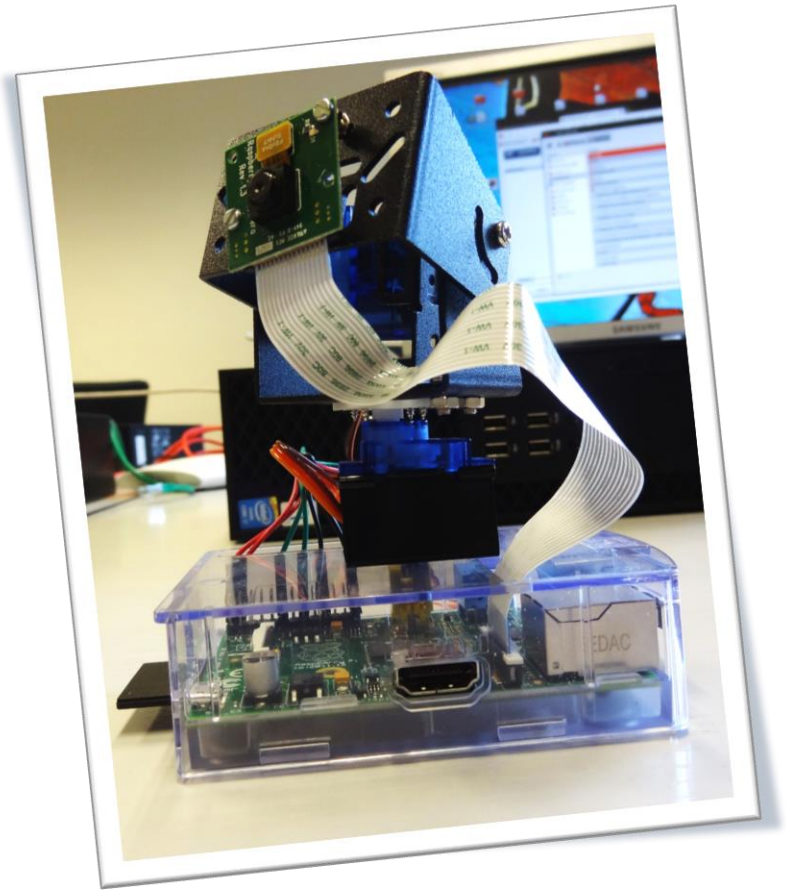
Développement avancé d'un système embarqué « satellite AGILE » sur carte Raspberry PI

Rapport de Projet Long

Lucie Beaussart
Thomas Bétous
Abdelkader Bouarfa
William Excoffon



INP TOULOUSE ENSEEIHT

LAAS-CNRS

Remerciements

Nous souhaitons remercier Jean-Charles Fabre pour son soutien et son implication dans ce projet que nous avons été ravis de mener sous sa supervision.

Nous remercions aussi Martin Cronel pour son travail, sa patience et le temps qu'il nous a accordé tout au long de ces quelques semaines.

Nos remerciements s'adressent également à Marie-Hélène Deredempt et Jean-Paul Blanquart (AIRBUS Defence & Space), dont les informations techniques nous ont été très précieuses lors de la confrontation de notre travail avec la réalité du monde industriel.

De plus, nous remercions chaleureusement Michaël Lauer pour la disponibilité et la gentillesse dont il a fait preuve durant notre séjour au LAAS.

Enfin, nous sommes reconnaissants envers les ateliers d'électronique et de mécanique du LAAS ainsi qu'envers Matthieu Roy dont les coups de pouce ont été précieux dans les moments critiques.

Table des matières

I. Introduction

Les systèmes spatiaux naissent de la mise en œuvre de moyens technologiques très coûteux. Avant même de parler du coût de la technologie envoyée dans l'espace pour réaliser les fonctions vitales d'un satellite, le prix du lancement lui-même est déjà faramineux : par exemple, pour envoyer une personne dans l'espace, on a besoin de trois fois son poids en or (au cours actuel). En moyenne le coût se situe autour de 15k€/kg ce qui amène le coût moyen du lancement d'un satellite d'observation de la Terre à environ 50 millions d'euros.

Un autre point très gênant pour les systèmes spatiaux est la difficulté voire l'absence de maintenance. Il faut donc assurer au maximum la survie d'un satellite pour rentabiliser au mieux sa production. Parmi les contraintes imposées, le système doit être le plus sûr de fonctionnement possible et être tolérant à plusieurs types de fautes. Récemment, les métiers de l'espace se sont intéressés aux standards de l'aéronautique qui offrent de belles performances de sûreté et de temps réel. Le standard ARINC653 définit un noyau avionique temps réel qui offre beaucoup de fonctionnalités pertinentes et qui justifient l'intérêt porté par les systèmes spatiaux.

Notre projet long s'inscrit en plein cœur des réalités et des contraintes industrielles d'aujourd'hui. La problématique des ressources partagées n'a toujours pas de solution connue à l'heure actuelle, dans un contexte technique où des données de taille importante (les images) sont amenées à être manipulées par plusieurs entités. En collaboration avec AIRBUS Defence & Space, notre projet s'appuie sur les travaux d'un Bureau d'Etudes Industriel concernant la prochaine génération de satellite d'observation de la Terre, « Satellite AGILE ». L'objectif principal du projet est de développer un simulateur avancé d'un satellite AGILE sur une carte Raspberry PI, l'objectif secondaire étant de faire progresser la réflexion sur l'utilisation des noyaux avioniques temps réel par les systèmes spatiaux.

II. Contexte et présentation du sujet

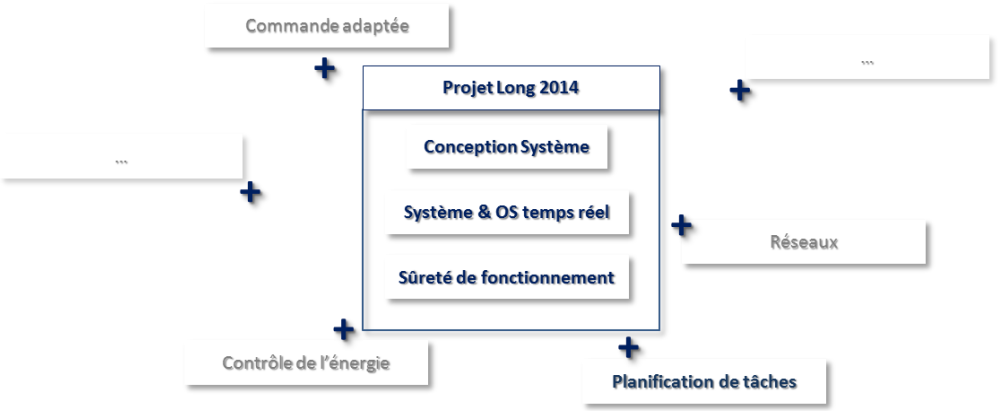
La Chaire d'Enseignement des Systèmes Embarqués Critiques (CESEC) est une chaire portée par trois écoles d'ingénieurs de Toulouse : ISAE, INSA et INPT-ENSEEIHT. Elle vise à renforcer les formations existantes dans le domaine des Systèmes Embarqués Critiques en accordant plus d'importance à la dimension « système ». Cette chaire permettra de développer une nouvelle pédagogie d'enseignement de l'ingénierie des systèmes embarqués critiques en proposant une plateforme de projets mutualisés conçue en collaboration avec l'industrie.

Le projet long intitulé *« Développement avancé d'un système embarqué « satellite AGILE » sur carte Raspberry PI »* se fonde sur les travaux effectués dans le cadre du Bureau d'Etudes Industriel « Satellite Agile » qui s'est déroulé de décembre 2013 à janvier 2014 à l'INPT-ENSEEIHT dans le cadre de la Chaire CESEC en collaboration avec Airbus Defence & Space.

Le projet « Satellite AGILE » complet est une synergie entre plusieurs thématiques apportant chacune sa contribution à la réalisation du projet final. A chaque étape, le projet devra conserver les apports

des thématiques déjà traitées et permettre l'arrivée de nouvelles « briques » pour construire finalement un véritable édifice de collaboration sur plusieurs années de travail.

Le récent BEI et le projet long qui fait l'objet de ce rapport se concentrent sur les aspects Conception Système, Système & OS Temps Réel et Sûreté de fonctionnement. Un autre groupe de projet long s'est concentré sur la thématique Planification de tâches en lien direct avec notre projet long.



Enfin, notre projet long s'est déroulé en grande partie à Toulouse au LAAS/CNRS, le Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique, dans l'équipe Tolérance aux Fautes et Sûreté de Fonctionnement Informatique (TSF), sous la tutelle de Jean-Charles FABRE, Professeur des Universités et Michaël LAUER, Maître de Conférences à l'Université Paul Sabatier.

III. Objectifs

Le projet long doit permettre à terme d'obtenir une simulation avancée du fonctionnement d'un satellite AGILE d'observation de la Terre avec tolérance aux fautes. Les objectifs se distinguent en cinq parties.

III. 1. Finalisation et portage du service de gestion d'images satellite sur une carte Raspberry PI

Il est d'abord nécessaire de compléter les fonctionnalités du simulateur du satellite pour satisfaire au mieux les conditions d'une mission spatiale d'observation de la Terre. Le code source doit être nettoyé et doit mieux respecter le Time & Space Partionning du standard ARINC653. Les fonctions concernant la mission (charge utile) et les fonctions concernant le contrôle du satellite (plateforme) doivent être séparées des partitions principales et doivent avoir un accès unique et réservé aux périphériques qui les concernent.

Le simulateur du satellite AGILE devra ensuite être hébergé sur un Raspberry PI tournant sous Linux. Il devient nécessaire d'assurer le fonctionnement du programme sur une architecture ARM, différente des processeurs Intel à architecture x86 des ordinateurs de l'INPT-ENSEEIHT sur lesquels a été développée la première version du simulateur.

III. 2. Utilisation d'une caméra pilotable pour une prise d'image effective

Le projet prend une dimension pratique considérable avec l'utilisation d'une caméra pilotable. Une caméra fixée sur une plateforme articulée nous permettra de capturer réellement des images. Deux servomoteurs offriront quant à eux la possibilité d'orienter la plateforme en lacet et en tangage et seront contrôlés directement par le Raspberry PI. Ce dispositif constituera une abstraction d'un satellite AGILE dans lequel, contrairement à notre cas, c'est le satellite lui-même qui est orienté, la caméra restant fixe par rapport à celui-ci.

L'utilisation de l'association d'une caméra et de servomoteurs ouvre la porte aux thématiques Commande Adaptée et Traitement de l'Image, et la programmation orientée objet respectée depuis le début du projet, nous permet de simuler simplement et arbitrairement ces fonctions en attendant qu'elles soient réalisées par d'autres groupes.

III. 3. Extension du simulateur ARINC653

Le simulateur du noyau avionique temps réel ARINC653 doit pouvoir fournir et respecter le plus de fonctionnalités du standard en ce qui concerne la sûreté de fonctionnement. Les limites des efforts à fournir seront déterminées en collaboration avec Marie-Hélène Deredempt et Jean-Paul Blanquart, ingénieurs chez AIRBUS Defence & Space. En outre, on ajoutera aux fonctionnalités déjà existantes et dans la mesure du possible le redémarrage de partitions sur détection de faute.

III. 4. R&D sur l'utilisation de noyaux IMA dans les plateformes satellites

Le travail à effectuer sera déterminé en collaboration avec AIRBUS Defence & Space et tentera de faire murir la réflexion sur les problématiques d'utilisation des noyaux IMA dans les systèmes satellites.

III. 5. Traitement d'un plan de mission

Un autre groupe de projet long travaille sur la planification des tâches d'une mission dans le but de fournir un plan qui optimise les bénéfices par rapport aux demandes des clients. Nous devons donc être capables en fin de projet de traiter tout plan issu de leur algorithme de génération de plans.

IV. Master Copy

Comme expliqué précédemment, nous n'avons pas démarré notre projet long de zéro. En effet, nous avions développé une première solution en groupes de 3 personnes lors du BEI qui s'est étalé sur 3 mois avec environ une trentaine d'heures de conception et de développement. Comme tous les participants du projet long ne faisaient pas partie du même groupe lors du BEI, la première étape de notre travail a consisté à réaliser une Master Copy, ou en d'autres termes une réunion des codes en ne conservant que les avantages de chaque solution.

Ce travail fut assez long car les adaptations ont été nombreuses afin que toutes les visions du simulateur puissent interagir les unes avec les autres. De plus, certaines nouvelles fonctions ont été ajoutées (comme le transfert d'image vers une station sol) car elles n'avaient pas pu être implémentées précédemment par manque de temps. Finalement, cette dernière version du code du BEI pouvait être divisée en deux parties distinctes : le simulateur de noyau ARINC653 et la simulation d'un satellite AGILE.

IV. 1. Simulateur de noyau ARINC653

Le simulateur en lui-même avait été développé lors d'un précédent projet et il se compose de plusieurs partitions respectant le principe de Time and Space Partitioning (TSP) qui sera détaillé dans la partie suivante. Chaque partition correspond à un processus (ou thread) indépendant, et elle est matérialisée par une fenêtre générée en python à l'aide du paquet graphique TkInter. En réalité, ces objets ne sont que des intermédiaires puisque l'affichage des fonctions codées en C/C++ est transmis à ces fenêtres par l'intermédiaire de *pipe*.

La préemption des partitions dans cet environnement temps réel est quant à elle réalisée par l'envoi de signaux d'arrêt (SIGSTOP) et de continuation (SIGCONT) ce qui assure un ordonnancement en boucle des partitions en fonction du temps alloué à chaque processus. Cette durée est paramétrable, et dans le cas de la Master Copy, elle était de 500 ms.

Enfin, le partage de données directement entre partitions étant impossible, tous les échanges passent par des mécanismes de communications. Ils sont mis en place à l'aide de socket en local et peuvent être de deux types. Le queuing est proche des mécanismes FIFO (First In First Out) puisque tous les messages sont conservés dans la boîte et leur lecture dans l'ordre d'envoi est assurée. Le sampling s'apparente plus à un principe de "tableau noir" : seul le dernier message est conservé et il reste en mémoire tant qu'un nouveau message n'est pas envoyé sur le port.

IV. 2. Simulation d'un satellite AGILE

Un satellite AGILE est un système très complexe composé de nombreux services comme la gestion de l'énergie, l'orientation ou encore la prise de vues. Étant donné le nombre d'heures allouées au BEI et au projet long, seuls certains aspects faisaient effectivement partie de notre cahier des charges. Il s'agissait de simuler un satellite AGILE de prise de vues qui est capable de communiquer avec des stations sol (localisées sur Terre) et qui tolère les fautes par crash. Nous allons à présent détailler les fonctionnalités présentes dans la Master Copy.

Tout d'abord un plan de vol est lu. Il s'agit d'un fichier texte qui comporte au maximum 50 lignes d'action et dont la forme se fonde sur des plans de vol réels c'est-à-dire qu'il comporte pour chaque action : un numéro, un type, un identifiant, une heure d'exécution et des paramètres (par exemple les angles à appliquer au Système de Commande d'Attitude et d'Orbite (SCAO) et la durée de la prise d'image pour ce type d'action). Chaque plan est généré automatiquement grâce à un programme qui garantit la faisabilité de chaque action en maîtrisant leurs durées et qui permet de choisir la fréquence d'occurrence de chaque type d'action ou encore la plage de valeurs des paramètres (angles, nombre d'images, etc.). Puis, le satellite

exécute l'action donc la date n'est pas encore passée. Cette action peut être de trois types : prise d'image (noté IMG), transfert d'images (noté TSF) ou changement de plan (noté PLA).

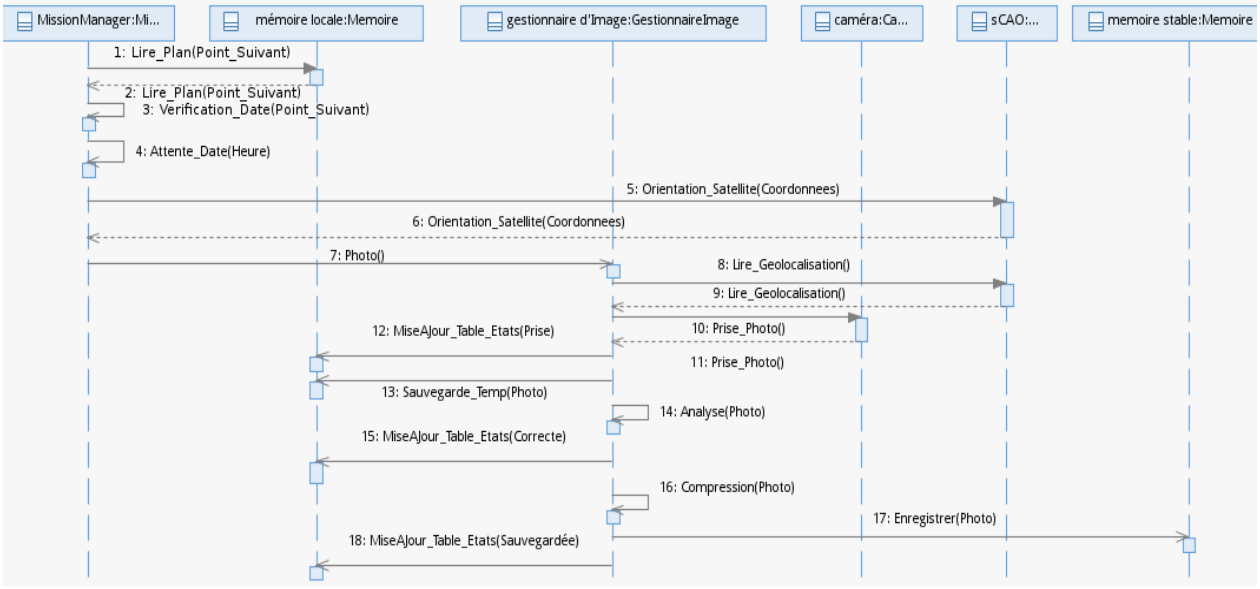
La prise d'image commence par l'orientation du SCAO selon les angles spécifiés dans le plan : son fonctionnement était simplement symbolisé par un message de confirmation comme aucune ressource matérielle ne nous permettait de simuler son comportement. Puis, le système patiente jusqu'à la date correspondant à la prise de vue. Il prend alors le cliché : une fois de plus, nous ne pouvions pas utiliser une véritable caméra donc cette action utilisait des images préenregistrées qui étaient copiées dans un dossier représentant la mémoire de la caméra. Chaque nouvelle image est tirée au hasard et est renommée selon l'identifiant mentionné dans le plan : cet identifiant est unique puisqu'il est correspond à la concaténation de la date (jour, mois, heure, minute et seconde) à laquelle la photo doit être prise. Ensuite, ce cliché est analysé afin de déterminer si sa qualité est suffisante en fonction de la couverture nuageuse. Une fois de plus, ce mécanisme se résumait à un tirage aléatoire : l'image avait 90% de chances d'être correcte. Dans tous les cas, la photo est compressée puis stockée dans la mémoire stable car même si elle est incorrecte, la décision finale de suppression revient au centre de contrôle. Tout cela se traduit par la création d'une archive *.zip et de la copie de cette dernière dans un autre dossier représentant la mémoire stable du satellite.

Le transfert d'image vers un centre de mission sur Terre via une station débute par une attente de la date exacte à laquelle l'action doit être réalisée. Puis les images (dont le nombre et les identifiants sont précisés dans le plan de vol) sont copiées du dossier de mémoire stable vers un dossier nommé station_sol. Entre chaque transfert d'image, le simulateur vérifie si la date limite d'exécution de l'action n'est pas dépassée afin de se rapprocher au plus de la réalité. En effet, le satellite ne peut réellement communiquer avec la Terre que lorsqu'il est en visibilité d'une station sol. Les actions de communication sont donc interrompues si le transfert dure plus longtemps que la durée maximale prévue qui est calculée par le centre de mission lors de la création du plan.
Enfin, lors d'un changement de plan, on génère automatiquement un nouveau plan dans le dossier prévu à cet effet. Cette action apparaît au rang 40 des plans ce qui garantit un déroulement en boucle du satellite qui ne sera jamais à court de plan.

En plus de ces aspects fonctionnels, le simulateur met en œuvre des mécanismes de sûreté de fonctionnement. Grâce au principe de redondance primary/backup, les fautes par crash sont tolérées : si la partition primary est amenée à se terminer, la partition backup prend alors le relais en reprenant les actions fonctionnelles telles que le primary les avait laissées.

Plusieurs stratégies sont possibles pour gérer la communication nécessaire au recouvrement, mais dans notre cas nous avons opté pour l'utilisation de checkpoint. Le primary envoie donc son avancement dans le plan de vol au backup afin que ce dernier sache depuis quelle ligne il doit redémarrer en cas de crash. Pour obtenir un recouvrement plus précis, nous avons aussi choisi de créer un vecteur d'état qui sera envoyé à chaque checkpoint : il est composé de 5 cases qui indiquent si l'état en question est vrai par le biais d'un booléen. Les états renseignent si la photo a été prise, analysée, jugée correcte, stockée et transmise. Ainsi, lors du recouvrement, la partition backup lira le dernier checkpoint reçu et aura donc connaissance du numéro de la dernière action effectuée, de l'identifiant de l'image concerné (ou du groupe d'images concerné pour une action de transfert) ainsi que de l'avancement de celle-ci par le biais de ce vecteur. Donc par exemple, si la partition primary plante alors que l'image a été seulement prise, le backup lira 10000 dans

le dernier checkpoint reçu, et procèdera à l'analyse et au stockage de celle-ci sans reproduire la capture d'image. En terme de communication, le mode utilisé pour l'envoi de ces points de reprise est le queuing. En effet, il est important de conserver tous les checkpoints : par exemple, les actions de transfert interviennent aléatoirement et donc elles ne sont pas consécutives aux images prises. Les notifications envoyées lors des actions de transfert concerneront alors des images qui ont été prises il y a plusieurs minutes, mais il est important de mettre à jour la dernière case de la table d'état. Le but est d'éviter que les images concernées soient transmises deux fois en cas de crash, ce qui gâcherait inutilement du temps dans la fenêtre de visibilité. Tous les checkpoints sont donc lus les uns après les autres et ils sont stockés après leur lecture dans une table ce qui assure leur conservation pour le plan en cours. En prenant en compte l'envoi des checkpoints, un diagramme de séquence d'une prise d'image peut alors se décomposer comme suit :



Mais pour être en mesure de réaliser le recouvrement, il faut pouvoir détecter un crash. Le code de la Master Copy réalise cela grâce à un watchdog : la partition primary envoie régulièrement un signal à la partition backup. Si le dernier signal reçu est plus ancien que le délai limite fixé par le watchdog alors le primary est considéré comme mort et le recouvrement est lancé. En pratique, ce signal est un simple message qui contient l'heure actuelle en seconde par rapport à une référence fixe. Ainsi, lors de la réception, le backup réalise une simple différence entre l'heure reçue et l'heure au moment de la réception et compare le résultat au décalage maximal admissible qui est de 2 secondes dans notre cas. Cette durée a été fixée en fonction du temps d'exécution au pire cas (WCET ou Worst Case Execution Time) qui se calcule en fonction du temps alloué à chaque partition (500 ms). Contrairement au cas des checkpoint, ici le mode de communication utilisé est le sampling. En effet, seule la valeur la plus récente nous intéresse puisque même si un message n'est pas lu entre temps, cette valeur sera suffisante pour savoir si la partition primary a effectivement crashé.

Concernant l'organisation du code, le développement a été réalisé en C++ et respecte donc les principes de la programmation orientée objet. Nous avions ainsi une classe pour chaque fonction principale. Les instances de ces classes sont appelées par les deux partitions : la partition primary est nommée Master et la partition backup se prénomme Slave. Dans un but de généricité et de simplification du programme, le

code de ces deux partitions est identique. Seul le mode (primary ou backup) change d'une partition à l'autre. Elles sont donc organisées de la même manière : un thread est créé pour gérer le comportement fonctionnel alors qu'un autre thread s'occupera des aspects non fonctionnels.

Le thread fonctionnel fait simplement appel à une instance de la classe Manager qui fait office de mission manager : dans le mode primary, elle lit le plan et réalise les actions détaillées ci-dessus alors que dans le mode back up elle ne fait rien. Quant au thread non fonctionnel, il permet à la partition primary d'envoyer régulièrement le signal au watchdog alors que la partition backup réceptionne ce message, vérifie qu'un crash n'est pas survenu et lit les checkpoints reçus en provenance du Manager et les stocke dans la table de checkpoints associée. En cas de crash, c'est ce thread qui permettra à la partition backup de réaliser le recouvrement sur la dernière action réalisée par le primary. Le mode du Manager exécuté dans le thread fonctionnel du Slave sera ensuite changé afin que celui-ci continue son déroulement classique en mode primary.

Cette structure à deux threads nous permet d'obtenir un certain parallélisme dans l'exécution des actions qui est nécessaire pour que les fautes soient tolérées. En effet, certaines actions durent plusieurs secondes comme l'attente d'une date. Cependant, il faut que le watchdog soit mis à jour pendant cette période sous peine de déclencher un recouvrement injustifié et donc de corrompre le fonctionnement du simulateur. Enfin, une temporisation de 500 ms (temps alloué à chaque partition) a été ajoutée dans le mode primary du thread non fonctionnel afin que la mise à jour du watchdog ne soit faite en moyenne qu'une fois par ordonnancement. Le but était de ne pas saturer le port sampling puisque nous avions constaté que la gestion des messages prenait énormément de temps.

En définitive, nous pouvons dire que cette version était assez avancée et répondait bien au cahier des charges du BEI. Son fonctionnement était lui aussi correct puisque les actions s'exécutaient comme nous le souhaitions et le recouvrement était très bien géré. Cependant, à l'image d'un satellite, ce sujet était très profond puisque de très nombreuses améliorations pouvaient être imaginées. C'est donc tout naturellement que plusieurs modifications sont venues compléter cette solution s'intégrant parfaitement au sujet de base.

V. Architecture matérielle et logicielle associée à la conception du simulateur

Un des objectifs de ce projet est l'émulation d'une plateforme satellite. Pour ce faire nous nous devons de rester fidèles aux architectures déjà en place ainsi qu'aux concepts de noyau avionique tels que présentés dans le standard 653. Dans un premier temps nous présenterons ici les spécificités hardware de la plateforme. Toutes ces spécificités ayant un impact direct sur la conception logicielle nous verrons ensuite quels ont été nos choix pour faire correspondre au mieux les deux architectures.

V. 1. Architecture matérielle

La plateforme est divisée en deux grandes parties. La première a pour mission de gérer les fonctions vitales du satellite. Elle comprend donc les organes de contrôle d'attitude et d'orbite (SCAO) ainsi que des périphériques de communication pour envoyer et recevoir des télécommandes. Une seconde partie est

consacrée à la charge utile, c'est l'ensemble des éléments nécessaires à la réalisation de la demande du client. Cette partie contient donc une caméra et de la mémoire dédiée au stockage des images. On dispose aussi d'un deuxième périphérique de communication dédié à l'envoi des télémesures.

Le SCAO est en fait un système de gestion de roues à réaction qui permet d'orienter le satellite en agissant sur les trois angles (Roulis, Tangage et Lacet). Ce contrôle fait l'objet de commande spécifique et dépend non seulement de la mission (orientation de la camera) mais aussi des contraintes thermiques liées à l'exposition solaire ou des télécommunications.

Les télécommandes (TC) sont des ordres brefs ne requérant que peu de bande passante à l'inverse des télémesures (TM) qui transmettent dans notre cas des images de tailles importantes. C'est pourquoi les deux modes de communication sont séparés. De plus, comme nous le verrons plus tard, leurs utilisations sont propres à des partitions différentes.

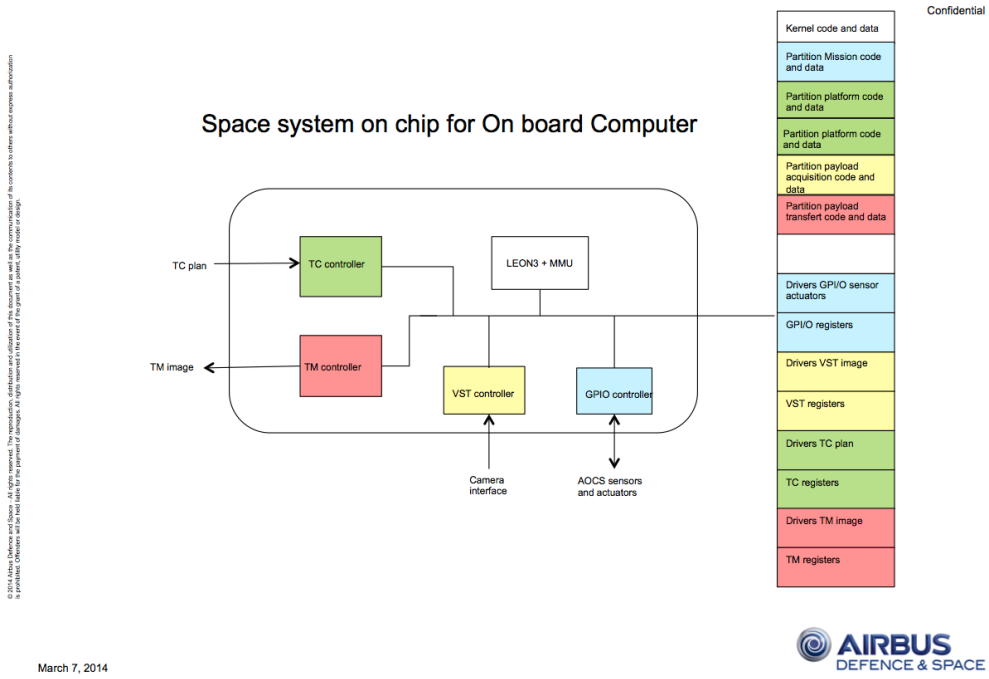
Une des grandes problématiques de l'implantation de noyau temps réel avionique est le respect du Time and Space Partitioning. En effet on souhaite travailler avec une ségrégation complète des différentes partitions logicielles utilisées. Malheureusement les solutions matérielles ne peuvent fournir des mémoires séparées physiquement pour chaque partition. Dans ce cas on utilise des gestionnaires de mémoire (Memory Manager Unit) capables de contrôler l'accès aux ressources. Le but étant ainsi d'empêcher une partition d'accéder à une zone mémoire qui ne lui est pas attribuée.

Ainsi on travaillera avec des ressources matérielles communes séparées logiciellement comme nous le verrons dans un second temps. On dispose d'ailleurs d'un seul et unique processeur ainsi que d'une seule mémoire pour l'ensemble des partitions. Ceci explique la présence nécessaire d'une sous-couche logicielle appelée Intergiciel (Middleware) capable de rendre la ségrégation possible.

Du point de vue hardware rien ne différencie un système partitionné d'un système temps réel. Les transmissions se font via des bus, le plus souvent répondant aux normes du standard MIL-STD-1553 qui assure une communication synchrone garantissant un temps de transmission maximal. Ces bus sont contrôlés par une entité logicielle séparée qui permet l'orchestration de l'émission et de la réception des messages sur le bus.

La représentation standard de ce type d'architecture est la suivante : le processeur est généralement de type LEON3 utilisant une architecture SPARC et synthétisable via VHDL pour des systèmes dits System-On-Chip (SOC). Ce processeur a été développé par l'Agence Spatiale Européenne et fait parti des standards pour la conception de satellites. Comme on peut le voir sur ce schéma le processeur a accès aux différents périphériques grâce à l'utilisation de bus dont les registres sont tous stockés dans une mémoire commune. En théorie donc tout programme fonctionnant sur le processeur peut accéder et utiliser ces différents organes. Or ceci est contraire aux règles de TSP. Ainsi on ajoute au processeur un ensemble de MMU capable de gérer l'accès à ces registres. C'est dans cette même mémoire que sont hébergés les codes et données propres à chacune des partitions.

Le schéma suivant nous donne une représentation de l'organisation d'un système intégré sur puce respectant les propriétés énoncées précédemment :



Ainsi c'est au software de faire le nécessaire pour s'assurer que le TSP est respecté et que le système est valide du point de vue de standards. C'est ce que nous allons voir dans la seconde partie.

V. 2. Architecture logicielle

Nous l'avons vu précédemment la gestion des contraintes du TSP se fait via les couches logicielles. La gestion des partitions est assurée par un noyau temps réel appelé middleware. Le rôle de ce noyau est de gérer l'arrêt et le redémarrage des partitions. Il connaît l'emplacement mémoire du code et du contexte relatif à chacune d'entre elles et doit donc décompresser le premier pour relancer l'exécution.

Ce changement d'état est en fait transparent pour la partition qui ne sait pas qu'elle est arrêtée. Elle s'exécutera sans se préoccuper du partage des ressources. En cas de dépassement mémoire c'est le noyau TR qui prendra la main et générera des exceptions qui seront interprétées dans la partition.

Le temps alloué à chaque partition est fixé lors de la configuration. Cette allocation est statique, un slot temporel ne peut être modifié ou supprimé. En cas de crash d'une partition, le slot ne peut pas non plus être réaffecté à une autre partition.

Pour le partitionnement spatial, on ne peut pas donner des droits d'accès à une même ressource à deux partitions. Ainsi si une partition P1 a accès à la caméra, donc aux registres de contrôle, une partition P2 ne peut en aucun cas y avoir accès. Ce cloisonnement nous oblige à mettre en place un système de communication par canaux. On établit ainsi des protocoles de communication interne en utilisant les ports libres de la machine et en leur associant des sockets dans la table des descripteurs. Tout ceci est géré par le middleware. Ces communications peuvent être de deux types, Sampling ou Queuing.

Tout ceci forme la base de l'architecture temps réel avionique telle que nous l'avons considérée. Dans notre cas nous utiliserons trois types de partitions:

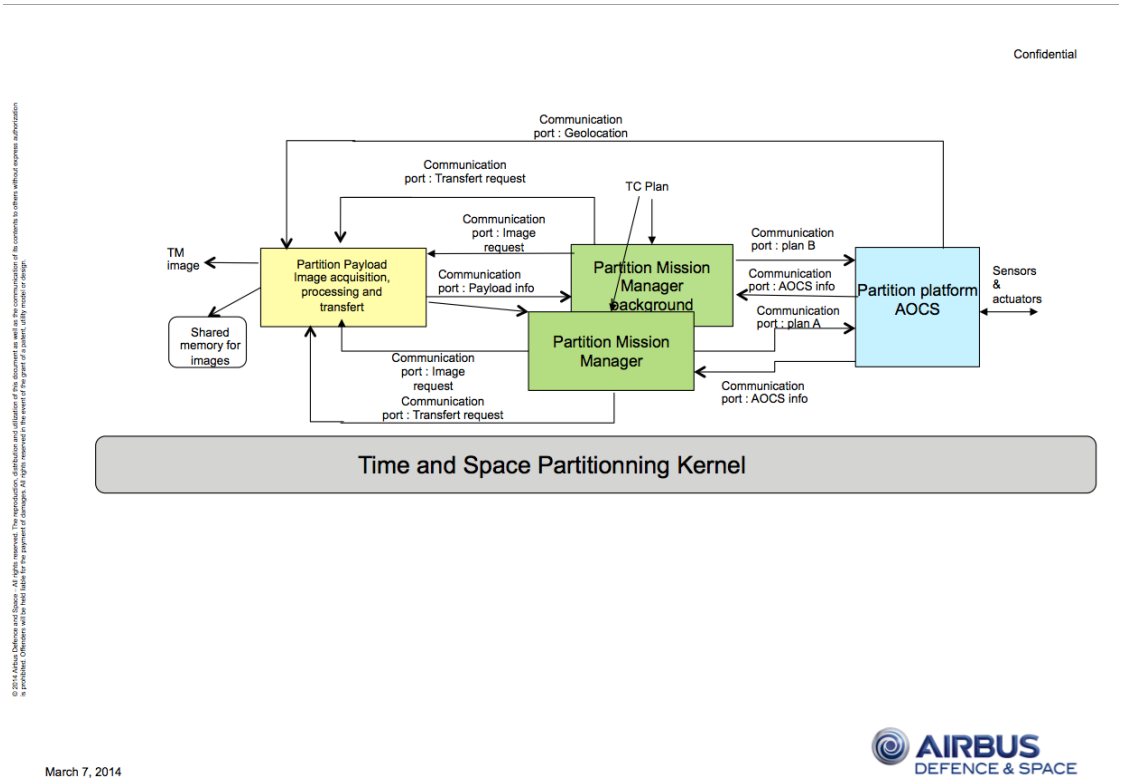
- un gestionnaire de mission
- une partition contenant le SCAO
- un gestionnaire de prise/envoi d'image

Dans le but de garantir la sûreté de fonctionnement de notre plateforme nous doublerons la partition gestionnaire de mission. Le mécanisme utilisé est appelé Primary Back-up Recovery, il s'agit d'une redondance tiède utilisant une partition dite primaire et une partition de sauvegarde qui peut reprendre les fonctionnalités du primaire en cas de crash.

Afin de garantir une plus grande sûreté, les deux partitions disposeront du même code source et seront lancées dans des modes différents lors de leur création. Cela garantit que des modifications du code soient automatiquement reportées sur les deux partitions.

L'architecture logicielle définit donc 4 partitions différentes dont les caractéristiques temporelles sont définies grâce à une étude temporelle réalisée a posteriori (cf VIII. Mesures).

La représentation suivante fait état de la façon dont s'organisent les partitions et quels sont les canaux de communications utilisés. La nature et l'utilité de chacun d'entre eux seront justifiées dans la suite de ce document.



On notera que les deux partitions de gestionnaire de mission utilisent une même ressource qui est le périphérique de télécommande par l'intermédiaire duquel ils reçoivent les plans. De prime abord cela semble contraire au principe de TSP. Toutefois le standard prévoit des fonctionnalités de contrôle du fonctionnement des partitions. Ainsi on est assuré que les deux partitions ne peuvent pas être simultanément en mode primaire et donc accéder à la même zone mémoire correspondant au périphérique de télécommande.

C'est en effet une particularité du modèle utilisé : il existe une partition propre au noyau qui s'assure du bon fonctionnement des partitions. Il est en effet prévu dans les standards que les partitions soient redémarrées lorsqu'un crash est détecté par la partition système. Ceci est un mécanisme en plus de celui mis en place entre les deux gestionnaires de mission.

Ces deux partitions communiquent entre elles pour utiliser un système de Watchdog. Ce processus a pour but de tester l'état de la partition primaire par la partition backup. Elles s'envoient également un ensemble de points de sauvegarde pour assurer une reprise efficace de la mission. Il semble logique que la partition système, lorsqu'elle relance l'une ou l'autre après un crash puisse choisir de la faire repartir en mode backup puisque la seconde est automatiquement passée en mode primaire.

Tout ceci doit faire l'objet de mesures temporelles poussées pour déterminer les temps de détection et de recouvrement que cette architecture logicielle impose.

Ces architectures logicielles et matérielles ont donc conditionné nos méthodes de travail ainsi que la direction à prendre pour le développement. L'intégration du projet sur une plateforme ARM ne garantit pas l'organisation matérielle souhaitée. Cependant le simulateur de noyau ARINC653 nous a permis de produire un travail cohérent avec les standards comme la partie suivante de ce rapport le montre.

VI. Extension du simulateur

VI. 1. Interface graphique

L'un des objectifs de ce projet long consistait à rajouter deux partitions chargées de gérer respectivement le SCAO (pour la partition appelée sobrement SCAO) et la prise, le stockage et l'envoi des images (partition se nommant Leica). Afin que l'exécution de notre programme soit plus lisible pour nous, il nous devenait donc nécessaire d'effectuer quelques ajustements au niveau de l'affichage. L'interface utilisateur étant programmée en python, nous avons ainsi eu l'occasion de découvrir ce langage.

L'interface graphique du programme se compose d'autant de fenêtres que de partitions créées, avec une fenêtre supplémentaire permettant de visualiser l'action du simulateur (l'ordonnancement des différentes partitions par exemple). Comme nous avons augmenté le nombre de partitions et donc le nombre de fenêtres à afficher, il nous a donc fallu faire varier la couleur habillant le fond des fenêtres afin de pouvoir nous y retrouver plus facilement. Chaque fenêtre se voyait donc attribuer une couleur en fonction de la partition qu'elle représentait. Nous avons aussi renommé le titre des fenêtres afin qu'il corresponde au nom de la partition associée pour plus de lisibilité.

De plus, comme la gestion du placement des fenêtres créées n'était pas vraiment effectuée, souvent au début de l'exécution les fenêtres s'empilaient et un repositionnement convénient était nécessaire. Nous

avons donc inséré un petit algorithme permettant, en fonction du nom associé à la fenêtre, de lui donner une position différente, de telle sorte que le comportement des 4 partitions était visible du premier coup d'œil. Enfin, afin de pouvoir toujours visualiser la dernière action en cours, nous avons rajouté un autoscrolling.

VI. 2. Nettoyage du simulateur

Lors de notre appropriation du code du simulateur, nous avons pu constater certains usages qui n'étaient pas conformes aux standards de la programmation ou qui n'assuraient pas une optimisation des ressources mémoires.

Nous avons ainsi par exemple regroupé la création des variables en début de code ; certaines de ces créations s'effectuaient entre autres au sein de boucles for ou while, afin de réinitialiser des vecteurs de variables par exemple. Nous avons opté pour une solution plus académique en vidant le vecteur avec la fonction *clear()*. Cependant, en nous renseignant sur cette fonction nous avons constaté qu'elle vidait effectivement le vecteur de son contenu, mais qu'elle ne le redimensionnait pas, ce qui n'optimisait pas l'utilisation de la mémoire. Nous déplaçons le vecteur vers lui-même avec la fonction *swap()* pour assurer une réinitialisation complète :

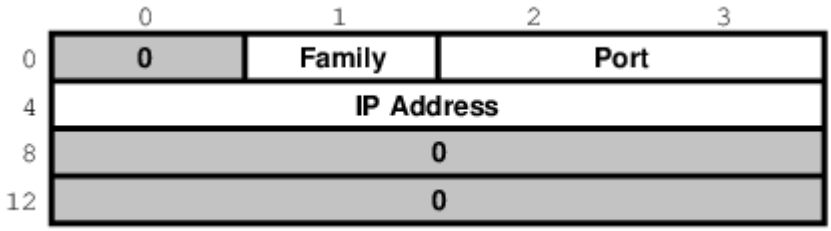
```
vWQportID.clear();

std::vector<int>(vWQportID).swap(vWQportID);
```

Nous avons aussi renommé certaines variables et certaines fonctions qui n'avaient pas un nom approprié à leur utilisation afin de permettre une meilleure compréhension du code par les élèves qui l'utiliseront.

VI. 3. Portage sur le Raspberry

L'objectif principal de notre projet long ayant été de faire fonctionner notre simulateur sur un Raspberry PI, une étape primordiale a été de rendre le simulateur de l'ARINC653 compatible avec la structure ARM de notre processeur.

L'incompatibilité du simulateur originel se trouvait au niveau de la communication entre les partitions, qui se fait par sockets. En effet, les sockets que nous utilisons gèrent les protocoles IP et utilisent la famille d'adresses AF_INET. Nous utilisons donc la structure fournissant les informations nécessaires au fonctionnement des sockets appelée sockaddr_in. Cette structure est organisée de la façon suivante :



Les champs importants sont sin_family, à l'octet 1 de la structure, sin_port, de longueur 16 bits stockée dans les octets 2 et 3, et sin_addr, un entier de 32 bits qui représente l'adresse IP, stockée dans les octets 4 à 7. Cependant, sin_addr ou sin_port ne sont pas stockés comme une seule entité de respectivement 32 et 16 bits mais comme des séquences de 4 et 2 octets. Contrairement à ce que nous avons pensé au début, le problème de venait pas de la différence "d'endianess" du Raspberry PI et du Linux car les deux étaient en Little Endian. En réalité, contrairement à un ordinateur sur Linux, lors de la création du socket le RaspberryPI remplit la structure avec les octets et récupère les informations de deux façons différentes ce qui fait que lors du remplissage de la structure de la sorte :

	0	1	2	3
0	0	2	0	13
4	192	43	244	18
8	0			
12	0			

Mais lorsqu'il est en situation de devoir récupérer les informations dans le socket, il interprète les informations de la structure d'une façon différente, ce qui implique qu'il est amené à croire qu'elle a été remplie de la façon suivante :

	0	1	2	3
0	0	2	13	0
4	18	244	43	192
8	0			
12	0			

Ceci fait donc que la lecture n'était pas effectuée dans le bon socket et qui produisait une erreur de type « bad file descriptor ». Pour résoudre ce problème nous avons donc utilisé les fonctions *htonl* (pour les données de format 32 bits) et *htons* (pour les données de format 16 bits) qui mettaient la structure dans le bon format :

```
struct sockaddr_in sa;
bzero(&sa,sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_addr.s_addr = htonl(INADDR_ANY);
sa.sin_port = htons(portID);
```

VI. 4. Gestion des partitions

Dans la Master copy que nous avions élaborée à partir des codes réalisés durant notre BE industriel, la sûreté de fonctionnement était assurée par le fait que pour la partition principale (la partition Master) une partition de secours était prête à prendre le relais en cas de défaillance. Cette protection n'était donc valable que pour une seule défaillance entraînant l'arrêt de la partition Master, ce qui nous paraissait un peu insuffisant. Nous nous sommes donc intéressés aux mécanismes de redémarrage de partitions car ils font parti des spécifications de la norme ARINC653.

VI. 4. 1. Redémarrage « manuel »

Ce type de redémarrage modélise un opérateur qui détruit une partition en fermant une fenêtre du simulateur. C'est donc une fermeture qui vient d'un acteur extérieur du programme et qui doit être correctement détectée et gérée.

Pour comprendre la pertinence des mécanismes de détection et de redémarrage mis en place, il nous faut faire un rappel rapide de comment sont créées et gérées les partitions. Lors du lancement du programme, le main lance le script python permettant de créer la fenêtre qui contiendra les informations qu'il enverra. Il exécute ensuite la fonction dans laquelle il restera toute la durée d'exécution du programme et qui permet de créer et de contrôler tous les autres processus.

La création de ces processus s'effectue en faisant un *fork*. Le *fork* a été privilégié par rapport à un thread car contrairement à ce dernier il a une zone mémoire indépendante, ce qui permet de bien simuler la TSP. A chaque création, le processus père enregistre le pid du processus qu'il vient de créer dans un tableau. Une fois créé, ce nouveau processus exécute ensuite un code différencié grâce à la fonction *execlp*. Cette fonction permet de sélectionner l'exécutable qui va être exécuté et lui fournit les argument initiaux, les argv[]. Si jamais l'appel de cette fonction se termine (ce qui ne devrait pas arriver dans un fonctionnement normal), un *exit()* permet de tuer le processus. Le processus principal contrôle l'arrêt et le redémarrage des autres processus grâce à un envoi de signaux à chaque partition.

Pour redémarrer une partition il faut tout d'abord détecter son arrêt, et cela ne pouvait être fait qu'au sein du processus principal. Cependant comme cette surveillance devait être permanente, il nous a fallu créer un nouveau thread. Nous avons fait le choix d'un thread et non d'un *fork* car nous souhaitions qu'ils partagent la même mémoire virtuelle pour pouvoir surveiller les processus qui ont été créés par le processus principal. Nous fournissons en argument de ce thread un objet de type pointeur de pid_t ; comme le thread exécute dans une boucle infinie un *waitpid* dont les arguments le configurent de telle sorte qu'il soit bloquant et qu'il attende la terminaison de n'importe quel processus, en enregistrant le pid obtenu en retour de *waitpid* dans ce pointeur, nous pouvons le transmettre au processus principal. Le processus principal, qui jusqu'à présent dans sa boucle infinie se contentait d'interrompre et de redémarrer les autres processus nés d'un fork, vérifie maintenant en amont de l'envoi de ces signaux si la valeur de la variable contenue dans ce pointeur a changé. Si c'est le cas, il compare le pid qu'il a obtenu avec les pid contenus dans le tableau des pid des processus qu'il a créés. Cela lui permet de connaître exactement quelle partition a été arrêtée et donc de la recréer de la même façon que décrit précédemment. Bien évidemment, il remplacera le pid ancien.

La possibilité de redémarrer une partition nous a poussés à nous interroger sur les modalités de ces redémarrages. En effet, il ne serait pas trop dérangeant que les partitions SCAO ou Leica redémarrent en reprenant de zéro car elles se contentent simplement de recevoir et d'exécuter des ordres, mais ce n'est pas le cas de Master. En effet, ce dernier démarre en mode primary, mais s'il est arrêté et a redémarré, le Slave prendrait le relai et serait déjà lui-même en mode primary, ce qui impliquerait que les deux partitions soient en mode primary ce qui n'est pas acceptable. Il faudrait au contraire qu'il démarre en mode backup afin de pouvoir surveiller le Slave. Mais la partition en démarrant ne sait pas qu'elle redémarre. Il nous a fallu donc instaurer le principe de mode de démarrage dans le simulateur. Le mode dans lequel une partition démarre ne sera donc plus décidé par le nom de la partition mais au lancement (ou au re lancement) par le simulateur.

VI. 4. 2. Redémarrage « automatique »

Nous avons souhaité mettre en place un autodiagnostic dans certaines partitions afin de laisser plus d'autonomie dans le satellite. Ces partitions pourraient donc repérer un dysfonctionnement interne et tenter un redémarrage pour essayer de le supprimer. Nous avions déjà mis en place la détection de fin de processus et le redémarrage. Il ne nous restait donc qu'à finir le processus correctement.

Dans un premier temps, on pourrait simplement imaginer que la terminaison d'un processus revenait juste à la fin du code exécuté par *exclp*. Le problème est que si le processus était effectivement terminé par cette méthode, ce n'est pas pour autant que le script python créant la fenêtre où les affichages de ce processus s'effectuent se terminerait ; si la fermeture de la fenêtre termine le processus, l'inverse est faux. Ainsi, le redémarrage de la partition ouvrait une autre fenêtre sans fermer l'autre, ce qui entraînait des problèmes d'affichage des messages. Il nous fallait donc trouver le moyen de récupérer le pid du script python qui était lancé par chaque processus afin de le tuer juste avant le redémarrage.

Le problème que nous avons rencontré à cette étape est que tous les scripts pythons lancés par les processus fils avaient le même nom, « gui.py », ce qui compliquait pour les partitions la récupération du pid du script python qui leur était associé.

Nous connaissions le moyen d'afficher les pid associés aux processus « gui.py » au moyen de la commande système ps -eo pid,command | grep \"gui.py\" | grep -v grep | awk '{print $1}' mais après avoir lancé cette commande système il nous a fallu trouver le moyen de récupérer la réponse dans le programme. Pour cela nous ouvrons un flux quand nous exécutons cette commande système et nous le vidons dans une variable au fur et à mesure jusqu'à ce qu'il soit vide :

```
char cmd[256]= "ps -eo pid,command | grep \"gui.py\" | grep -v grep | awk '{print $1}'";

if ((ptr = popen(cmd, "r")) != NULL)
        while (fgets(buf1, BUFSIZ, ptr) != NULL)
                (void) printf("%s", buf1);
        pclose(ptr);
else
        fprintf(stderr, "Echec de popen\n");
```

Dans notre cas, comme nous avons placé la récupération des pid juste après le lancement du script python par la partition, le dernier pid récupéré est le bon.

Cette solution fonctionne très bien sur Linux, mais nous avons constaté que ce n'était pas le cas sur le Raspberry PI. En effet, toutes les partitions, normalement créées l'une après l'autre par le processus père, voyaient les pid de l'intégralité des scripts pythons lancés par toutes les partitions du programme. Nous ne pouvions donc pas nous contenter de prendre le dernier récupéré. Nous avons donc été obligés de donner en *argv* du main de chaque partition la position du pid qu'elle devait récupérer et une indication qui

renseignait si elle était en redémarrage ou non. Si c'est la première fois qu'elle est lancée, elle récupère le pid selon sa position sinon elle récupère le dernier. Cette façon de procéder nous semble moins juste car elle implique de signifier à la partition qu'elle a redémarré et son ordre de lancement. Nous avons donc fait varier la solution utilisée selon le nom de la machine exécutant le programme; s'il est « raspberryPI-TSF » le pid est obtenu par la seconde solution, sinon par la première.

VII. Intégration sur Raspberry PI et améliorations associées

VII. 1. Présentation du Raspberry PI

Le Raspberry PI est un nano-ordinateur à processeur ARM. Bénéficiant d'une faible consommation, les processeurs ARM sont devenus dominants dans le domaine de l'informatique embarquée, en particulier dans la téléphonie mobile et dans les tablettes. Ces processeurs ont la particularité d'être à architecture RISC (*Reduced Instruction Set Computer*). Comme son nom l'indique, elle propose un jeu relativement réduit d'instructions simples, faciles à décoder, et qui peuvent être exécutées chacune le plus souvent en un seul cycle d'horloge : le pipeline est utilisé de manière optimale et le processeur est de faible consommation. L'architecture RISC offrant à puissance égale un microprocesseur de plus petite taille, elle a permis de développer des SoC (*System on Chip*) permettant d'agglomérer l'ensemble des processeurs et contrôleurs sur une seule puce, sur une surface dont le côté avoisine 1 à 2 cm. Cela permet d'obtenir un ordinateur de dimension très compacte, inférieure à 10 × 6 cm.

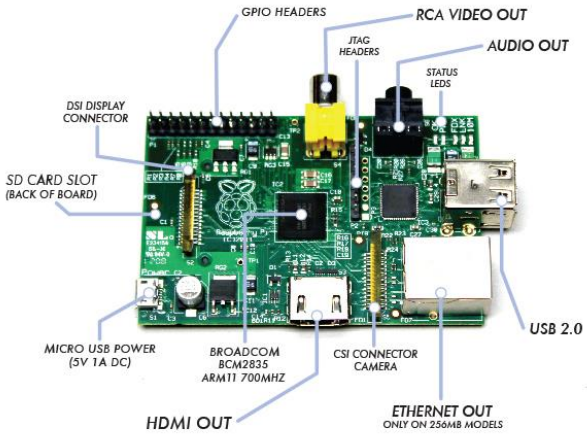
Ce nano-ordinateur qui a ainsi la taille d'une carte de crédit nous permet de travailler avec un processeur taillé pour l'embarqué. Il est dans sa version de base fourni nu (carte mère seule, sans boîtier, alimentation, clavier, souris ni écran) dans l'objectif de diminuer les coûts et d'encourager l'utilisation de matériel de récupération. Il est sans doute le moins cher au monde puisqu'il ne coûte que 25€. Outre l'aspect économique, on lui trouvera donc au moins trois intérêts : le côté ludique de l'apprentissage de la programmation qui est mis en avant, incitant à la manipulation et à la débrouillardise, l'encouragement de la jeunesse à la programmation, et l'accès facilité (car « low cost ») à la programmation par les pays qui n'en avaient pas nécessairement les moyens auparavant.

Nous disposons du modèle plus complet B revision 2 : le processeur est un Broadcomm ARM1176JZF-S (ARMv6) cadencé à 700 Mhz associé à 512Mo de RAM (256 pour la version de base). Nous l'avons rapidement overclocké à 900 MHz sans aucun souci. Il peut monter jusqu'à 1100 MHz. Son GPU BMC Videocore IV permet de décoder des flux Blu-Ray full HD (1080p 30 images par seconde). Il est suffisamment ouvert et puissant pour permettre une grande palette d'utilisations : on trouve de nombreuses idées et beaucoup d'inspiration sur le net, allant de l'émulation de consoles vidéos à de la télésurveillance, en passant par le classique MediaCenter.

Le Raspberry PI propose en outre une sortie vidéo HDMI, une RCA (composite), une sortie audio Jack 3.5mm, deux ports USB, une interface Ethernet 10/100Mbits, un lecteur de carte SDHC / MMC, carte qui fait office de disque dur hébergeant OS et données. Dans notre cas, le Raspberry PI tournera sous une des distributions de Linux faites pour lui, Raspbian, dérivée de Debian. Mais il y a plus intéressant pour nous. On trouve des ports GPIO (*General Purpose Input Output*, en français entrée/sortie pour un usage général) offrant une interface complètement configurable pour générer plusieurs types de signaux de contrôle de

périphériques, utiles par exemple pour piloter les servomoteurs en notre possession. La carte accueille aussi un port CSI (*Camera Serial Interface*) utilisé pour connecter le module caméra à notre disposition.

Nous avons donc tous les outils nécessaires pour réaliser une véritable maquette de simulation d'un satellite AGILE grâce à « ce couteau suisse électronique » qu'est le Raspberry PI.



VII. 2. Ajout de partitions et de leurs communications associées

La Master Copy abrite les fonctions nécessaires au bon déroulement de la mission du satellite au sens « objet » : beaucoup de méthodes existent mais leur code source n'est qu'élémentaire et ne réalise pas encore les fonctionnalités associées. C'est le cas par exemple de la méthode *orienter* de l'objet *SCAO* qui pour l'instant ne réalise qu'un affichage des paramètres reçus en entrée. Avec une maquette à disposition, il devient nécessaire de compléter certaines de ces méthodes plus ou moins « vides ».

Dans la Master Copy, c'est l'objet *Manager* présent dans les partitions MASTER et SLAVE qui se sert directement des objets du satellite pour accomplir la mission. Il intervient donc à la fois sur le côté plateforme avec le contrôle du satellite et sur le côté mission en se servant de la charge utile à l'observation de la Terre. Ceci pose au moins deux problèmes qui provoquent le même conflit avec le standard ARINC653 :

- Une partition a accès à toutes les ressources du satellite, ce qui fait que si cette partition rencontre un problème, il y a un risque que celui-ci se propage dans tout le satellite
- Les deux partitions MASTER et SLAVE ont accès par exemple à la fois à la caméra et à la mémoire stable, et elles gèrent les images. Même si c'est peu probable, il se pourrait qu'à cause d'une faute de développement, le MASTER et le SLAVE se retrouvent en mode PRIMARY en même temps. Même si l'ordonnanceur impose une seule partition active à la fois, on ne sait pas quelles conséquences ce cas particulier pourrait avoir sur la mémoire stable ou sur les composants, surtout s'ils reçoivent des ordres contradictoires.

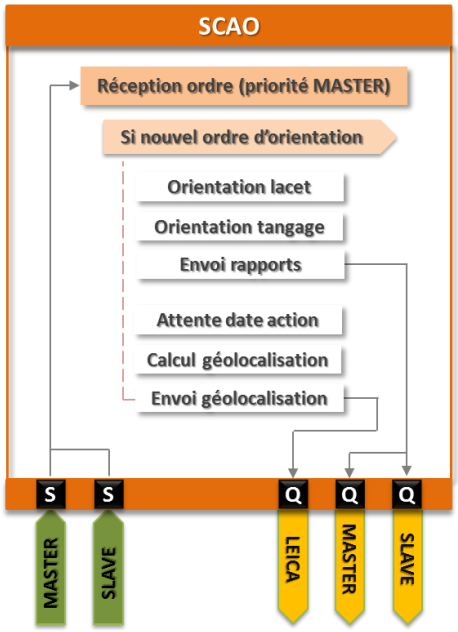
Dans ces deux cas au moins, le partitionnement spatial du standard ARINC653 n'est pas suffisamment respecté. Les objets comme le SCAO ne doivent traiter qu'un seul ordre à la fois, et doivent pouvoir filtrer deux ordres consécutifs et identiques. Il devient nécessaire d'isoler chaque fonctionnalité du simulateur du satellite dans une partition pour que le système soit sûr de fonctionnement.

Les deux partitions MASTER et SLAVE de la Master Copy jouant alors le rôle de centre nerveux du satellite, celle qui est en mode PRIMARY se contente d'envoyer des ordres aux bonnes partitions qui elles les exécutent et renvoient un rapport aux Managers des deux partitions MASTER et SLAVE. Deux partitions vont donc réaliser respectivement les fonctions de contrôle d'attitude et d'orbite du satellite (partition SCAO) et de prise, analyse, stockage et transmission d'image (partition Leica). La partition en mode PRIMARY envoie des ordres de capture, de transfert ou d'orientation à la partition correspondante et envoie l'avancement de la mission à sa partition BACKUP.

Des canaux de communication de chaque type (Sampling et Queuing) sont ouverts dans les deux sens entre chaque partition selon la nécessité.

VII. 2. 1. Ajout de la partition SCAO de la plateforme

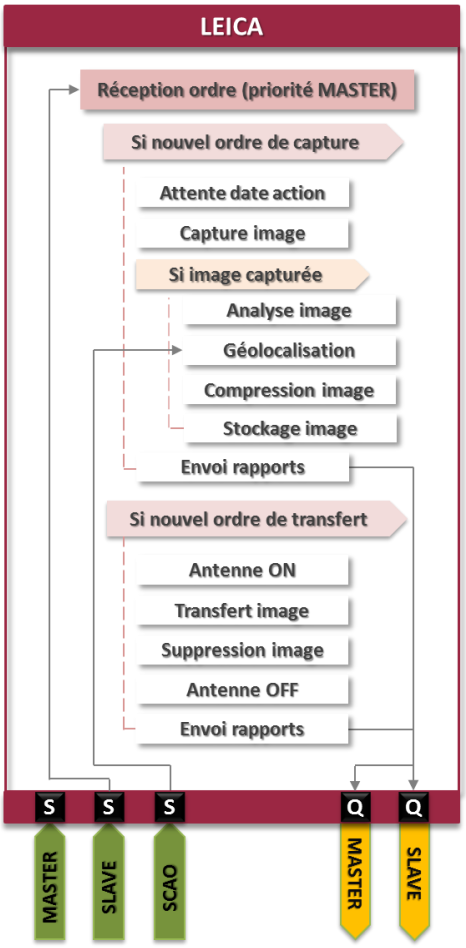
La partition SCAO héberge les fonctions de la « plateforme » : aujourd'hui elle permet d'orienter le satellite selon des angles en lacet et en tangage et de fournir la géolocalisation du satellite, mais à l'avenir elle pourra offrir d'autres fonctions liées au contrôle du satellite comme par exemple la commande adaptée des moteurs, ou le module StarTracker.



Après une phase d'initialisation, la partition SCAO entre dans un régime permanent de lecture et d'exécution d'ordre. En début de cycle, elle consultera d'abord le port Sampling en réception du MASTER pour vérifier s'il y a une demande d'orientation du satellite. L'orientation obtenue est comparée à la dernière orientation reçue (du MASTER) ainsi qu'à l'orientation actuelle. Si elle est nouvelle, elle est traitée, autrement elle est ignorée. Si aucune nouvelle orientation n'est reçue depuis le MASTER, on lit alors le port Sampling en réception depuis le SLAVE, et on traite la demande d'orientation si elle est nouvelle. Autrement on ignore la requête. Puis on reprend depuis le début du cycle après avoir marqué une petite pause pour éviter de consulter trop souvent et inutilement les ports Sampling, car ils ne seront pas remis à jour au moins tant que la partition SCAO est active (une seule partition active à la fois).

Si une nouvelle requête est reçue, on en extrait les valeurs des angles en lacet et tangage, puis on oriente le satellite selon ses valeurs. La partition envoie ensuite un rapport de l'exécution de la requête au MASTER et au SLAVE, puis elle attend la date de prise d'image pour déterminer la géolocalisation du satellite qu'elle enverra enfin sur son port Queuing en écriture vers la partition Leica.

VII. 2. 2. Ajout de la partition Leica de charge utile



La partition dénommée Leica accueille les fonctions concernant l'obtention et la gestion des images. Etant la seule à avoir accès à la mémoire stable pour des raisons de partitionnement spatial, elle s'occupe également de la transmission des images aux stations sol. C'est cette partition qui traduit la mission d'observation du satellite.

La partition Leica suit le même principe que la partition SCAO concernant la lecture et le traitement d'ordre en provenance du MASTER ou du SLAVE. A chaque cycle, elle vérifiera si elle a reçu un nouvel ordre en donnant la priorité à la partition MASTER.

Si un nouvel ordre est reçu, on le traite selon sa nature qui pour l'instant peut être de deux sortes : action prise d'image IMG, ou action transfert d'image TSF.

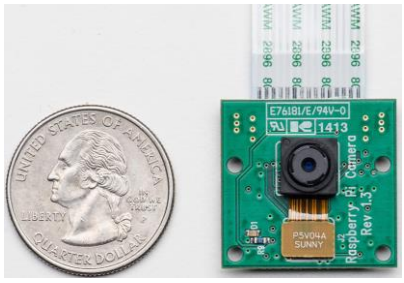
S'il s'agit d'un ordre de capture d'image, on attend la date de l'action puis on demande à la caméra de prendre l'image. Si l'image est bien capturée, on l'analyse puis on consulte le port Sampling en réception depuis le SCAO pour récupérer la géolocalisation de l'image. Une fois l'image marquée, on la compresse puis on la stocke dans la mémoire stable.

S'il s'agit d'un ordre de transfert d'image, on active l'antenne de la carte de communication, on lance la procédure de transfert d'image, on choisit ensuite de supprimer l'image de la mémoire stable puis on désactive l'antenne.

Peu importe l'ordre, à la fin de l'exécution, on envoie un rapport aux Managers sur les ports Queuing en écriture vers MASTER et SLAVE.

VII. 3. Ajout d'un module caméra

L'ajout d'une caméra représente une réelle amélioration depuis la Master Copy, qui ne manipulait que des images préenregistrées. Le module caméra en question est développé spécialement pour le rapsberry PI. Ses dimensions sont 8.5 x 8.5 x 5 mm (soit l'équivalent d'une pièce de monnaie) pour un poids de 2.4 g. Elle peut filmer en haute définition (jusqu'à 1080p) et prendre des photos de résolution 2592x1944 avec son capteur de 5 mégapixels. Elle se connecte directement au port CSI (interface caméra) de la carte via un câble PCB. Aucun driver n'est nécessaire puisque le firmware supporte de base la gestion de ce module. Seule son activation est nécessaire, ce qui aura pour effet d'allouer suffisamment de mémoire GPU à la caméra afin qu'elle puisse fonctionner correctement.



Concernant la fonction qui nous intéresse, à savoir la prise de photo, elle s'effectue simplement à l'aide de la commande système *raspistill* à laquelle on peut ajouter un certain nombre d'options comme la durée de la prise de vue (en réalité le module fonctionne pendant un certain temps et ne conserve que la dernière image), le nom du fichier de sortie, la qualité, la longitude et latitude au moment de la capture ... Son intégration a donc été très simple puisqu'il a suffi de remplacer la commande système de copie par celle-ci.
Concernant les données GPS, elles ont été intégrées aux photos par le biais de balises métadonnées utilisant la spécification EXIF. Le but serait d'ajouter à chaque image les coordonnées GPS correspondant à la géolocalisation du satellite au moment de la prise de vue. Cependant, nous ne sommes pas en mesure de calculer ces coordonnées puisque nous ne connaissons pas la position du satellite à un instant donné. Nous avons alors fait le choix d'associer des coordonnées fixes à chaque image bien qu'il aurait été possible de les déterminer aléatoirement. Ces coordonnées (43°33'48.3"N 1°28'38.8"E) correspondent à la position GPS de la salle du LAAS dans laquelle nous avons travaillé tout au long de notre projet long.

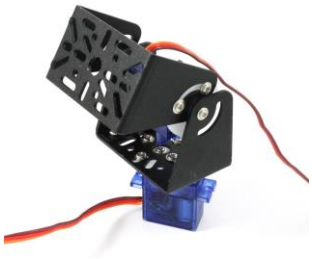
Néanmoins l'ajout de ces données de géolocalisation dans l'image se fait en deux temps. En effet, c'est théoriquement la partition SCAO qui communique à la partition Leica les données GPS actuelles. Nous avons donc considéré que ces données étaient ajoutées après la capture afin de ne pas compromettre la prise d'image par une communication trop lente avec le SCAO. Ainsi, dans un premier temps, l'image est prise avec des coordonnées GPS par défaut (pour permettre leur modification par la suite), puis après réception des coordonnées par le SCAO (qui dans notre cas sont toujours les mêmes, mais qui peuvent faire l'objet d'une implémentation future), elles sont intégrées grâce à la commande *exif* (EXIF Tag Parsing Library). (Cette même commande permet aussi de lire les données EXIF d'une image.)

Ce module a été testé avec succès et nous pouvons dire que l'ajout du module caméra a été une étape importante du projet afin de se rapprocher au mieux du comportement du satellite.

VII. 4. Intégration des moteurs

L'ajout des moteurs représentait aussi une étape très importante du projet en vue du parallèle avec un satellite. Le matériel utilisé était le Dagu Pan/Tilt Kit qui est composé de deux armatures et de deux servomoteurs. Une fois monté, cela permet d'obtenir des rotations de 0 à 180° selon les axes de lacet et de tangage.

Ces moteurs comportent un fil d'alimentation, un fil de masse et un fil de commande. Ils supportent une tension de 4,5 V à 6 V, donc nous les avons alimentés directement avec les ports GPIO 2-1 et 2-2 du Raspberry PI qui fournissent une tension de 5V. Il en va de même pour le fil de masse puisque 5 ports se prêtent à cette utilisation. Quant à la commande, ces servomoteurs sont pilotés en Modulation de Largeur d'Impulsion (MLI ou PWM en anglais) c'est-à-dire qu'une impulsion est envoyée à la commande du moteur à intervalles réguliers (dans notre cas environ tous les 20 ms pour une fréquence proche des 50 Hz) et que la position imposée au moteur est déterminée par la durée (ou largeur) de cette impulsion. Cela permet notamment de forcer les moteurs à un certain angle sans pour autant faire tourner le moteur en continu.



Le Raspberry PI est capable de générer un signal utilisable en MLI grâce à 1 des 26 ports GPIO. Cependant, chaque moteur doit être piloté indépendamment et doit donc posséder son propre signal de

commande, nous n'avons donc pas utilisé cette solution. À la place, nous avons opté pour une génération MLI semi-logicielle qui donne accès à beaucoup plus de contrôle. En effet, plusieurs ports peuvent être programmés pour envoyer un signal dont les caractéristiques sont définies par l'utilisateur. De nombreux projets qui permettent de générer ces signaux sont disponibles. Nous nous sommes tournés vers pi-blaster (de Tomas Sarlandie et Michael Vitousek) car il consomme très peu de ressource processeur et se veut très précis dans la génération d'impulsion, ce que nous avons pu vérifier grâce à un oscilloscope. Il est basé sur un autre projet, ServoBlaster de Richard Hirst, qui pour commander des servomoteurs utilise un principe très intéressant : il créé un buffer circulaire dans lequel il stocke les valeurs que doit prendre la pulsation. Puis ce buffer est envoyé aux ports concernés via le procédé d'accès direct à la mémoire (DMA) ce qui garantit sa rapidité. La génération PWM du Raspberry PI impose alors le déclenchement des commandes toutes les 10 µs. En associant à cela un buffer de 2000 points, on peut alors contrôler pleinement un signal de 20 ms c'est-à-dire de fréquence 50 Hz.

En pratique, le projet pi-blaster se présente sous la forme d'un programme codé en C, à partir duquel on créé un exécutable qui est par la suite suffisant pour commander la génération de signaux. L'interface entre l'utilisateur et les impulsions est gérée grâce à un fichier FIFO créé dans le dossier */dev* : dès qu'une commande de la forme <Num_PIN>=<%PWM/100> est écrite dans ce fichier, le signal associé est généré sur le port Num_PIN. 8 ports différents peuvent être utilisés et la fréquence peut être modifiée (dans notre cas, elle valait bien entendu 50 Hz).

Comme les ordres lus dans le plan sont des angles en degrés, nous avons créé une fonction qui permet d'envoyer une commande au moteur et dont les paramètres d'entrées sont des angles. Pour cela, nous avons testé manuellement quelles valeurs envoyées aux moteurs correspondaient aux butées de celui-ci. Ces valeurs étaient bien sûr différentes pour chaque servomoteur. Ensuite, une interpolation linéaire permet de déduire la valeur à appliquer en fonction de l'angle souhaité et vice-versa.

Enfin, nous avons pu noter durant nos tests que certains angles faisaient vibrer les servomoteurs. Cela se produisait proche des butées, et c'est pour cette raison que nous avons mis en place une sécurité. Ainsi, seuls des angles entre 5° et 175° peuvent être appliqués même si en théorie, les angles sont introduits dans le plan de manière à ne pas dépasser les valeurs limites.
Cependant, ces vibrations intervenaient aussi aléatoirement et nous avons associé cela à une torsion mécanique entre la fixation des moteurs et la commande luttant pour imposer un angle. Pour pallier ce problème, nous avons ajouté une nouvelle sécurité : après chaque envoi de commande, nous demandions l'arrêt des moteurs. Cette commande était envoyée après un certain temps afin que l'on soit sûr que les moteurs sont bien placés. En pratique cela se matérialise par une temporisation de 75 ms, valeur définie par tests, qui est la même que la temporisation utilisée entre l'envoi de commande sur deux servomoteurs différents.

Finalement, nous obtenions un placement des moteurs très rapide (quelques ms) et précis. Notons tout de même les quelques problèmes d'alimentation que nous avons connus au cours de l'intégration des moteurs. En effet, au début du projet, le Raspberry PI se figeait pour la plupart des commandes envoyées aux moteurs et nous étions obligés de le redémarrer. En faisant quelques recherches, nous nous sommes rendu qu'il s'agissait d'un problème connu puisque les périphériques qui demandent beaucoup de courant causent cet arrêt. Nous avons donc fait une grande majorité des tests avec une alimentation externe. Cependant, nous faisions fausse route, car après des tests plus poussés, nous nous sommes rendu compte

qu'il s'agissait en réalité d'un problème de câble d'alimentation : le câble micro USB utilisé initialement ne fournissait pas assez de courant au Raspberry PI. Après remplacement, nous avons été en mesure d'alimenter correctement les deux servomoteurs sans alimentation externe en utilisant une prise USB ou un adaptateur secteur. Il s'agissait donc que d'un problème de matériel défectueux, mais qui nous a coûté beaucoup de temps.

VII. 5. Communication Bord / Sol

Afin de se rapprocher le plus possible d'un satellite en tant qu'entité autonome, il était aussi important que les actions sensées être réalisées par la station sol et le centre de commande soit effectivement faites par une autre machine que le Raspberry PI.

La première modification était liée à la réception de plan puisque le programme se contentait de lire le plan présent dans un dossier. Nous avons alors procédé en deux étapes. Tout d'abord, nous avons créé un plan par défaut qui est exécuté si aucun autre plan ne peut être chargé. Il ne comporte qu'une seule action d'attente ce qui signifie que le comportement du montage lors de cette action est facilement paramétrable au sein même du Mission Manager. Nous avons choisi arbitrairement de faire bouger le moteur lié à l'axe de lacet d'un bout à l'autre de la plage d'angles disponible durant cette action d'attente. Puis, nous avons implémenté la réception de plan : dès que le plan en cours (par défaut ou non) est terminé, le contenu du dossier lié aux plans est scanné à la recherche d'un nouveau plan (fichier NewPlan.txt). Il suffisait alors d'envoyer un nouveau fichier plan dans le dossier adéquat pour que celui-ci soit chargé, et ce depuis n'importe quelle machine ayant accès au Raspberry PI. Cet envoi était bien souvent réalisé à l'aide du protocole SSH (commande *scp*). Toujours dans l'optique de respecter au mieux les principes de la norme ARINC653, nous avons localisé cette action de réception dans un nouvel objet de communication. Le but était de bien différencier les actions de réception de plan et d'envoi d'images qui ne sont pas exécutées par les mêmes partitions.

La deuxième modification concernait l'action d'envoi, à savoir dans notre cas, le transfert d'images. Auparavant elle était symbolisée par une simple copie entre le dossier de mémoire stable et un dossier représentant la station sol. Nous avons donc remplacé cette action par un déplacement SSH vers un ordinateur distant. Comme cette commande nécessite un mot de passe, nous avons fait le choix de faire un échange de clés RSA (utilisées pour sécuriser le protocole SSH) entre le Raspberry PI et la machine faisant office de station sol. Ceci permettait de ne pas interrompre le déroulement du programme en obligeant l'utilisateur à rentrer un mot de passe.
Du côté de la machine faisant office de station sol, un script shell a été développé pour surveiller l'arriver de nouvelles images. À chaque réception, les images sont alors décompressées puis affichées à tour de rôle.

Notons enfin que deux moyens de communication ont été mis en œuvre sur le Raspberry PI à savoir une liaison filaire (ethernet) et une liaison sans fil (Wi-Fi). On peut donc pousser le partitionnement jusqu'à utiliser un mode de communication différent pour l'envoi d'images et la réception de plan. Il suffit alors d'utiliser les adresses IP correspondant aux machines propres à chaque connexion.

VIII. Mesures

Afin d'évaluer les performances de notre solution, mais aussi pour ajuster au mieux certains paramètres temporels (temps alloué aux partitions, délai limite du watchdog, etc.), nous avons réalisé certaines mesures. Elles ont toutes été réalisées sur le Raspberry PI en sachant que le lancement du simulateur s'effectuait "à distance" via SSH. Cela peut donc justifier certains temps qui paraissent importants étant donné le peu de ressources que possède cet ordinateur miniature.

Tout d'abord, nous avons débuté par l'ordonnancement car même si le simulateur avait été auparavant testé grâce à de nombreux cas d'utilisations, il nous semblait important de vérifier que le temps alloué à chaque partition était bien respecté. En effet, le simulateur avait subi beaucoup de modifications depuis ces phases de test et le nombre de partitions contrôlées avait doublé.
Comme nous l'avons dit précédemment, l'arrêt et la continuité des partitions sont réalisés grâce aux signaux SIGCONT et SIGSTOP. Nos premières mesures ont donc été réalisées à l'aide de prise d'heure (fonction *gettimeofday* de la library sys/time.h) avant ou après l'envoi de ces signaux selon les cas. Les résultats étaient alors moyennés sur un nombre élevé de cycles (100 à 200). Ainsi, nous avons pu vérifier que le temps alloué à chaque partition était effectivement de 250 ms avec un dépassement en moyenne de 5 ms ce qui est une bonne performance. Concernant le temps interpartition, c'est-à-dire entre l'envoi du signal SIGSTOP à la partition i et l'envoi du signal SIGCONT à la partition i+1 (modulo 4), nous avons constaté qu'il était négligeable puisqu'égal en moyenne à 5 µs. Enfin, le temps d'envoi des signaux était lui aussi très faible puisqu'égal en moyenne à 20 µs pour SIGCONT et à 37 µs. Ces trois dernières mesures nous permettent de faire une hypothèse importante : il est possible de considérer la durée d'un cycle d'ordonnancement comme égale à 1 s (4x250ms) car les durées d'activation d'une partition à l'autre sont extrêmement faibles (inférieurs à 1 ms !). Cette remarque est très utile pour l'interprétation des mesures suivantes.

Après cette évaluation du simulateur en lui-même, nous nous sommes intéressés aux mesures directes à l'intérieur des partitions. Notre première idée était de capturer les signaux envoyés à chaque partition afin d'avoir une mesure précise du réel temps nécessaire à chaque partition pour effectuer ses actions. Cependant, bien que la manipulation du signal SIGCONT était aisée notamment grâce aux fonctions *signal* et *sigaction* de la library *signal.h*, il en était autrement pour le signal SIGSTOP qui par définition ne peut être capturé. Nous avons alors tenté d'envoyer un autre signal au même moment que ce dernier était envoyé, mais sans grand résultat. En effet, peu importe le signal que l'on envoyait, sa capture n'avait pas lieu et la partition en question subissait bien souvent un crash puisque le processus associé était tué. Nous nous sommes alors résolus à faire des mesures sans tenir compte de la préemption de chaque partition. Pour cela, nous avons placé des prises d'heures autour des actions principales de chaque partition ainsi qu'en début et en fin pour obtenir la durée totale d'exécution d'une action. Les temps donnés ci-après doivent donc être interprétés en tenant compte de la durée d'un cycle total (1 s).

Le but principal de ces mesures était de savoir dans quelle mesure nous pouvions modifier les temps alloués à chaque partition en fonction de la charge de chacune d'entre elles. En effet, la partition SCAO par exemple semble avoir besoin de moins de temps que les autres pour exécuter ses actions étant donné la rapidité de placement des moteurs. Malheureusement, nos mesures ne sont pas allées dans ce sens car la réduction du temps alloué à cette partition s'est révélée plutôt néfaste pour les performances générales du simulateur. Si on divisait par deux le temps alloué à la partition SCAO, son temps d'exécution global passait de 4,9 s à 7,9 s en moyenne. Or ce ralentissement ne profitait pas vraiment aux autres partitions puisque la

partition Leica, par exemple, voyait son temps d'exécution s'améliorait seulement de 2 s, passant de 9 s à 7 s en moyenne ! De plus, cette configuration amenait à plusieurs problèmes puisqu'un faible nombre d'actions étaient exécutées étant donné le temps important pris par l'ensemble des deux partitions pour placer les moteurs et traiter les actions.

Même en étant plus modérés dans la réduction du temps alloué à la partition SCAO c'est-à-dire en utilisant la valeur de 200 ms, les résultats n'étaient pas très concluants. La partition SCAO voyait son temps d'exécution augmentait de 0,8 s en moyenne (de 4,9 s à 5,7 s) alors que celui de la partition Leica ne diminuait que de 1 s en moyenne (de 9 s à 8 s). Il semble donc qu'il ne soit pas possible d'améliorer de manière visible les performances du simulateur en modifiant les temps alloués aux partitions.

Une explication concernant cette impossibilité est probablement liée à la communication. En effet, toutes les partitions utilisent au moins 1 port de communication pour interagir avec les autres. Et comme nous l'avions déjà remarqué lors de l'élaboration de la Master Copy, ce sont les lectures et écritures en sampling et queuing qui prennent du temps dans l'exécution du simulateur. Ainsi, si une amélioration sensible et durable des performances est recherchée, elle devra passer par une modification des méthodes de communications ou bien une amélioration de celles déjà existantes.

Enfin, le délai limite acceptable par le watchdog n'a pas été modifié depuis la Master Copy. En effet, même si le temps alloué aux partitions Master et Slave a diminué, le cycle général est inchangé et donc la valeur de 2 s est toujours appropriée afin de garantir un bon ratio entre les mauvaises détections et les détections lentes. La temporisation entre l'envoi de signaux au watchdog est aussi restée inchangée car l'envoi avait lieu un nombre suffisant de fois par rapport au délai limite fixé précédemment. Le recouvrement a par ailleurs été testé avec succès, mais avec un temps de réaction important (environ 10 s) dû une fois de plus à la lenteur des canaux de communication.

IX. Perspectives

IX. 1. Perspectives fonctionnelles

IX. 1. 1. Gestion des points de reprise

Comme nous ne pouvions pas nous permettre de perdre les rapports envoyés par la partition Leica et par la partition SCAO aux partitions exécutant les plans reçus - donc les partitions Master et Slave, nous les transmettons en utilisant un port queueing. De plus, afin de s'assurer qu'aucune perte de ces rapports n'a lieu lors du phénomène de recovery, nous les envoyons systématiquement au Slave et au Master.

Ces mesures de précautions n'auraient pas d'inconvénients si le satellite sur lequel ce programme serait éventuellement implémenté subit un dysfonctionnement de ses partitions Master et Slave très régulièrement et que dans un intervalle entre deux redémarrages le nombre d'images n'est pas trop grand. Dans un cas contraire, on assisterait à un engorgement du port queuing entre la partition en mode back-up et Leica ou SCAO, les messages n'étant parcourus, et donc vidés uniquement lors du recouvrement.

Une solution évidente serait donc de purger régulièrement ces ports queuing lorsque la partition est en mode backup. Le problème est qu'actuellement le seul moyen de connaître le contenu d'un port est de le vider. Effectuer une purge régulièrement nous obligerait donc à vider entièrement le port (comme nous ne

connaissons pas le nombre de messages total sur le port et qu'une fois qu'un message est enlevé du port nous ne pouvons pas l'y remettre) et nous prendrions donc le risque de perdre des rapports, ce qui n'est pas acceptable.

Une purge intelligente sera donc à prévoir.

IX. 1. 2. Contrôle des partitions

Lorsque nous nous sommes entretenus avec Marie-Hélène Deredempt, nous avons appris qu'actuellement, le redémarrage des partitions était ordonné au satellite par un opérateur au sol.

Dans notre programme, le Raspberry PI est plus autonome et certaines partitions (Leica et SCAO) ont la possibilité de « se suicider » si elles s'aperçoivent que leur fonctionnement est incorrect. Cette option pourrait être d'ailleurs donnée ultérieurement aux autres partitions du système en considérant attentivement les conditions d'extinction. Cependant, ces décisions prises en autonomie ne peuvent pas rester invisibles pour l'opérateur assurant la maintenance au sol (modélisé par l'étudiant sur l'ordinateur connecté au Raspberry). Il faudrait donc assurer une transmission de rapport de maintenance du Raspberry à l'ordinateur. De plus, pour laisser la possibilité à l'opérateur de contrôler le fonctionnement du Raspberry à distance, il serait intéressant de donner la possibilité de relancer les partitions à partir d'un ordre envoyé par SSH.

IX. 1. 3. Traitement des images

Du point de vue de l'exécution de la mission, notre programme exécute la prise des images, leur compression, leur stockage et leur envoi vers une station sol. Nous avons donc laissé de côté une fonctionnalité additionnelle : l'analyse des images à bord, qui nécessitait un travail d'analyse des images que la durée du projet long ne nous permettait pas d'aborder. Par extension il faudrait aussi mener une réflexion sur l'utilisation de ces résultats d'analyse : faudra-t-il quand même l'enregistrer ? Et si on l'enregistre, est-ce qu'on l'envoie par défaut ou est-ce qu'on la signale à l'opérateur, qui prendra par la suite la décision de la faire transmettre ?

IX. 1. 4. Implémentation du plan

Dans notre programme, nous utilisons des plans que nous générons sans aucune considération de faisabilité liée aux caractéristiques physiques du satellite simulé. En effet, le soin de générer un plan réellement faisable par un satellite a été laissé à un autre groupe de projet long. Malheureusement la durée du projet long ne nous a pas permis d'adapter notre programme à ce nouveau format de plan.

IX. 2. Perspectives non fonctionnelles

Dans le standard ARINC653, on peut construire des ports Queuing ou Sampling entre deux partitions. Ces ports peuvent être unidirectionnels ou bidirectionnels.

Actuellement pour communiquer entre deux partitions, des objets C++ existent dans le simulateur pour chaque type de port. Ces objets offrent des méthodes de lecture et d’écriture dans ces ports qui

demandent en argument selon les cas et en outre les numéros des ports et les numéros des sockets associées. Malheureusement pour l'instant, nous sommes contraints de n'utiliser que des ports bidirectionnels (deux ports unidirectionnels inverses). En effet, pour configurer les ports de communication entre les partitions, on écrit en dur dans un fichier texte pour chaque port voulu sa nature, sa partition émettrice, sa partition destinataire et son numéro. Lors de l'initialisation de chaque partition, le simulateur ne créé des sockets que pour les ports en réception de la partition et les associe grâce à la fonction *bind* au numéro de port correspondant. Or, dans les objets de communication, les méthodes d'écriture des objets de communication demandent pour l'instant en plus du numéro de port en écriture un descripteur de socket, qui n'a pas été créé étant donné que le port est en écriture…

La communication dans le simulateur se faisant en mode datagramme, c'est la fonction *sendto* qui est utilisée. Elle demande comme paramètres en outre un descripteur de socket et un pointeur sur une structure *sockaddr* qui contient notamment l'adresse et le port de destination. Finalement, la socket n'est pas réellement cruciale et n'a pas besoin d'être créée à l'avance pour l'écriture. On pourrait donc envisager tout simplement de créer la socket dynamiquement dans la méthode elle-même à chaque appel, ou bien, pour éviter tout « dynamisme » impropre aux systèmes temps réel, on pourrait créer une socket dédiée à l'écriture pour chaque partition à l'initialisation de celle-ci. Ainsi les méthodes d'écriture dans les ports de communication ne demanderaient plus de spécifier une socket en argument, et seul le numéro du port de destination serait réellement pertinent.

Notons qu'il serait encore plus avantageux de bénéficier de méthodes de communication ayant comme uniques paramètres le nom ou le numéro de la partition concernée, son type de port et le numéro du port en question (dans la liste des ports de ce type pour cette partition), ce qui permettrait de faire en partie abstraction du fichier texte de configuration de la communication dans le code source des partitions, qui pour l'instant en est directement et pleinement dépendantes.

X. Bilan

X. 1. Bilan technique

Notre projet long a été l'occasion de prendre connaissance de nouveaux concepts et nous a confrontés à des problèmes que nous avons dû résoudre grâce à de nouveaux outils logiciels.

Tout d'abord, nous avons eu l'occasion de travailler sur un Raspberry PI, nano ordinateur très personnalisable. Ceci nous a permis d'en apprendre plus sur l'architecture ARM et de mettre en relation le monde physique et le monde logiciel dans un environnement moderne.

De plus, nous avions eu pour la première fois le travail de nous insérer dans un projet déjà commencé, et lorsque nous nous approprions le code du simulateur ARIC 653 nous avons constaté le manque de lisibilité de certaines parties de ce dernier ; par exemple nous avons trouvé des accesseurs appelés comme des constructeurs, ce qui rendait difficile la relecture. Nous avons donc décidé de rendre le code plus compréhensible, ce qui nous a poussés à approfondir nos connaissances sur les bonnes pratiques de programmations en C++.

Nous avons été aussi confrontés à la situation où nous étions quatre à travailler en même temps sur le code, ce qui nous a amenés à modifier simultanément les mêmes fichiers. Ainsi, lors des mises en commun, nous risquions de perdre du temps à s'assurer qu'aucune modification n'ait été oubliée. C'est pour cela que nous avons décidé d'utiliser le gestionnaire de version Git. Ce logiciel permettait de stocker sur le serveur du LAAS nos différentes versions dans des branches et de fusionner ces dernières lorsque nous souhaitons faire des mises en commun. Cette dernière était facilitée par son côté automatique ; nous n'avions à agir que quand des modifications étaient en conflit.

Enfin, pour la première fois dans notre scolarité à l'ENSEEIHT, notre travail risquerait d'être utilisé ultérieurement par des étudiants de l'ENSEEIHT de troisième année, nous avons décidé de produire une documentation pour leur faciliter l'appropriation de notre travail. Devant l'importance de notre code nous avons choisi de nous orienter vers de la génération automatique de documentation à partir du code grâce au logiciel Doxygen. Ainsi, en formalisant nos commentaires, nous pouvions produire une documentation en HTML facilement navigable.

X. 2. Bilan personnel

Au-delà des considérations techniques nous avons pour la première fois été vraiment mis dans une position d'ingénieur. Nous avions à nous coordonner seuls en tant qu'équipe et à formaliser seuls nos problèmes. De plus il n'y avait pas vraiment de bonne solution ni de professeurs avec une correction idéale à nous proposer, ce qui nous a laissé une grande liberté dans la résolution de nos problèmes. De plus, une fois que nous avions accompli le cahier des charges minimal de notre projet long –adapter le simulateur à une maquette physique composée d'un RaspberryPI et d'une caméra montée sur une plateforme orientable et parvenir à commander la prise d'image et l'orientation de la caméra, nous étions relativement libres du choix des améliorations que nous apportons, ce qui nous a mis en situation d'évaluer notre capacité à mettre en place ces améliorations et le temps que cela risquerait de prendre.

Nous avons aussi eu l'occasion d'expérimenter et de comprendre l'intérêt de respecter les méthodes de génie logiciel. Nous nous sommes rendu compte que la méthode incrémentale avait été celle que nous avons suivie instinctivement, et que lorsque nous avons dévié de cette bonne pratique, les problèmes s'accumulaient.

Enfin, sur un plan plus personnel, nous avons tous trouvé que ce projet était le plus intéressant auquel nous avons participé durant nos trois années à l'ENSEEIHT, de par le sujet, sa longueur, et la liberté d'action qui nous a été permise.

Annexes

I. Références

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell and Carl Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, Vol. 1, No 1 January-March 2004.

Christopher B. Watkins and Randy Walter, Transitioning from Federated Avionics Architectures to Integrated Modular Avionics, 26th Digital Avionics Systems Conference (DASC) Dallas, Texas, October 2007.

ARINC 653 - An Avionics Standard for Safe, Partitioned Systems, Wind River Systems / IEEE Seminar, August 2008, Retrieved 2009-05-30.)

Frank Uyeda, Lecture 7: Memory Management, CSE 120 : Principles of Operating Systems UC San Diego, December 2013.

Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, Timo Latvala and Laura Nummila, Formal Development and Assessment of a Reconfigurable On-board Satellite System, TUCS Technical Reports 1038, Turku Centre for Computer Science, 2012

EXIF Tag Parsing Library : http://libexif.sourceforge.net/

Pi-blaster : https://github.com/mvitousek/pi-blaster

ServoBlaster : https://github.com/richardghirst/PiBits

II. Documentation Doxygen