelante.
pRealPriority	Un pointeur sur une variable dans laquelle la valeur de la priorité sera placée.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.

aaTaskSetPriority

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

Description TOC §↑

Cette fonction permet de changer la priorité de base de la tâche. La priorité de base est celle qui est spécifiée lors de la création de la tâche, et est utilisée quand le mécanisme d'héritage de priorité n'est pas actif.

taskId	L'identificateur de la tâche dont il faut changer la priorité. Si l'identificateur est AA_SELFTASKID, alors il s'agit de la tâche appelante.
newBasePriority	La nouvelle priorité de la tâche entre 1 et AA_PRIO_COUNT-1

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.

aaTaskSuspend

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

Description TOC §↑

La tâche spécifiée est placé dans l'état suspendu : même si elle est prête à s'exécuter elle ne le fait pas. Elle ne pourra reprendre son activité qu'après avoir été réactivée par *aaTaskResume()*.

Si une tâche est suspendue pendant qu'elle attend une ressource (mutex, sémaphore, …) elle ne sera effectivement suspendue qu'après avoir obtenu la ressource, qui devient alors indisponible pour les autres tâches pendant tout le temps de suspension de la tâche ayant obtenu la ressource.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.

aaTaskResume

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

Description TOC §↑

La tâche spécifiée est réactivée si elle est dans l'état suspendu.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Argument invalide

aaTaskDelay

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

Description TOC §↑

La tâche courante est mise en état d'attente pour la durée spécifiée en nombre de ticks système. Cela provoque l'activation de la tâche active la plus prioritaire, s'il y en a une.

La tâche en attente ne consomme pas de CPU.

delay	La durée d'attente en ticks, de 0 à 0xFFFFFFFE. La valeur AA_INFINITE spécifie une durée infinie.

Valeur de retour

Aucune

aaTaskWaikeUp

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

Description TOC §↑

Interrompt l'attente d'une tâche qui a appelé *aaTaskDelay()*. La tâche spécifiée est immédiatement placée dans l'état prêt, et ne peut pas savoir qu'elle a été réveillée plus tôt que prévu.

Cette fonction ne peut pas interrompre l'attente d'une tâche en attente d'un événement : signal, sémaphore, mutex, queue, etc.

Valeur de retour

Aucune

aaTaskSelfId

```
aaTaskId_t    aaTaskSelfId   (void)
```

Description TOC §↑

Permet d'obtenir l'identificateur de la tâche appelante.

Appelée à partir d'une fonction d'interruption elle retourne l'identificateur de la tâche interrompue (la tâche « courante »).

Valeur de retour

L'identificateur de la tâche appelante.

aaTaskYield

void aaTaskYield (void)

Description TOC §↑

Permet à la tâche courante d'abandonner son droit à exécution à une autre tâche, qui a donc la même priorité. La tâche courante est placée en fin de liste des tâches prêtes avec cette priorité.

S'il n'y a pas d'autre tâche avec la même priorité, la tâche courante continue son exécution.

Cette fonction permet donc de gérer le temps d'exécution de manière coopérative entre tâches de même priorité.

Valeur de retour

Aucune

aaTaskGetName

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

Description TOC §↑

Permet d'obtenir un pointeur sur le nom de la tâche spécifiée.

Exemple, pour obtenir le nom de la tâche courante :

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

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.

aaTaskCheckStack

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

Description TOC §↑

Permet d'obtenir la place non utilisée dans la pile de la tâche spécifiée, en nombre de mots de type *bspStackType_t*.

Il faut que l'indicateur AA_FLAG_STACKCHECK ait été spécifié lors de la création de la tâche, ce qui a provoqué l'initialisation de la pile de la tâche avec un marqueur.

Cette fonction, dont le temps d'exécution peut être important, n'utilise pas de section critique, par conséquent il ne faut pas que la tâche spécifiée soit détruite pendant l'exécution de cette fonction.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Argument invalide.
AA_ESTATE	AA_FLAG_STACKCHECK n'a pas été spécifié pour cette tâche.

aaTaskInfo

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

Description TOC §↑

Permet d'obtenir des informations et statistiques sur les tâches.

pInfo	adresse d'un tableau de structures qui sera rempli par la fonction
size	Nombre d'éléments dans la structure pInfo. Il est utile que ce nombre soit au moins égal au nombre de tâches créées.
pReturnSize	Nombre d'éléments de pInfo utilisés par la fonction.
pCpuTotal	Adresse d'une variables dans laquelle sera placé le temps en micro secondes utilisé par les tâches depuis le dernier appel à *aaTaskStatClear()*.
pCriticalUsage	Adresse d'une variables dans laquelle sera placé le temps passé dans la section critique la plus longue depuis le dernier appel à *aaTaskStatClear()*.
flags	non utilisé.

La structure des informations est la suivante :

```
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 ;
```

Les noms des états des tâches sont disponibles dans le tableau *aaTaskStateName[]*. Par exemple :

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

Cette fonction utilise une section critique afin que l'état de toutes les tâches soit cohérent. Si le nombre de tâches est important, la durée de la section critique peut être importante.

Valeur de retour

AA_ENONE Pas d'erreur.

aaTaskStatClear

```
void      aaTaskStatClear    (void)
```

Description TOC §↑

Initialise à 0 les statistiques des tâches qui peuvent être obtenues avec *aaTaskInfo()*.

Valeur de retour

Aucune

4.3 Mutex

aaMutexCreate

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

Description TOC §↑

Permet de créer un mutex. Ce mutex permet d'assurer un accès exclusif à une ressource telle qu'un périphérique ou une structure de données.

Il a des caractéristiques particulières :

- Il peut être acquis de façon récursive : la même tâche peut acquérir plusieurs fois le même mutex, puis le libérer autant de fois qu'elle l'a acquis.
- Le mutex utilise un algorithme d'héritage de priorité des tâches, afin d'éviter le phénomène d'inversion de priorité illimité, qui a pour effet qu'une tâche de basse priorité empêche une autre tâche de plus haute priorité de s'exécuter.
- Si une tâche est détruite alors qu'elle détient un mutex, celui-ci ne sera pas rendu, et la ressource dont l'accès est protégé reste verrouillée.
- Si une tâche est suspendue alors qu'elle détient un mutex, le mutex reste acquis par la tâche.

pMutexId	Pointeur sur une variable qui recevra l'identificateur du mutex.

Valeur de retour

AA_ENONE	Pas d'erreur, le mutex est créé.
AA_EDEPLETED	Il n'y a plus de descripteur de mutex disponible.
AA_ENOTALLOWED	Non autorisé à partir d'une fonction d'interruption

aaMutexDelete

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

Description TOC §↑

Permet de détruire un mutex, et de placer son descripteur dans la file des descripteurs de mutex libres.

Le mutex doit être libre pour pouvoir être détruit.

Valeur de retour

AA_ENONE	Pas d'erreur, le mutex est détruit
AA_ESTATE	Le mutex n'est pas en état d'être détruit (acquis par une tâche).

AA_ENOTALLOWED	Non autorisé à partir d'une fonction d'interruption
AA_EARG	Identificateur de mutex invalide.

aaMutexIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de mutex fourni est valide :

- Il correspond à un mutex
- Le mutex existe (a été créé et n'a pas été détruit).

mutexId L'identificateur de mutex à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaMutexTake

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

Description TOC §↑

Cette fonction permet d'acquérir le mutex. Si le mutex est déjà acquis par une autre tâche, la tâche appelante est suspendue jusqu'à obtention du mutex ou échéance du timeout.

mutexId	L'identificateur du mutex
timeOut	La valeur du timeout en tick système, de 0 (pas d'attente) à 0xFFFFFFFE ou AA_INFINITE.

Cette fonction ne peut pas être utilisée par une fonction d'interruption.

Valeur de retour

AA_ENONE	Pas d'erreur, le mutex est acquis
AA_EARG	Argument invalide.
AA_ENOTALLOWED	Opération non autorisée, à partir d'une interruption par exemple.
AA_EWOULDBLOCK	Mutex non acquis, timeout spécifié égal à 0.

AA_ETIMEOUT	Mutex non acquis, retour sur timeout.
AA_EFAIL	Overflow du compteur du mutex.

aaMutexTryTake

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

Description TOC §↑

Tente d'obtenir le mutex, avec un timeout de 0.

C'est une macro équivalente à : *aaMutexTake (mutexId, 0)*.

Cette macro ne peut pas être utilisée par une fonction d'interruption.

Valeur de retour

Les mêmes que *aaMutexTake()*.

aaMutexGive

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

Description TOC §↑

Libère le mutex spécifié. Si le compteur de récursivité tombe à 0, le mutex est effectivement libéré, et si une autre tâche est en attente de ce mutex elle l'obtient.

Si une autre tâche obtient le mutex libéré et que cette tâche a une priorité réelle supérieure à la priorité de base de la tâche appelante, elle préempte immédiatement la tâche appelante.

Cette fonction ne peut pas être utilisée par une fonction d'interruption.

Valeur de retour

AA_ENONE	Pas d'erreur, le mutex est acquis
AA_EARG	Argument invalide.
AA_ENOTALLOWED	Opération non autorisée, à partir d'une interruption par exemple.
AA_ESTATE	Le mutex n'est pas en état d'être libéré : acquis par une autre tâche, déjà libéré.

4.4 Sémaphore

aaSemCreate

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

Description TOC §↑

Permet de créer un sémaphore à compteur, et d'initialiser sa valeur. Un sémaphore gère toujours les tâches en attente par ordre de priorité décroissante.

Si une tâche est détruite alors qu'elle détient un sémaphore, celui-ci ne sera pas rendu, et la ressource dont l'accès est protégé reste verrouillée.

Si une tâche est suspendue alors qu'elle détient un sémaphore, le sémaphore reste acquis par la tâche.

count	La valeur initiale du compteur du sémaphore (de -32768 à +32767)
pSemId	Un pointeur sur la variable qui contiendra l'identificateur du sémaphore créé.

Valeur de retour

AA_ENONE	Pas d'erreur, le sémaphore est créé.
AA_EDEPLETED	Il n'y a plus de descripteur de sémaphore disponible.

aaSemDelete

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

Description TOC §↑

Permet de détruire un sémaphore et de libérer ses ressources.

Si des tâches sont en attente du sémaphore, elles sont toutes libérées par un appel à *aaSemFlush()*.

semId	L'identificateur du sémaphore à détruire.

Valeur de retour

AA_ENONE	Pas d'erreur, le sémaphore est détruit.
AA_EARG	L'identificateur n'est pas un identificateur valide.

aaSemIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de sémaphore fourni est valide :

- Il correspond à un sémaphore
- Le sémaphore existe (a été créé et n'a pas été détruit).

`semId` L'identificateur de sémaphore à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaSemTake

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

Description TOC §↑

Permet d'acquérir le sémaphore dont l'identificateur est spécifié. Si le compteur du sémaphore est inférieur ou égal à 0 la tâche est mise en attente jusqu'à ce que le compteur devienne positif et qu'aucune autre tâche de priorité supérieure n'est en attente du sémaphore.

Cette fonction ne peut pas être utilisée par une fonction d'interruption.

semId L'identificateur du sémaphore à acquérir.

Valeur de retour

AA_ENONE	Pas d'erreur, le sémaphore est acquis.
AA_EARG	L'identificateur n'est pas un identificateur valide.
AA_ENOTALLOWED	Opération non autorisée à une interruption
AA_EWOULDBLOCK	Non acquis et timeout 0
AA_EFLUSH	Non acquis, libéré par *aaSemFlush()*.
AA_ETIMEOUT	Non acquis, timeout échu.

aaSemTryTake

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

Description TOC §↑

Tente d'obtenir le sémaphore avec un timeout de 0.

C'est une macro équivalente à : *aaSemTake (semId, 0)*.

Cette macro ne peut pas être utilisée par une fonction d'interruption.

Valeur de retour

Les mêmes que *aaSemTake()*.

aaSemGive

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

Description TOC §↑

Permet d'incrémenter le compteur d'un sémaphore, et éventuellement de l'affecter à une tâche en attente s'il est positif.

Cette fonction peu être appelée à partir d'une fonction d'interruption.

Valeur de retour

AA_ENONE	Pas d'erreur, le sémaphore est libéré.
AA_EARG	L'identificateur n'est pas un identificateur valide.

aaSemFlush

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

Description TOC §↑

Libère de façon atomique toutes les tâches en attente. Toutes les tâches sont libérées avant que l'une d'elle puisse éventuellement être activée.

L'état du sémaphore est inchangé.

Cette opération est utile pour réaliser une sorte de broadcast pour synchroniser des tâches.

Cette fonction peu être appelée à partir d'une fonction d'interruption.

Valeur de retour

AA_ENONE	Pas d'erreur, les tâches en attente sont libérées.

AA_EARG	L’identificateur n’est pas un identificateur valide.

aaSemReset

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

Description TOC §↑

Permet d’initialiser le compteur d’un sémaphore. Cela n’est autorisé que s‘‘il n’y a pas de tâche en attente d’acquisition du sémaphore.

Cette fonction peu être appelée à partir d’une fonction d’interruption.

Valeur de retour

AA_ENONE	Pas d’erreur, le sémaphore est initialisé.
AA_EARG	L’identificateur n’est pas un identificateur valide.
AA_ESTATE	Non effectué, il y a des tâches en attente.

4.5 Signaux inter tâches

aaSignalWait

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

Description TOC §↑

Permet à une tâche d'attendre qu'un ou plusieurs de ses signaux soient positionnés. Si les signaux sont déjà présents lors de l'appel, la tâche retourne immédiatement. Sinon elle est désactivée jusqu'à ce que les signaux soient positionnés ou que le timeout arrive à son terme.

sigsIn	Un masque de bits qui indique ceux qu'il faut attendre.
pSigsOut	Un pointeur sur une variable qui contient au retour de la fonction un masque de bits qui correspond aux signaux de sigsIn qui ont provoqué le retour. Ce masque de bits peut être différent de sigsIn si le mode AA_SIGNAL_OR a été utilisé. *pSigsOut* peut être NULL si le masque de signaux en sortie n'est pas utile.
mode	Le traitement des signaux à utiliser pour provoquer le retour de la fonction.
timeout	Le timeout en ticks système.

Les deux modes disponibles sont :

AA_SIGNAL_AND	La tâche attend jusqu'à ce que tous les signaux demandés soient positionnés.
AA_SIGNAL_OR	La tâche attend jusqu'à ce qu'au moins un des signaux demandés soit positionnés.

Cette fonction est interdite aux fonctions d'interruptions.

Valeur de retour

AA_ENONE	Les signaux attendus sont positionnés
AA_ENOTALLOWED	Non autorisé à partir d'une fonction d'interruption.
AA_EWOULDBLOCK	Les signaux attendus ne sont pas présents, et le timeout vaut 0.
AA_ETIMEOUT	Les signaux n'ont pas été positionnés avant l'échéance du timeout.

aaSignalSend

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

Description TOC §↑§↑

Positionne les signaux *sigs* dans la tâche spécifiée.

Si la tâche spécifiée est en attente de ces signaux, elle est immédiatement activée.

Cette fonction peut être utilisée par une fonction d'interruption.

taskId	L'identificateur de la tâche à signaler
sigs	Le masque des signaux à positionner.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	L'identificateur n'est pas un identificateur valide.

aaSignalPulse

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

Description TOC §↑

Positionne les signaux sigs et les RAZ immédiatement. Autrement dit cette fonction est équivalente à *aaSignalSend()*, mais les signaux sigs ne sont pas mémorisés.

Si la tâche est en attente et que la combinaison des signaux déjà présents et de sigs correspond à ce qu'attend la tâche celle ci est réveillée, et les signaux attendus sont mis à 0.

Dans tous les cas les signaux sigs sont mis à 0.

Cette fonction peut être utilisée par une fonction d'interruption.

taskId	L'identificateur de la tâche à signaler
sigs	Le masque des signaux à positionner.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	L'identificateur n'est pas un identificateur valide.

aaSignalClear

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

Description

TOC §↑

Permet de mettre à 0 les signaux *sigs* de la tâche *taskId*.

On peut employer AA_SGNAL_ALL comme valeur de *sigs* pour mette à 0 tous les signaux.

Valeur de retour

Aucune.

4.6 Allocation de mémoire dynamique

aaMalloc

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

Description TOC §↑

Permet d'allouer un bloc de mémoire dynamique. L'algorithme d'allocation dépend de la configuration du noyau.

size La taille du bloc en octets

Interdit pour une fonction d'interruption

Valeur de retour

Un pointeur sur le bloc alloué en cas de succès, la valeur NULL en cas d'échec.

aaCalloc

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

Description TOC §↑

Permet d'allouer un bloc de mémoire dynamique pour un tableau de *nmemb* éléments, chacun de taille *size*. La taille du bloc alloué est *nmemb*size* octets.

L'algorithme d'allocation dépend de la configuration du noyau.

nmemb Le nombre d'éléments à allouer

size La taille d'un élément en octets

Interdit pour une fonction d'interruption

Valeur de retour

Un pointeur sur le bloc alloué en cas de succès, la valeur NULL en cas d'échec.

aaRealloc

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

Description TOC §↑

Permet de changer la taille d'un bloc de mémoire dynamique précédemment alloué.

Interdit pour une fonction d'interruption

Valeur de retour

Un pointeur sur le nouveau bloc alloué en cas de succès, la valeur NULL en cas d'échec.

aaFree

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

Description TOC §↑

Permet de libérer un bloc de mémoire dynamique précédemment alloué.

Interdit pour une fonction d'interruption

Valeur de retour

Aucune

aaTryFree

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

Description TOC §↑

Tente de libérer un bloc de mémoire dynamique précédemment alloué. Si cela n'est pas possible la tâche appelante n'est pas bloquée.

La libération est impossible si le mutex de protection de l'allocation de mémoire dynamique est déjà acquis par une autre tâche.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_ENOTALLOWED	Opération non autorisée, à partir d'une interruption par exemple.
AA_EWOULDBLOCK	Libération impossible.

aaMemPoolCheck

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

Description

TOC §↑

Permet de vérifier l'intégrité des liens entre les blocs de mémoire dynamique. Cela peut permettre de détecter des écritures au-delà de la taille des blocs.

Utilise l'algorithme de vérification associé à l'allocation de mémoire dynamique, par exemple *tlsfCheck()*.

Interdit pour une fonction d'interruption

Valeur de retour

Celles de l'algorithme de vérification associé à l'allocation de mémoire dynamique.

4.7 Partition mémoire TLSF

tlsfInit

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

Description TOC §↑

Permet d'initialiser une partition mémoire gérée par l'algorithme TLSF

pMem	L'adresse de la partition
size	La taille de la partition en octets

Valeur de retour

En cas de succès : Le handle de la partition à utiliser avec les autres fonctions de gestion de partition TLSF.

En cas d'échec : NULL

tlsfMalloc

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

Description TOC §↑

Permet d'allouer un bloc mémoire.

hTlsf	le handle de la partition
size	La taille du bloc à allouer en octets

Valeur de retour

En cas de succès : l'adresse du bloc alloué.

En cas d'échec : NULL

tlsfCalloc

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

Description TOC §↑

Permet d'allouer un bloc mémoire de taille : `nmemb*size`.

hTlsf	le handle de la partition
nmemb	Le nombre d'éléments
size	La taille d'un élément

Valeur de retour

En cas de succès : l'adresse du bloc alloué.

En cas d'échec : NULL

tlsfFree

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

Description TOC §↑

Permet de restituer un bloc mémoire à la partition. Ce bloc doit avoir été alloué avec *tlsfMalloc()*.

hTlsf	le handle de la partition
ptr	L'adresse du bloc à libérer

Valeur de retour

Aucune

tlsfRealloc

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

Description TOC §↑

Permet de changer la taille d'un bloc alloué par *tlsfMalloc()*. Ou *tlsfRealloc()*.

hTlsf	le handle de la partition
ptr	L'adresse du bloc

size La nouvelle taille en octets

Valeur de retour

En cas de succès : l'adresse du bloc alloué.

En cas d'échec : NULL

tlsfCheck

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

Description TOC §↑

Permet de faire une vérification de l'intégrité de la partition mémoire.

hTlsf le handle de la partition

bVerbose Si ce paramètre vaut 0 seul la valeur de retour permet d'apprécier le résultat du test. Si le paramètre vaut 1 alors des informations supplémentaires sont envoyées avec *aaPrintf()*.

Valeur de retour

En cas de succès AA_ENONE.

En cas d'échec AA_FAIL

4.8 Partition mémoire par bloc

aaInitMallocBloc

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

Description TOC §↑

Permet d'initialiser une partition d'allocation par bloc.

pBloc	L'adresse de la partition
size	La taille de la partition en octets
pId	Un pointeur sur une variable qui contient le handle de la partition au retour de la fonction

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Argument invalide, la partition n'est pas créée.

aaMallocBloc

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

Description TOC §↑

Permet d'allouer un bloc dans la partition

blockId	Le handle de la partition
size	La taille du bloc à allouer en octets
ppBloc	Un pointeur sur une variable qui contient l'adresse du bloc alloué au retour de la fonction

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Argument invalide, le bloc n'est pas alloué.

aaMallocBlocFreeSize

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

Description

TOC §↑

Permet de connaître l'espace libre dans la partition

blockId	Le handle de la partition
pSize	Un pointeur sur une variable qui contient le nombre d'octets libres dans la partition au retour de la fonction

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Argument invalide la taille n'est pas renseignée.

4.9 Log et console

aaLogMes

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

Description TOC §↑

Permet d'envoyer une demande de trace à la tâche *tLogM*.

La trace n'est pas effectuée immédiatement mais placée dans une queue de messages et traité ultérieurement par la tâche *tLogM*, la fonction peut donc être appelée par une fonction d'interruption.

Les arguments ne doivent pas être des pointeurs sur des entités volatiles (allouées sur la pile de la tâche appelante par exemple).

.

Valeur de retour

Aucune

aaLogMesSetPutChar

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

Description TOC §↑

Permet de spécifier à la tâche *tLogM* la fonction à utiliser pour transmettre chaque caractère des messages de trace. Cela permet de choisir d'envoyer les messages vers la console, un UART ou SWO par exemple.

Valeur de retour

Aucune

aaPrintf

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

Description TOC §↑

Cette fonction a la même syntaxe que la fonction *printf()* du standard 'C'. Elle permet un formatage simplifié des messages afin d'occuper peu de place et d'être rapide.

Les formats supportés sont :

c	Un caractère
d i	Nombre décimal signé
u	Nombre décimal non signé
x X	Nombre hexadécimal
o	Nombre octal
b	Nombre binaire
f	nombre flottant (avec restrictions)
s	Chaîne de caractères
p	Pointer vers void (sortie similaire à %08X)
f	Float ou double.

Le format g n'est pas supporté.

Des champs optionnels peuvent apparaître entre % et le caractère de format, dans l'ordre suivant :

« - »	Cadrage à gauche.
« 0 »	Ajout de 0 devant pour compéter la longueur si le champ « width » est spécifié.
« width »	Nombre minimum de caractères à générer, peut être remplacé par *.
« l »	Spécification de taille, acceptée mais ignorée.

Le format %f a été introduit pour ne pas avoir à utiliser la librairie standard qui occupe beaucoup d'espace en FLASH et en RAM. Cependant cet affichage doit être considéré comme indicatif et ne supporte pas toutes les fonctionnalités de la librairie standard (pas ne NaN par exemple, ou précision inférieure).

Le format %f est validé et configure dans aacfg.h avec les constantes AA_WITH_FLOAT_PRINT, AA_FLOAT_T et AA_FLOAT_SEP.

Par exemple avec le format %7.2f la valeur -3.128 est affichée par ' -3.13'

Valeur de retour

Le nombre de caractères émis.

aaPrintfEx

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

Description

TOC §↑

Permet comme *aaPrintf()* de formater un message, mais en spécifiant la fonction à employer pour émettre les caractères.

Valeur de retour

Comme *aaPrintf()*.

aaSnPrintf

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

Description

TOC §↑

Permet comme *aaPrintf()* de formater un message, mais en le copiant dans le buffer pBuffer. C'est un équivalent à la fonction snprintf() de C99, mais sans allocation dynamique de mémoire, et un usage plus modéré de la pile.

pBuffer	La chaîne qui reçoit les caractères
size	La taille maximale à utiliser dans *pBuffer*, y compris le caractère nul final
fmt	Le format
…	Les arguments utilisés par le format

La chaîne contenue dans pBuffer est toujours terminée par '\0', même s'il y a eu débordement.

Valeur de retour

Le nombre de caractères qui sont générés, sans compter le '\0' final, en supposant que size est suffisant. Si ce nombre est supérieur ou égal à size, il y a eu troncation.

aaGets

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

Description

TOC §↑

Lit au plus size - 1 caractères depuis la console et les place dans le tampon pointé par pBuffer. La lecture s'arrête après un retour-chariot, qui n'est pas placé dans le tampon. Un caractère nul est placé à la fin de la ligne.

Certains caractères spéciaux et séquences ANSI sont gérées : backspaces, flèches, sup, home, end.

pBuffer	La chaine qui reçoit les caractères
size	La taille maximale à utiliser dans pBuffer, y compris le nul final

Valeur de retour

Le nombre de caractères effectivement renvoyés, non compté le nul final.

aaSetStdOut

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

Description

TOC §↑

Permet de spécifier la fonction à utiliser par *aaPrintf()* et *aaPutChar()* pour émettre un caractère.

En général la fonction *fnPutc()* permet d'émettre un caractère vers la console.

Valeur de retour

Aucune

aaSetStdIn

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

Description TOC §↑

Permet de spécifier la fonction à utiliser par *aaGetChar()* pour acquérir un caractère.

En général la fonction *fnGetc()* permet d'acquérir un caractère provenant de la console.

Valeur de retour

Aucune

aaPutChar

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

Description TOC §↑

Macro qui permet d'envoyer un caractère avec la fonction configurée avec *aaSetStdOut()*. L'utilisation de cette macro permet d'écrire des applications indépendantes du périphérique utilisé.

Valeur de retour

Aucune

aaGetChar

```
char      aaGetChar           (void)
```

Description TOC §↑

Macro qui permet d'acquérir un caractère avec la fonction configurée avec *aaSetStdIn()*. L'utilisation de cette macro permet d'écrire des applications indépendantes du périphérique utilisé.

Valeur de retour

Aucune

4.10 Timers logiciels

aaTimerCreate

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

Description TOC §↑

Permet de créer un timer et d'obtenir son identificateur.

pTimerId Un pointeur sur la variable qui contiendra l'identificateur du timer créé.

Valeur de retour

AA_ENONE	Pas d'erreur, le timer est créé.
AA_EARG	*pTimerId* n'est pas valide.
AA_EDEPLETED	Il n'y a plus de descripteur de timer disponible.

aaTimerDelete

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

Description TOC §↑

Désactive le timer, et le place dans la liste des timers libres. Il ne peut plus être utilisé.

Valeur de retour

AA_ENONE	Pas d'erreur, le timer est détruit.
AA_EARG	*timerId* n'est pas un indicateur valide.
AA_ESTATE	Ce timer est déjà détruit.

aaTimerIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de timer fourni est valide :

- Il correspond à un timer
- Le timer existe (a été créé et n'a pas été détruit).

`timerId` L'identificateur de timer à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaTimerSet

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

Description TOC §↑

Configure le timer.

timerId	L'identificateur du timer à utiliser.
callback	Pointeur sur la fonction qui sera appelée à l'échéance du timer, ce pointeur ne doit pas être NULL. Le prototype le la fonction est : *typedef uint32_t (* aaTimerCallback) (uintptr_t arg) ;*
arg	L'argument transmis à la callback.
timeout	Le délai du timer en tick système. Ce délai doit être entre 1 et AA_INFINITE-1.

Valeur de retour

AA_ENONE	Pas d'erreur, le timer est configuré.
AA_EARG	Un argument est invalide.
AA_ESTATE	Ce timer n'est pas utilisable (non créé, …).

aaTimerStart

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

Description TOC §↑

Permet démarrer le timer. Il faut que le timer ait été au préalable configuré avec *aaTimerSet()*.

Si le timer est déjà démarré, il est redémarré avec sa durée initiale.

Valeur de retour

AA_ENONE	Pas d'erreur, le timer est démarré.
AA_EARG	*timerId* est invalide.
AA_ESTATE	Ce timer n'est pas utilisable (non créé, …).

aaTimerStop

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

Description TOC §↑

Permet d'arrêter le timer avant l'échéance du timeout. Ce n'est pas une erreur d'arrêter un timer déjà arrêté.

Valeur de retour

AA_ENONE	Pas d'erreur, le timer est arrêté.
AA_EARG	*timerId* est invalide.
AA_ESTATE	Ce timer n'est pas utilisable (non créé, …).

4.11 Queues de messages

aaQueueCreate

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

Description TOC §↑

Permet de créer une queue de messages.

pQueueId Un pointeur sur la variable qui contiendra l'identificateur de la queue créé.

msgSize La taille maximale des messages en octets entre 1 et 65535.

msgCount Le nombre maximal de messages que la queue peut contenir, entre 1 et 65535.

pBuffer Si le tampon des messages est fourni par l'utilisateur, *pBuffer* est un pointeur sur un espace d'au moins *msgSize*msgCount* octets. Si le tampon doit être alloué par le noyau, alors pBuffer vaut NULL.

flags Une combinaison d' indicateurs :

- AA_QUEUE_PRIORITY Les tâches en attente sont traitées par priorité (exclusif de AA_QUEUE_FIFO).
- AA_QUEUE_FIFO Les tâches en attente sont traitées en FIFO (exclusif de AA_QUEUE_PRIORITY).
- AA_QUEUE_POINTER Les messages sont des pointeurs. Dans ce cas la valeur du paramètre msgSize est ignorée, la taille des messages est implicitement la taille d'un pointeur.

Pour que le noyau puisse allouer et libérer le tampon, l'allocation dynamique de mémoire doit être autorisée.

Valeur de retour

AA_ENONE Pas d'erreur, la queue est créé.

AA_EARG *pQueueId* n'est pas valide.

AA_EMEMORY Le tampon des messages n'a pas pu être alloué.

AA_EDEPLETED Il n'y a plus de descripteur de queue disponible.

aaQueueDelete

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

Description TOC §↑

Détruit la queue et la place dans la liste des queues libres.

Le tampon de messages est libéré s'il a été alloué par le noyau, sinon l'utilisateur doit s'en charger.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	pQueueId n'est pas valide.

aaQueueIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de queue fourni est valide :

- Il correspond à une queue
- Le timer existe (a été créée et n'a pas été détruite).

`timerId` L'identificateur de queue à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaQueueGive

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

Description TOC §↑

Permet d'ajouter un message à la queue.

pQueueId	L'identificateur de la queue.
pData	L'adresse du message à copier dans la queue.
size	La taille du message. Si size vaut 0, alors la taille spécifiée lors de la création de la queue est utilisée.
timeout	Le délai d'attente si la queue est pleine. Si le timeout vaut 0, la fonction retourne immédiatement.

La valeur du paramètre *pData* de *aaQueueGive()* dépend de l'utilisation de AA_QUEUE_POINTER à la création du pool.

Sans AA_QUEUE_POINTER	pData **est un pointeur** sur l'information à mettre dans la queue,
Avec AA_QUEUE_POINTER	pData **est** l'information à mettre dans la queue (par conséquent ce n'est pas un pointeur sur un pointeur).

Si le paramètre timeout est à 0, cela revient à utiliser une fonction qui pourrait s'appeler « *aaQueueTryGive()* ». Si le message n'a pas pu être placé dan la queue, la fonction retourne immédiatement la valeur AA_EWOULDBLOCK.

Si cette fonction est appelée par une interruption, le timeout est ignoré et considéré à 0.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Un argument n'est pas valide.
AA_EWOULDBLOCK	La queue est pleine et le timeout est 0, ou l'appelant est une interruption.
AA_ETIMEOUT	La queue est pleine et le timeout est arrivé à échéance.

aaQueueTake

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

Description TOC §↑

Permet d'extraire un message de la queue.

pQueueId L'identificateur de la queue.

pData L'adresse où copier le message extrait de la queue.

size La taille du message à copier. Si size vaut 0, alors la taille spécifiée lors de la création de la queue est utilisée. Dans le ca d'utilisation de AA_QUEUE_POINTER size est ignoré.

timeout Le délai d'attente si la queue est vide. Si le timeout vaut 0, la fonction retourne immédiatement.

Si le paramètre timeout est à 0, cela revient à utiliser une fonction qui pourrait s'appeler « *aaQueueTryTake()* ».

Si cette fonction est appelée par un driver d'interruption, le timeout est ignoré et considéré à 0.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Un argument n'est pas valide.
AA_EWOULDBLOCK	La queue est vide et le timeout est 0, ou l'appelant est une interruption.
AA_ETIMEOUT	La queue est vide et le timeout est arrivé à échéance.

aaQueuePeek

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

Description TOC §↑

Permet d'obtenir l'adresse du message dans la queue. Cela permet d'inspecter le message avant de le retirer de la queue.

Cette fonction doit être utilisée avec précaution s'il y a plusieurs lecteur de la queue : pendant qu'une tâche inspecte un message, une autre tâche peut le retire de la queue.

pQueueId L'identificateur de la queue.

pData	L'adresse du message à copier dans la queue.
timeout	Le délai d'attente si la queue est pleine. Si le *timeout* vaut 0, la fonction retourne immédiatement.

Si cette fonction est appelée par une interruption, le *timeout* est ignoré et considéré à 0.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Un argument n'est pas valide.
AA_EWOULDBLOCK	La queue est pleine et le *timeout* est 0, ou l'appelant est une interruption.
AA_ETIMEOUT	La queue est pleine et le *timeout* est arrivé à échéance.

aaQueuePurge

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

Description TOC §↑

Permet d'enlever le premier message de la queue sans le lire.

Cela peut être utilisé conjointement avec aaQueuePeek(), s'il n'est pas nécessaire de lire le message.

pQueueId L'identificateur de la queue.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Un argument n'est pas valide.

aaQueueGetCount

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

Description TOC §↑

Copie dans la variable pointée par *pCount* le nombre de messages présents dans la queue.

Valeur de retour

AA_ENONE	Pas d'erreur
AA_EARG	Un argument n'est pas valide.

4.12 Pool de tampons

aaBufferPoolCreate

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

Description TOC §↑

Permet de créée un descripteur pour un pool de tampons

pPoold	Un pointeur sur la variable qui contiendra l'identificateur du pool créé.
bufCount	Le nombre de tampons du pool.
bufSize	La taille d'un tampon en octets.
pBuffer	Si le pool des tampons est fourni par l'utilisateur, *pBuffer* est un pointeur sur un espace d'au moins *bufSize*bufCount* octets. Si le tampon doit être alloué par le noyau, alors pBuffer vaut NULL.

Pour que le noyau puisse allouer et libérer le pool, l'allocation dynamique de mémoire doit être autorisée.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Un argument n'est pas valide.
AA_EMEMORY	Le pool de tampons n'a pas pu être alloué.
AA_EDEPLETED	Il n'y a plus de descripteur de pool disponible.

aaBufferPoolDelete

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

Description TOC §↑

Permet de détruire un pool de tampons, et de placer son descripteur dans la liste des descripteurs libres.

bufPoolId	L'identificateur du pool
bForce	Si *bForce* est à 0, le pool n'est détruit que si tous les tampons du pool sont libres (rendus au pool). Si *bForce* est à 1, le tampon est détruit inconditionnellement.

Le pool de tampons est libéré s'il a été alloué par le noyau, sinon l'utilisateur doit s'en charger.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Un argument n'est pas valide.

aaBufferPoolIsId

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

Description TOC §↑

Cette fonction permet de vérifier que l'identificateur de pool de tampon fourni est valide :

- L'identificateur correspond à un pool de tampon
- Le pool de tampon existe (a été créé et n'a pas été détruit).

`bufPoolId`	L'identificateur de pool de tampon à vérifier.

Valeur de retour

AA_ENONE	L'identificateur est valide.
AA_EFAIL	L'identificateur n'est pas valide.

aaBufferPoolTake

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

Description TOC §↑

Permet d'obtenir un tampon du pool.

bufPoolId	L'identificateur du pool
ppBuffer	Un pointeur sur un pointeur qui contient l'adresse du tampon au retour de la fonction.

Valeur de retour

AA_ENONE	Pas d'erreur.
AA_EARG	Un argument n'est pas valide.
AA_EDEPLETED	Il n'y a pas de tampon disponible.

aaBufferPoolGive

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

Description TOC §↑

Permet de rendre au tampon au pool.

bufPoolId L'identificateur du pool

pBuffer L'adresse du tampon à rendre au pool.

Valeur de retour

AA_ENONE Pas d'erreur.

AA_EARG Un argument n'est pas valide.

aaBufferPoolGetCount

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

Description TOC §↑

Permet d'obtenir le nombre de tampons disponibles du pool.

bufPoolId L'identificateur du pool.

pCount L'adresse d'une variable qui contiendra le compte de tampons disponibles.

Valeur de retour

AA_ENONE Pas d'erreur.

AA_EARG Un argument n'est pas valide.

aaBufferPoolReset

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

Description TOC §↑

Permet de réinitialiser le pool dans son état lors de sa création.

Valeur de retour

AA_ENONE Pas d'erreur.

AA_EARG Un argument n'est pas valide.

4.13 Fonctions utilisateur

Ce chapitre liste les fonctions de l'application connues du noyau.

userInitTask

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

Description TOC §↑

Cette fonction est appelée par l'initialisation du noyau et c'est la première tâche de l'application.

Lors de l'appel de cette fonction le noyau est complétement démarré, toutes les ressources sont disponibles.

L'application doit obligatoirement définir cette fonction.

Valeur de retour

Aucune

aaUserReleaseStack

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

Description TOC §↑

Si une tâche utilise une pile statique allouée par l'utilisateur, alors lors de la destruction de cette tâche la callback *aaUserReleaseStack()* est appelée. Cela donne l'opportunité à l'utilisateur de savoir qu'un bloc mémoire se libère et de le gérer en conséquence.

pStack Un pointeur sur le bloc mémoire à libérer.

size La taille du bloc mémoire.

Si le bloc mémoire est libéré, la fonction doit retourner AA_ENONE.

Si le bloc mémoire n'est pas libéré, la fonction doit retourner AA_EFAIL. La callback sera rappelée plus tard pour une nouvelle tentative de libération.

Une version « weak » de cette fonction est définie par le noyau, elle retourne toujours AA_ENONE. Si l'application n'a pas l'usage de cette fonction elle n'a pas besoin de la définir.

Valeur de retour

AA_ENONE Le bloc mémoire est libéré.

AA_EFAIL Le bloc mémoire n'est pas libéré.

aaUserNotify

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

Description TOC §↑

Cette fonction permet au noyau de prévenir l'utilisateur de certains événements.

Ces événements sont identifiés par la valeur du paramètre *event* :

AA_NOTIFY_STACKOVFL La pile de la tâche à débordé. *arg* est l'identificateur de tâche.

AA_NOTIFY_STACKTHR Le seuil de surveillance de la pile a été atteint. *arg* est l'identificateur de tâche.

Pour que ces événements soient détectés et transmis à l'utilisateur la tâche doit être créée avec le flag *AA_FLAG_STACKCHECK*.

Ces événements ne sont notifiés qu'une seule fois.

Une version « weak » de cette fonction est définie par le noyau. Si l'application n'a pas l'usage de cette fonction elle n'a pas besoin de la définir.

Valeur de retour

Aucune

4.14 Board Support Package

bspGetTickRate

```
uint32_t  bspGetTickRate     (void)
```

Description TOC §↑

Permet de connaître la fréquence de l'horloge système (tick).

Valeur de retour

La fréquence de l'horloge système en hertz.

bspSetTickRate

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

Description TOC §↑

Permet de spécifier la fréquence de l'horloge système (tick). N'est pas utilisable si le mode tick étiré est choisi.

tickHz La fréquence de l'horloge système en hertz.

Valeur de retour

AA_ENONE	L'horloge système est configurée
AA_FAIL	L'horloge système n'est pas configurée

bspGetSysClock

```
uint32_t  bspGetSysClock     (void)
```

Description TOC §↑

Permet de connaître la fréquence d'horloge du processeur.

Valeur de retour

La fréquence d'horloge du processeur

bspResetHardware

```
void      bspResetHardware   (void)
```

Description

TOC §↑

Permet de réaliser un reset du processeur par logiciel.

Valeur de retour

Aucune.

bspOutput

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

Description

TOC §↑

Permet de positionner l'état d'une sortie GPIO configurée par le BSP.

En général les LEDs d'une carte d'évaluation sont définies par le BSP, et des constantes sont utilisées pour accéder à ces sorties.

num Le numéro de GPIO qui peut être défini par le BSP, par exemple BSP_LED0.

state L'état à affecter à la sortie GPIO : 0 ou 1.

Cette fonction est déclarée « inline » et si les paramètres sont des constantes le code généré se réduit à une seule instruction assembleur, ce qui rend cette fonction très peu intrusive.

Valeur de retour

Aucune

bspInput

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

Description

TOC §↑

Permet de lire l'état d'une entrée GPIO configurée par le BSP.

En général les boutons d'une carte d'évaluation sont définis par le BSP, et des constantes sont utilisées pour accéder à ces entrées.

num — Le numéro de GPIO qui peut être défini par le BSP, par exemple BSP_BUTTON0.

Valeur de retour

La valeur de l'entrée GPIO : 0 ou 1.

bspMainStackCheck

```
uint32_t  bspMainStackCheck  (void)
```

Description

TOC §↑

Permet de connaître l'utilisation de la pile système.

Valeur de retour

Le nombre de mots inutilisés dans la pile système.

Les fonctions de « time stamp » permettent de mesurer des périodes de temps en utilisant le compteur de cycle du processeur. Le compteur est sur 32 bits, et la résolution correspond à la fréquence de l'horloge système.

Par exemple si l'horloge système est à 168 MHz, la résolution du compteur est de 5.95 ns. Dans ces conditions le compteur de 32 bits reboucle au bout d'environ 25.5 secondes, ce qui correspond à la période de temps maximale mesurable, suivant la formule : Tmax = 2^{32} / *bspGetSysClock()*.

Fréquence CPU	Délai maximal mesurable
64 MHz	67.1 s
168 MHz	25.5 s
400 MHz	10.7 s

Des mesures en haute résolution peuvent être faites en utilisant les fonctions *bspTsGet()* et *bspRawDelta()*.

Il est parfois nécessaire d'accumuler des durées pendant un temps supérieur à celui permis par les 32 bits et la résolution du compteur de cycles. Pour cela on peut utiliser *bspTsDelta()* qui fournit une durée en µs. Il est ainsi possible d'accumuler des durées pendant environ 1h:11mn. Faites attention aux pertes dues à l'accumulation des arrondis

Attention : chaque delta reste soumis à la contrainte de rebouclage du compteur de cycle, et doit donc être inférieur à cette durée de rebouclage.

bspTsGet

```
uint32_t  bspTsGet          (void)
```

Description

TOC §↑

Donne la valeur courante du compteur de cycle du processeur. La résolution est celle de la fréquence de l'horloge système, qui peut être obtenue avec *bspGetSysClock()*.

Cette fonction est déclarée « inline ».

Valeur de retour

La valeur courante du compteur de cycle.

bspRawTsDelta

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

Description

TOC §↑

Acquiert la valeur courante du compteur de cycle et en soustrait la valeur pointée par *pTs*, le résultat de la soustraction est la valeur retournée. Elle correspond au nombre de cycles écoulés entre l'appel précédent à *bspTsGet()* ou *bspTsDelta()* qui a fourni la valeur pointée par *pTs*, et l'instant de l'appel de cette fonction.

Après le calcul la valeur courante du compteur est placée dans la variable pointée par *pTs*.

Cette fonction est déclarée « inline ».

Valeur de retour

La différence entre la valeur courante du compteur et la valeur pointée par *pTs*.

bspTsDelta

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

Description

TOC §↑

Effectue les mêmes opérations que *bspRawTsGet()*, mais la valeur retournée est convertie en µs.

Cela permet par exemple d'accumuler des délais jusqu'à plus d'une heure dans une variable 32 bits non signée.

Cette fonction est déclarée « inline ».

Exemple pour vérifier la durée d'une seconde du noyau :

```
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) ;
```

Valeur de retour

La différence entre la valeur courante du compteur et la valeur pointée par *pTs* convertie en µs.

bspDelayUs

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

Description TOC §↑

Exécute une attente active de la durée en micro seconde passée en paramètre. La tâche appelante n'est pas suspendue pour la durée du délai et donc consomme du temps CPU.

Par contre la tâche peut être préemptée par une tâche de priorité supérieure qui devient prête. Le temps d'attente en paramètre est donc un temps minimum.

La précision du délai est de l'ordre de 1% au delà de 10µs.

swoInit

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

Description TOC §↑

Initialise le mécanisme ITM qui permet d'émettre des traces par la broche SWO en mode DEBUG.

portBits	Un masque qui permet de spécifier les stimulus à configurer.
cpuCoreFreqHz	La fréquence d'horloge du processeur en Hz.
baudrate	La fréquence bit de la sortie SWO

Les valeurs de *cpuCoreFreqHz* et *baudrate* sont utilisées pour calculer le diviseur nécessaire à la génération de la fréquence *baudrate*.

Pour savoir si le mécanisme SWO est opérationnel, utiliser *swoIsEnabled()*.

Valeur de retour

Aucune

swoIsEnabled

```
void      swoIsEnabled       (void)
```

Description TOC §↑

Permet de savoir si mécanisme SWO est initialisé et opérationnel.

Si cette fonction est appelée alors que l'application est gérée par un débugger, alors le mécaniste SWO est opérationnel, et la fonction *swoIsEnabled()* retournera une valeur non nulle.

Si cette fonction est appelée alors que l'application n'est pas gérée par du débugger (application lancée par un reset physique par exemple), alors le mécaniste SWO n'est pas opérationnel, et la fonction *swoIsEnabled()* retournera 0. Dans ce cas les fonctions de gestion SWO peuvent être appelées, mais ne feront rien.

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 §↑

Ces fonctions permettent démettre des mots de différentes longueurs sur le stimulus *portNo*, de 0 à 31, de l'ITM.

Valeur de retour

Aucune

swoPutChar, swoPutStr

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

Description TOC §↑

swoPutChar() permet d'émettre un caractère sur le stimulus 0 de l'ITM. Cette fonction est utilisée par le BSP pour spécifier la redirection des sorties de *logMes()* vers la sortie SWO.

La fonction *swoPutStr()* permet d'émettre une chaîne de caractère terminée par 0 sur le stimulus 0 de l'ITM.

Valeur de retour

Aucune

4.15 Fonctions intrinsèques

Le noyau utilise des fonctions intrinsèques du compilateur si elles existent, sinon le BSP doit les fournir. Les fonctions utilisées sont donc disponibles pour les applications. On se reportera avantageusement à la documentation du compilateur pour la signification et l'utilisation de ces fonctions

Correspondances pour 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
```

L'utilisation des fonctions intrinsèques permet dans ne nombreuses utilisations de se passer de bibliothèque « C », du type newlib ou newlib-nano.

5 Licence

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 non commercially. 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.