Chapitre 8
Noyau Et Services

8.1 Introduction

Ce chapitre constitue le cœur de l'utilisation avancée d'un système d'exploitation temps réel (RTOS). Après avoir introduit dans le chapitre précédent les concepts fondamentaux de tâche et d'ordonnancement, il s'agit désormais de maîtriser l'ensemble des services offerts par le noyau pour construire des applications multitâches robustes, déterministes et efficaces. Réparti sur trois semaines, ce module aborde successivement les mécanismes de synchronisation et de communication inter-tâches, les stratégies de gestion mémoire adaptées aux contraintes temps réel, et enfin les services temporels ainsi que les techniques de supervision et de débogage avancées. Chaque notion sera illustrée par des exemples pratiques implémentés sur le microcontrôleur cible, permettant de passer progressivement d'une utilisation basique du RTOS à une maîtrise approfondie de ses capacités.

8.2 Synchronisation Et Communication Inter-Tâches

La première semaine est consacrée aux mécanismes fondamentaux qui permettent aux différentes tâches d'un système temps réel de coopérer de manière ordonnée et sécurisée. Dans un environnement multitâque préemptif, l'absence de tels mécanismes conduirait inévitablement à des incohérences de données, des interblocages ou des comportements non déterministes. Trois familles d'outils sont étudiées : les sémaphores pour la signalisation, les mutex pour l'exclusion mutuelle avec gestion des priorités, et les files de messages pour les échanges de données structurées.

8.2.1 Sémaphores : Signalisation Et Comptage

Les sémaphores constituent le mécanisme de signalisation le plus élémentaire et le plus polyvalent offert par la plupart des RTOS. Nous distinguerons d'abord les sémaphores binaires, qui ne prennent que les valeurs 0 ou 1 et sont particulièrement adaptés à la synchronisation entre une routine d'interruption (ISR) et une tâche : lorsque l'interruption survient, elle libère le sémaphore, réveillant ainsi la tâche en attente qui peut alors traiter l'événement. Les sémaphores compteurs, quant à eux, peuvent prendre une valeur entière supérieure à 1 et sont utilisés pour gérer l'accès à un nombre limité de ressources identiques, comme un pool de plusieurs tampons mémoire. Nous analyserons en détail les opérations fondamentales que sont la prise (pendant ou take) et la libération (poster ou give), en insistant sur leur caractère atomique et sur les comportements d'attente avec ou sans temporisation.

8.2.2 Mutex Et Protection Des Ressources Partagées

Les mutex (pour mutual exclusion) sont des sémaphores spécialisés conçus spécifiquement pour protéger l'accès à une ressource partagée, qu'il s'agisse d'une variable globale, d'un

périphérique matériel ou d'une zone mémoire. Contrairement aux sémaphores binaires classiques, les mutex intègrent un mécanisme essentiel dans les systèmes temps réel préemptifs : l'héritage de priorité. Nous étudierons en profondeur le problème de l'inversion de priorité, phénomène redoutable où une tâche de haute priorité se trouve bloquée indirectement par une tâche de basse priorité qui détient un mutex. L'héritage de priorité permet de résoudre ce problème en élevant temporairement la priorité de la tâche détentrice du mutex au niveau de la tâche en attente la plus prioritaire. Nous aborderons également les bonnes pratiques d'utilisation des mutex, notamment l'importance de respecter la discipline de prise et de libération au sein d'une même tâche, ainsi que les risques d'interblocage (deadlock) lorsque plusieurs mutex sont utilisés de manière imbriquée.

8.2.3 Files De Messages : Communication Structurée

Au-delà de la simple signalisation, les tâches ont souvent besoin d'échanger des données complexes. Les files de messages (queues) offrent un mécanisme de communication inter-tâches à la fois fiable et flexible. Nous verrons comment une file de messages permet d'envoyer des données d'une tâche à une autre, ou d'une interruption vers une tâche, en garantissant la copie sécurisée des données dans un tampon géré par le noyau. Les concepts fondamentaux seront détaillés : création d'une file avec une taille fixe d'éléments, envoi (send) avec comportement bloquant ou non-bloquant selon que la file est pleine ou non, réception (receive) avec attente optionnelle, et capacité à surveiller simultanément plusieurs files via des mécanismes de multiplexage. Des cas d'usage concrets illustreront comment combiner files de messages et sémaphores pour construire des architectures logicielles modulaires, comme la mise en œuvre d'un pipeline de traitement de données ou d'un serveur de requêtes asynchrones.

8.3 Gestion Mémoire En Environnement Temps Réel

La deuxième semaine aborde un aspect critique de la conception des systèmes embarqués temps réel : la gestion de la mémoire. Contrairement aux systèmes d'exploitation généralistes qui privilégient la flexibilité, un RTOS doit offrir des mécanismes de gestion mémoire déterministes, c'est-à-dire dont le temps d'exécution est prévisible et borné. Nous comparerons les deux grandes familles d'approches : l'allocation dynamique classique (heap) et les méthodes à base de pools de mémoire fixes, en analysant leurs compromis respectifs en termes de flexibilité, de déterminisme et de risques de fragmentation.

8.3.1 Allocation Dynamique Et Problématique De La Fragmentation

Nous commencerons par examiner l'allocation dynamique de mémoire, telle qu'elle est pratiquée avec les fonctions malloc() et free() en langage C. Bien que flexible et familière pour les développeurs issus du monde des systèmes généralistes, cette approche présente des inconvénients majeurs en environnement temps réel. Le principal risque est la fragmentation de la mémoire : après une série d'allocations et de libérations de tailles variées, le tas (heap) peut se retrouver morcelé en multiples petits blocs libres non contigus, rendant impossible

l'allocation d'un bloc de grande taille alors même que la quantité totale de mémoire disponible serait suffisante. Plus problématique encore, le temps d'exécution des opérations d'allocation n'est pas constant et peut varier considérablement selon l'état du tas, ce qui introduit de la gigue (jitter) incompatible avec les contraintes temps réel dur. Nous analyserons également les risques de fuites mémoire et de corruption de tas, particulièrement insidieux dans les systèmes embarqués destinés à fonctionner sans interruption pendant de longues périodes.

8.3.2 Pools De Mémoire Fixes : Déterminisme Et Fiabilité

Face aux limitations de l'allocation dynamique traditionnelle, les RTOS proposent généralement des mécanismes de pools de mémoire fixes (memory pools ou fixed-size block allocators). Le principe est simple : au démarrage du système, une zone mémoire statique est découpée en un nombre fixe de blocs de taille identique. L'allocation et la libération d'un bloc s'effectuent alors en temps constant (O(1)), sans risque de fragmentation puisque tous les blocs sont interchangeables. Nous verrons comment créer et utiliser ces pools, en distinguant les pools statiques, configurés à la compilation, des pools dynamiques créés à l'exécution. Cette approche est particulièrement adaptée pour les structures de données récurrentes comme les tampons de communication, les descripteurs de tâches temporaires, ou les éléments de listes chaînées utilisées dans les protocoles réseau. Nous discuterons également des stratégies hybrides, combinant pools de différentes tailles pour répondre à des besoins variés tout en maintenant le déterminisme.

8.3.3. Stratégies De Gestion Mémoire Pour Les Systèmes Temps Réel

Au-delà des mécanismes offerts par le RTOS, nous aborderons les bonnes pratiques d'architecture mémoire pour les systèmes embarqués critiques. L'accent sera mis sur la distinction entre mémoire statique (allouée à la compilation), mémoire automatique (pile ou stack), et mémoire dynamique contrôlée. Nous étudierons la gestion des piles de tâches, paramètre crucial car chaque tâche dispose de sa propre pile dont la taille doit être dimensionnée avec soin pour éviter les débordements (stack overflow) tout en minimisant la consommation mémoire. Des techniques de diagnostic comme le remplissage des piles par un motif connu (canary) et la surveillance périodique de l'utilisation réelle seront présentées. Enfin, nous verrons comment certaines architectures critiques imposent l'interdiction pure et simple de toute allocation dynamique post-initialisation, conformément aux normes de sécurité comme MISRA-C ou DO-178C, et comment concevoir des systèmes répondant à ces exigences.

8.4 Gestion Du Temps, Supervision Et Débogage

La troisième semaine finalise l'étude des services noyau en abordant deux aspects complémentaires : d'une part la gestion du temps avec les timers logiciels, d'autre part les techniques avancées de supervision et de débogage qui sont essentielles pour valider le comportement d'un système temps réel. Cette semaine met l'accent sur la transition entre le

développement et la mise en production, en outillant le développeur pour diagnostiquer et résoudre les problèmes subtils propres aux environnements multitâches.

8.4.1 Timers Logiciels : Temporalité Flexible

Au-delà des timers matériels étudiés au chapitre 4, les RTOS proposent des timers logiciels qui permettent d'exécuter des fonctions de rappel (callback) après un délai spécifié ou de manière périodique. Nous verrons comment ces timers logiciels sont implémentés par le noyau, généralement en s'appuyant sur une tâche système dédiée (la timer service task) et une file d'événements temporels. La distinction sera faite entre les timers à déclenchement unique (one-shot) et les timers périodiques (auto-reload). Nous analyserons les compromis liés à l'exécution des fonctions de rappel : celles-ci s'exécutent dans le contexte de la tâche système et doivent donc être courtes pour ne pas affecter l'ordonnancement global. Des alternatives comme la création de tâches dédiées réveillées par des notifications temporelles seront également présentées pour les traitements plus longs. Enfin, nous aborderons la synchronisation temporelle précise avec les services de délai (delay) et d'obtention de l'heure système (tick count), en discutant des implications de la résolution temporelle définie par la fréquence du tick du RTOS.

8.4.2 Supervision : Watchdog Logiciel Et Surveillance Des Tâches

La robustesse d'un système temps réel repose non seulement sur sa correction fonctionnelle mais aussi sur sa capacité à détecter et à réagir face à des dysfonctionnements. Nous étudierons d'abord l'intégration du watchdog matériel (chien de garde) avec le RTOS, en présentant les stratégies de rafraîchissement (kicking) adaptées aux architectures multitâches, comme l'utilisation d'une tâche de supervision dédiée. Puis nous explorerons les mécanismes de watchdog logiciel plus sophistiqués, permettant de surveiller individuellement chaque tâche. Ces mécanismes reposent sur le principe du "cœur qui bat" (heartbeat) : chaque tâche superviseée doit signaler périodiquement son bon fonctionnement à une tâche monitrice. L'absence de signal dans un délai imparti déclenche une procédure de récupération, qui peut aller de la simple notification à la réinitialisation sélective de la tâche défaillante ou du système entier. Nous verrons également comment détecter les situations d'interblocage (deadlock) et d'inversion de priorité non résolue à l'aide de compteurs de temps d'attente et d'alertes configurées.

8.4.3 Débogage Avancé En Conditions Réelles

Le dernier volet de ce chapitre est consacré aux techniques de débogage spécifiques aux systèmes temps réel, où les méthodes traditionnelles comme le débogage pas à pas sont souvent inopérantes car elles perturbent le comportement temporel du système. Nous aborderons le débogage par traces, qui consiste à enregistrer séquentiellement des événements clés (changements de contexte, prises de sémaphores, envois de messages, entrées dans les ISR) dans un tampon circulaire en mémoire, pour une analyse post-mortem ou en temps réel via une liaison série. L'utilisation d'outils de visualisation comme les traceurs (Tracealyzer, Percepio) sera présentée, permettant de reconstituer graphiquement l'exécution des tâches et

d'identifier visuellement les problèmes d'ordonnancement, de latence excessive ou de non-respect des échéances. Nous discuterons également des techniques de débogage à distance (JTAG/SWD) avec points d'arrêt conditionnels et points de déclenchement (watchpoints), en insistant sur l'importance de ne pas altérer le comportement temps réel lors de ces opérations. Enfin, la mise en œuvre de tests de charge et de validation temporelle, consistant à instrumenter le code pour mesurer les latences maximales et minimales des différentes opérations critiques, sera présentée comme une étape indispensable avant la mise en production d'un système embarqué temps réel.