Documentation de OMEGAAA

Ordinateur de **M**esure, d'**E**valuation, de **G**estion
et d'**A**nalyse pour l'**A**éronautique et l'**A**érospatiale

FIGURE 0.1 – Logo du projet OMEGAAA

Alexis Paillard

26 octobre 2024

Table des matières

Table des figures

Liste des codes

Mise en contexte

Ce document a pour but de présenter l'OMEGAAA, un concepte d'ordinateur de bord pour une fusée amateur. Le projet a été mené par Alexis Paillard et se base en majeur partie sur le projet Unknown réalisé dans le cadre de l'association AeroIPSA.

- AeroIPSA

AéroIPSA est une association étudiante de l'école d'ingénieurs IPSA qui conçoit et réalise entièrement des projets fonctionnels en rapport avec le secteur aérospatial. Elle rassemble des étudiants autour de projets d'astromodélisme, principalement de lanceurs, mais aussi de Cansats (micro-satellites atmosphérique). Cela permet aux différents membres de l'association d'appliquer les notions apprises durant leur cursus au profit d'un projet d'envergure et d'acquérir les compétences nécessaires dans leur futur métier d'ingénieur.

- Unknown

Unknown est un projet de module électronique de fusée expérimentale réalisé par Vincent Fauquembergue et Alexis Paillard. Les expériences du projet Unknown ont été les suivantes :

- **Expérience principale** : Relocaliser une fusée après son lancement grâce à un module de télémesure LoRa renvoyant les données GNSS tout au long du vol.
- **Expérience secondaire** : Réalisation d'une collecte de données provenant de nombreux capteurs, barométrique et centrale inertielle, afin de reconstituer le vol après récupération des données stockées sur une mémoire flash.

L'un des objectifs du projet Unknown était de réaliser un module électronique se voyant plus facilement intégrable dans une fusée amateur. Cela s'est traduit par l'utilisation d'une seule carte électronique ayant tous ses composants directement soudés dessus ainsi que l'utilisation d'un microcontrôleur autre que l'Arduino ou que la Teensy. Le choix fait s'est porté sur un STM32F4 de STMicroelectronics. La programmation de ce microcontrôleur a été réalisée en C et un bon nombre des drivers nécessaires ont dû être réadapté par les membres du projet ce qui a permis d'acquérir de nouvelles compétences en programmation bas niveau.

L'annexe A présente le rapport du projet Unknown ainsi que des photos du module.

- OMEGAAA

OMEGAAA est l'acronyme de **O**rdinateur de **M**esure, d'**E**valuation, de **G**estion et d'**A**nalyse pour l'**A**éronautique et l'**A**érospatiale. Il s'agit d'un concepte d'ordinateur de bord qui a pour but de gérer les différentes phases de vol d'une fusée amateur, tout en effectuant des mesures notamment d'accélérations et de vitesses de rotation (à l'aide d'un IMU). Ces données sont ensuite traitées pour en extraire la position et l'attitude de la fusée et enregistrées dans une mémoire. L'OMEGAA gère également une communication par télémétrie avec une station au sol.

Ce module fait suite au projet Unknown en reprenant les composants électroniques de ce dernier. Il a pour but de continuer à développer les compétences en programmation acquises lors du projet Unknown et d'en développer de nouvelle comme la création du RTOS [1]. Cet ordinateur de bord étant fictif, il n'a pas été réalisé mais l'ensemble des travaux et des recherches effectuées pour sa conception sont présentées dans ce document et ont pour but de servir de support pour la réalisation d'un tel module électronique.

1. Un RTOS (**R**eal **T**ime **O**perating **S**ystem) est un système d'exploitation conçu pour gérer les ressources matérielles de manière à garantir que des tâches critiques soient exécutées parallèlement à d'autre et dans des délais stricts et prévisibles. Ce type de système est laregement employé dans des application tels que les systèmes embarqués, les dispositifs médicaux, les systèmes de contrôle industriel, et les applications aéronautiques et aérospatiales...

1 Objectifs

L'objectif de l'OMEGAAA est d'avoir un ordinateur de bord relativement puissant mais également polyvalant. Afin de démontrer les capacités de l'OMEGAAA, les objectifs suivants ont été fixés. L'OMEGAAA doit être capable :

- D'acquérir des données de divers capteurs.
- D'effectuer des calculs sur les données acquises.
- De stocker bons nombres de données.
- De communiquer avec l'extérieur par radio.
- D'aborder différents comportements en fonction de l'état de la fusée.
- D'effectuer l'ensemble de ces tâches en optimisant les temps de processuces.

Le logigramme suivant présente les 5 étapes principales qu'une fusée peut avoir. L'algorithme générale du programme de l'OMEGAAA est alors présenté. A noter que le schéma ne fait qu'exposer brievement la logique du programme et n'est pas exhaustif.

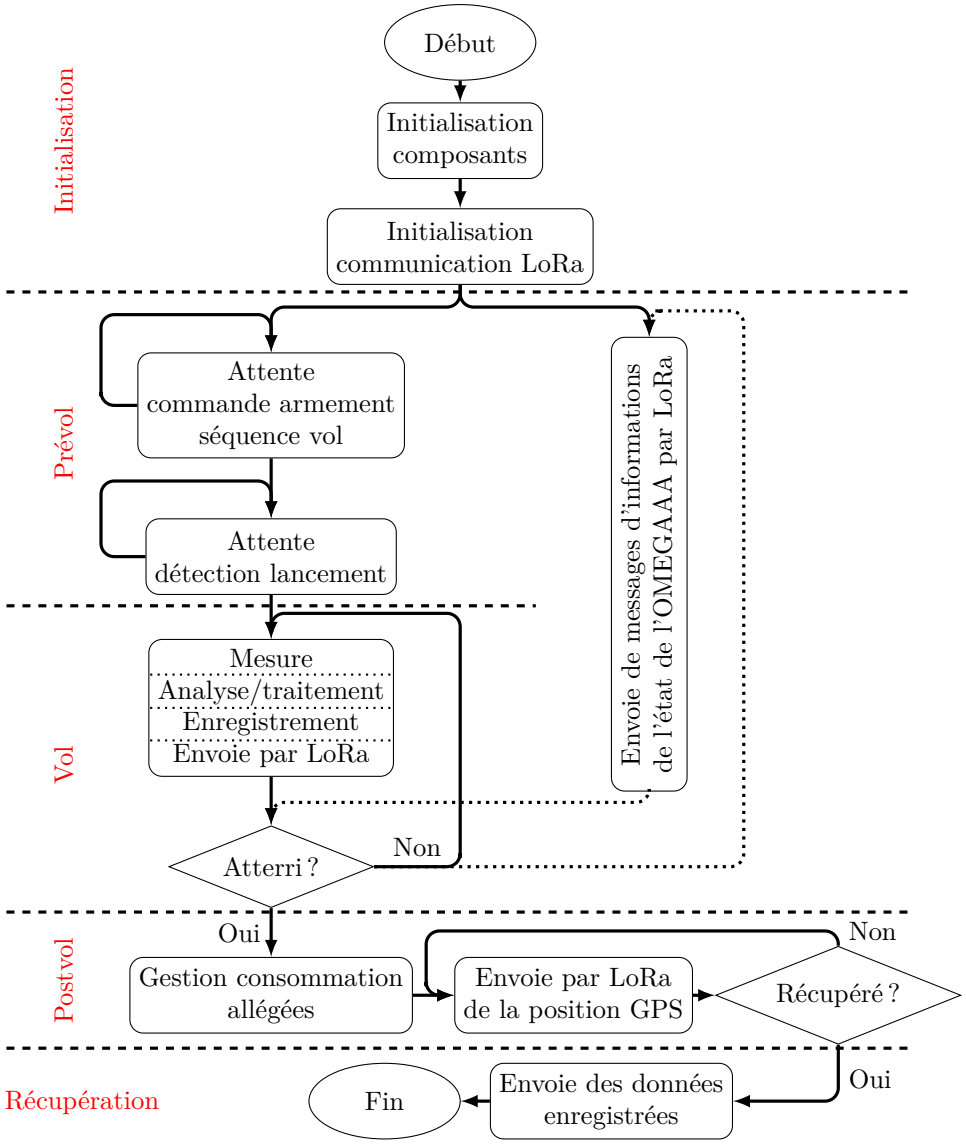


FIGURE 1.1 – Logigramme du programme de l'OMEGAAA

2 Projet STM

Le programme de l'OMEGAAA est écrit à l'aide de l'environnement de développement STMicroelectronics. Ce dernier permet de créer des projets pour les microcontrôleurs de la firme. Chaque projets est composé de plusieurs sous-dossiers et fichiers. Le HAL [2] fournit par STMicroelectronics étant écrit en C, et que bons nombres d'information, tutoriels et exemples sont disponibles en C, le programme de l'OMEGAAA utilisera ce language.

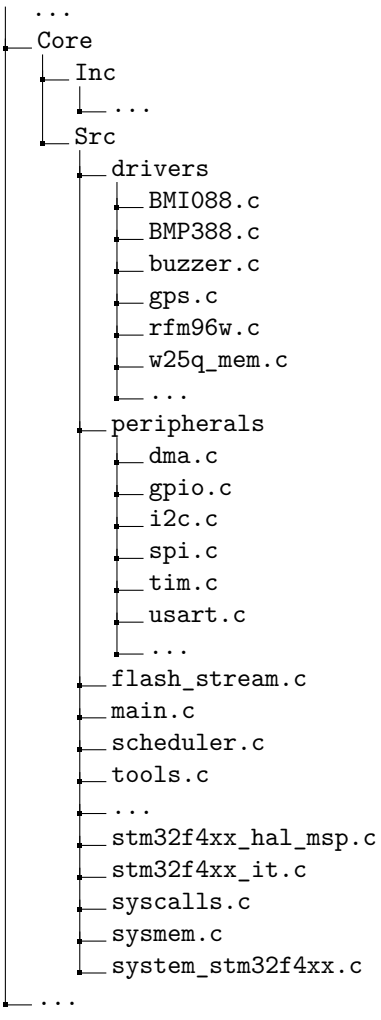


FIGURE 2.1 – Arboressance du programme

L'intégralité du programme de l'OMEGAAA se trouve dans le dossier `"Core"`. Ce dernier est composé des fichiers de déclarations (.h) dans le dossier `Inc` et des fichiers de programme (.c) dans le dossier `Src`. Tous les programmes relatifs aux composants externes au microcontrôleur (capteurs, ...) se trouve dans le dossier `drivers` et ceux relatifs à la gestion des composants internes (gestionnaires des périphériques, ...) dans `peripherals`.

On retrouve dans le dossier `Inc` symétriquement la même arboressance de dossiers et fichiers que dans le dossier `Src` à l'extention près (non des `".c"` mais des `".h"`).

Le point d'entrée du programme se trouve dans la fonction `main` du fichier du même nom.

Le fichier `scheduler.c` contient l'implémentation de l'ordonanceur et la définition d'une tâche (cf. 3.4).

Le fichier `flash_stream.c` contient les structures et fonctions permettant de lire et d'écrire des données dans une mémoire tout en gérant les pointeurs de lecture et d'écriture.

Le fichier `tools.c` contient des fonctions utilisées à différents endroits du programme.

Enfin, les fichiers `stm32f4xx_hal_msp.c`, `stm32f4xx_it.c`, `syscalls.c`, `sysmem.c` et `system_stm32f4xx.c` sont des fichiers de configuration, de gestion des interruptions, de gestion de la mémoire et de gestion du système général du microcontrôleur. Ces fichiers sont générés automatiquement par l'environnement de développement et bien qu'ils le peuvent, ils ne seront pas modifiés par l'utilisateur.

2. HAL (pour **H**ardware **A**bstraction **L**ayer) est une bibliothèque logicielle qui fournit une interface de programmation pour les périphériques matériels.

3 Programmation asynchrone

L'un des objectifs principales du logiciel est de garantir une fréquence de mesure des capteurs la plus élevée possible. Pour cela, le programme doit être capable d'effectuer plusieurs tâches en parallèle. En effet par exemple, l'envoi d'un message de télémesure par LoRa prend un certain temps durant lequel le programme ne doit pas être bloqué. Pour cela, les différentes tâches seront exécutées de manière asynchorne [3] et non nécessairement séquentiellement.

Dans le cas de l'OMEGAAA, l'architecture asynchrone se caractérise par deux objets :
— Les tâches : ce sont les différentes actions que le programme peut effectuer en parallèle.
— L'ordonanceur : c'est une structure contenant notamment les tâches à exécuter.

3.1 Une tâche (TASK)

Contrairement à une fonction classique, une tâche est une fonction qui peut commencer, interrompre et reprendre son exécution en plusieurs temps. Dans l'OMEGAAA, une tâche se matérialise par une structure de données contenant trois éléments : une fonction classique (`func` [4]), un indice de référence(`idx` [4]) et une sous-structure de données appelée "contexte d'exécution" (`context` [4]). Le contexte d'exécution contient les informations nécessaires pour reprendre l'exécution de la tâche là où elle s'était arrêtée auparavant.

Par simplicité et pour des raisons de performances, bien que chaque type de tâche a besoin d'un contexte d'exécution unique, `context` est une zone mémoire de taille fixe. Cette taille est définie par la constante `ASYNC_CONTEXT_DEFAULT_BYTES_SIZE` [4] et est par défaut de 64 octets. Cela permet d'uniformiser la structure de toutes les tâches. Cette taille doit être suffisante pour contenir le contexte d'exécution de n'importe quelle tâche et doit être redéfini en fonction des besoin du programme tout en considérant les limites de mémoire du microcontrôleur.

3.2 L'ordonanceur (SCHEDULER)

L'ordonanceur est une structure de données qui contient les tâches à exécuter. Il se matérialise par une structure contenant quatre éléments : le nombre de tâches en cours d'exécution (`length` [4]), l'indice de la tâche actuellement en cours d'exécution (`current_idx` [4]), le nombre de tâches exécutées depuis le dernier tour de boucle (`nbr_has_run_task` [4]), un tableau de pointeurs de tâches (`tasks` [4]) et un tableau de booléens indiquant quels sont les pointeurs du tableau `tasks` faisant référence à des tâches en cours d'exécution (`running` [4]). L'ordonanceur gère également la zone mémoire allouée dans le tas pour les tâches. Il ne peut gérer qu'un nombre limité de tâches définie par la constante `MAX_TASKS_NBR` [4] et vaut par défaut 50. Là aussi, si plus (ou moins) de tâches seront exéctutées en parallèle, il est nécessaire de redéfinir cette constante en conséquence.

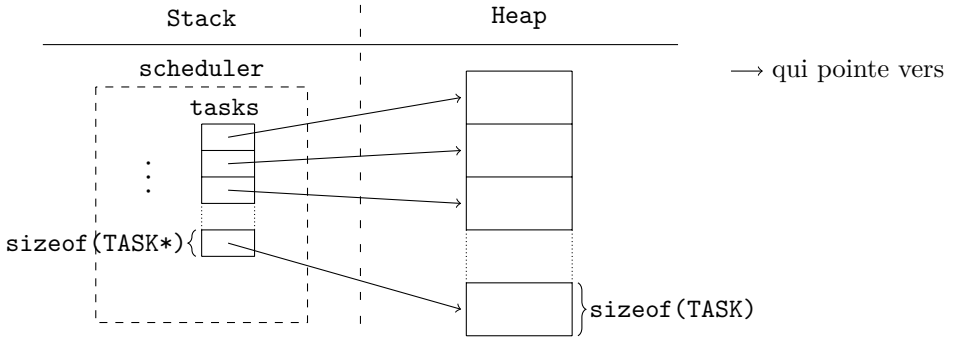


Figure 3.1 – L'agencement des tâches dans le tas et leur lien avec l'ordonanceur

3. La programmation asynchrone est un modèle de programmation qui donne l'illusion qu'un ordinateur effectue plusieurs tâches en parallèle, sans attendre la fin de chaque tâche avant de passer à la suivante.
4. cf Code 3.1

3.3 Exemple 1 : Ordonancement des tâches

Pour mieux comprendre le fonctionnement de l'ordonanceur et des tâches, un exemple de scénario d'utilisation est donné ci-dessous.

- En premier lieu, les tâches A, B, C, D et E sont ajoutées dans une zone mémoire alloué au moment de l'initilisation de l'ordonanceur et leur addresse mémoire respective sont placées dans le tableau `task`. Puisqu'elles sont toutes en cours d'exécution, les booléens correspondants à leur index dans le tableau `running` sont à `true`. (L'addresse mémoire de chaque tâche sera noté "&X" où X est le nom de la tâche.)

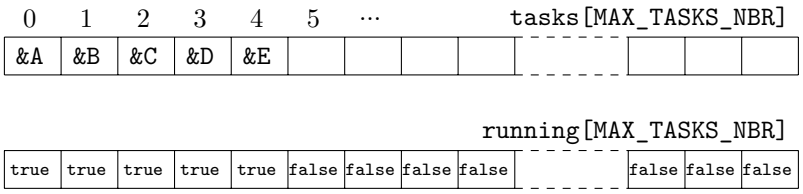


FIGURE 3.2 – Schéma des tableaux tasks et running de l'ordonanceur (1)

- Ensuite, la tâche C se termine et se retire de l'ordonanceur. Son booléen correspondant dans le tableau `running` est alors à `false`. Prenons cet exemple pour expliquer comment les tâches sont exécutées par l'ordonanceur. A chaque fois que la fonction `run_scheduler`[4] est appelée, l'ordonanceur exécute la tâche suivante. Cette dernière correspond à la première tâche dont le booléen correspondant dans le tableau `running` est à `true` depuis l'index `current_idx` jusqu'à ce que l'ensemble des tâches aient été exécutées. Dans le cas de la figure 3.3, au premier tour de boucle, la tâche A est exécutée, puis la tâche B, puis la tâche D et enfin la tâche E. La tâche C n'est pas exécutée car son booléen correspondant dans le tableau `running` est à `false`. `current_idx` passe alors respectivement de 0, 1, 3, et 4. Ensuite, puisque toutes les tâches ont été parcourues une fois, la tâche suivante est la tâche A. Cela est répété jusqu'à ce que toutes les tâches soient retirées de l'ordonanceur.

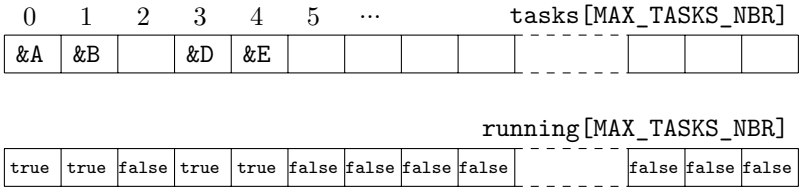


FIGURE 3.3 – Schéma des tableaux tasks et running de l'ordonanceur (2)

- Depuis l'exemple ci-haut, les tâches F et G sont ajoutées à l'ordonanceur. L'ordonanceur cherche alors la première place libre dans le tableau `tasks` pour y ajouter la tâche, soit le premier index où le booléen correspondant dans le tableau `running` est à `false`. Dans ce cas, c'est l'index 3. La tâche F est alors ajoutée à cet index et le booléen correspondant dans le tableau `running` est mis à `true`. La tâche G est ajoutée de la même manière à l'index 5.

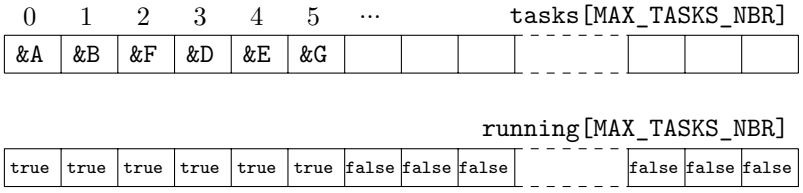


FIGURE 3.4 – Schéma des tableaux tasks et running de l'ordonanceur (3)

3.4 Implémentation

Ci-dessous est présenté le code source de la déclaration des fonctions principales associées à l'ordonanceur et aux tâches. L'implémentation même de ces fonctions n'est pas présentée ici puisque jugé trop longue et non essentielle à la compréhension du concept. L'implémentation complète est disponible dans le code source du projet.

Code 3.1 – Définition des tâches et de l'ordonanceur (scheduler.h)

```h
#include <stdbool.h>
#include "stm32f4xx_hal.h"

#define ASYNC_CONTEXT_DEFAULT_BYTES_SIZE 64
#define MAX_TASKS_NBR 50

// fonction a executer (prend en param le scheduler et le contexte)
typedef void(*__task_func_t)(void*, void*);

typedef struct {
    __task_func_t    func;
    size_t           idx;
    bool            *is_done;
    uint8_t          context[ASYNC_CONTEXT_DEFAULT_BYTES_SIZE];
} TASK;

typedef struct {
    size_t    length;
    size_t    current_idx;
    size_t    nbr_has_run_task;
    TASK     *tasks[MAX_TASKS_NBR];
    bool      running[MAX_TASKS_NBR];
} SCHEDULER;

typedef void(*task_func_t)(SCHEDULER*, TASK*);


// Initialise l'ordonanceur notamment en allouant la zone mémoire pour les tâches.
void init_scheduler(SCHEDULER *scheduler);

// Initialise une tâche, la place dans la zone mémoire allouée, l'ajoute à la liste
// des tâches, met à jour le tableau running (active la tâche), les variables de
// l'ordonanceur et renvoie l'addresse mémoire de la tâche.
TASK *add_task(SCHEDULER *scheduler, task_func_t func);

// Met à jour le tableau running (désactive la tâche), les variables de l'ordonanceur
// et le drapeau "*is_done" de la tâche s'il est défini.
void kill_task(SCHEDULER *scheduler, TASK *task);

// Exécute une tâche
void run_task(SCHEDULER *scheduler, TASK *task);

// Exécute la tâche suivante
void run_scheduler(SCHEDULER *scheduler);
```

3.5 Exemple 2 : Création et logique d'une tâches

Afin de mieux comprendre le fonctionnement de l'ordonanceur et des tâches, un exemple d'implémentation est donné ci-dessous. Cet exemple est composé de deux tâches :

— `task1` : une tâche qui créer la tâche 2 et affiche un message d'état du programme.
— `task2` : une tâche simulant une opération prenant du temps.

On suppose que la fonction `HAL_GetTick` retourne le temps écoulé en millisecondes depuis le démarrage du programme.

Code 3.2 – Exemple asynchrone

```c
#include <stdbool.h>
#include "stm32f4xx_hal.h"
#include "scheduler.h"

typedef enum {
    TASK1_INIT,
    TASK1_WAIT,
    TASK1_END
} TASK1_STATE;

typedef struct {
    TASK1_STATE state;
    bool        task2_done;
    uint32_t    last_time;
} ASYNC_task1_CONTEXT;

typedef struct {
    bool         started;
    uint32_t     delay;
    uint32_t     last_time;
} ASYNC_task2_CONTEXT;


void ASYNC_task1_init(TASK *task);
void ASYNC_task2_init(TASK *task, uint32_t delay);
void ASYNC_task1(SCHEDULER *scheduler, TASK *task);
void ASYNC_task2(SCHEDULER *scheduler, TASK *task);


// ========================================


void ASYNC_task1_init(TASK *task) {
    ASYNC_task1_CONTEXT *context = (ASYNC_task1_CONTEXT*)task->context;

    context->state = TASK1_STATE_INIT;
    context->task2_done = false;
    context->last_time = 0;
}

void ASYNC_task2_init(TASK *task, uint32_t delay) {
    ASYNC_task2_CONTEXT *context = (ASYNC_task2_CONTEXT*)task->context;

    context->started = false;
    context->delay = delay;
    context->last_time = 0;
}

```

```
49
50  void ASYNC_task1(SCHEDULER *scheduler, TASK *task) {
51      ASYNC_task1_CONTEXT *context = (ASYNC_task1_CONTEXT*)task->context;
52
53      switch (context->state) {
54          case TASK1_INIT:
55              printf("Task 1 : Start\n");
56              TASK *task = add_task(scheduler, ASYNC_task2);
57              ASYNC_task2_init((ASYNC_task2_CONTEXT*)(task->context),
58                               &context->task2_done, 5000);
59              task->is_done = &(context->task2_done);      // Partage drapeau
60              context->state = TASK1_WAIT;
61              printf("Task 1 : Wait ");
62              context->last_time = HAL_GetTick();          // Initialise timer
63              break;
64          case TASK1_WAIT:
65              if (HAL_GetTick() - context->last_time > 200) {
66                  context->last_time = HAL_GetTick();      // Réinitialise timer
67                  printf(".");
68              }
69              if (context->task2_done) {                   // Vérifie tâche 2 terminée
70                  context->state = TASK1_END;
71              }
72              break;
73          case TASK1_END:
74              printf("\nTask 1 : End\n");
75              kill_task(scheduler, task);
76              break;
77      }
78  }
79
80  void ASYNC_task2(SCHEDULER *scheduler, TASK *task) {
81      ASYNC_task2_CONTEXT *context = (ASYNC_task2_CONTEXT*)task->context;
82
83      if (!context->started) {
84          context->started = true;
85          context->last_time = HAL_GetTick();              // Initialise timer
86      }
87      if (HAL_GetTick() - context->last_time > context->delay) {
88          kill_task(scheduler, task);                      // Drapeau *is_done mis à jour
89      }
90  }
91
92  // ======================================================
93
94  int main() {
95      SCHEDULER scheduler;
96      init_scheduler(&scheduler);
97
98      TASK *task = add_task(&scheduler, ASYNC_task1);
99      ASYNC_task1_init((ASYNC_task1_CONTEXT*)(task->context));
100
101     while (scheduler.length > 0) {
102         run_scheduler(&scheduler);
103     }
104     return 0;
105 }
```

Cela produit le résultat suivant :

Code 3.3 – Résultat xemple asynchrone

```
Task 1 : Start
Task 1 : Wait ........................
Task 1 : End
```

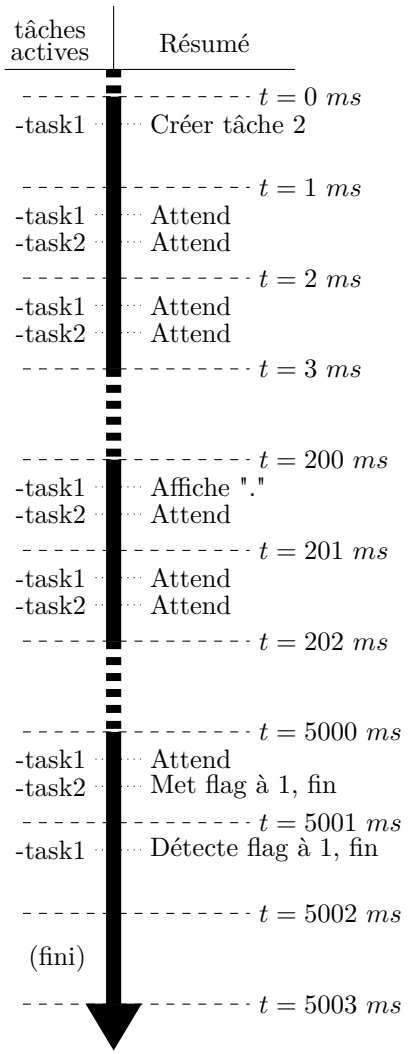


FIGURE 3.5 – Chronologie de l'exemple de programmation asynchrone

La figure ci-contre montre la chronologie de l'exemple. A noter que le temps d'exécution est donné à titre indicatif et ne correspond pas à une échelle de temps réelle.

Le programme commence par créer et initialiser l'ordonanceur (ligne 95 -96). C'est à dire que de la mémoire est alloué pour les tâches, que les éléments du tableau `running` sont tous mis à `false` et que les autres variables de l'ordonanceur sont également initialisées. Ensuite, la tâche 1 est créée et ajoutée à l'ordonanceur. Cela est réalisé par l'appelle de la fonction `add_task`. Cette fonction cherche la première place libre dans la zone mémoire allouée à cet effet dans le tas, initialise une nouvelle tâche à cet endroit et renvoie un pointeur vers cette dernière. Le context de la tâche est ensuite initialisé par sa fonction relative (ici `ASYNC_task1_init`) (lignes 98-99). Enfin, le programme entre dans une boucle et exécute la fonction `run_scheduler` tant que des tâches se trouvent active dans l'ordonanceur.
Au début de l'exécution, seule la tâche 1 se trouve active, elle est alors exécutée. Ligne 51, le contexte d'exécution de la tâche 1 est récupéré et l'état de la tâche est vérifié. La plupart des tâches sont des machines à états finis [5]. Dans le cas de la tâche 1, elle se trouve au début dans l'état `TASK1_INIT`, elle affiche donc le message "Task 1 : Start", créer la tâche 2 en l'ajoutant à l'ordonanceur, affiche "Task 1 : Wait" et passe à l'état `TASK1_WAIT`.
La tâche 1 est alors mise en pause et la tâche 2 est exécutée. Cette dernière est une tâche simulant une opération prenant du temps. Elle démarre tout d'abord un timer de 5000 ms et vérifie à chaque nouvelle exécution si ce temps s'est écoulé. Lorsque c'est le cas, la tâche 2 se retire de l'ordonanceur. Quand la tâche 1 détecte la fin d'exéctution de la tâche 2 (via la mise à jour du drapeau "`task2_done`"), elle affiche "Task 1 : End" et se retire également de l'ordonanceur. Dans le cas contraire, la tâche 1 continue d'afficher des points toutes les 200 ms grâce à un timer propre. Ainsi, 25 points sont affichés correspondant à un point toutes les 200 ms pendant 5 secondes.

5. Machine à état : Une machine à état (ou automate fini) est un modèle de calcul utilisé pour représenter des systèmes avec un nombre fini d'états. Ce modèle est utilisé dans divers domaines comme les protocoles de communication, les jeux vidéo, les systèmes embarqués, etc.

4 Composants externes

Cette section présente les divers composants électroniques utilisés dans le projet OMEGAAA ainsi que leur driver associé.

4.1 Vue d'ensemble

L'OMEGAAA possède plusieurs composants que l'ont peut regrouper en quatre catégories en plus du microcontroleur : les capteurs, les composants de stockage, les témoins et les composants de communication.

Le microcontroleur est le cerveau de l'OMEGAAA. Il possède le programme informatique, les moyens pour l'exécuter et gérer les autres composants électroniques. l'un des concepts clé utilisé dans le microcontroleur est sa capacité à effectuer des échanges d'informations entre les composants par DMA [6]. Le projet OMEGAAA se base sur l'utilisation d'un STM32F411CEx.

Les différents capteurs sont :
— Un IMU (Inertial Measurement Unit) qui mesure l'accélération de translation et la vitesse de rotation de la fusée (ref : BMI088).
— Un GPS qui mesure la position de la fusée et sa vitesse (ref : MAX-M10Q).
— Un baromètre qui mesure l'altitude de la fusée et un thermomètre. (ref : BMP388).

Un seul composant de stockage est utilisé : une mémoire flash de 16 Mo (ref : W25Q128JVPQ).

Les témoins sont des composants qui permettent de savoir dans quel état se trouve l'OMEGAAA. Ils peuvent être sonores ou lumineux. Dans la version actuelle de l'OMEGAAA, un buzzer passif est utilisé permettant la création de mélodie.

Un seul composant de communication est utilisé, un module de télémétrie par protocole et modulation LoRa (ref : RFM98W). A noté que le microcontroleur possède aussi des moyens de communocation interfaçable avec un ordinateur (via USB).

Un cinquième type de composant pourrait être ajouté : les actionneurs. Ils permetteraient d'agire physiquement sur des éléments de la fusée. Par exemple, un servomoteur contrôlant une surface portante.

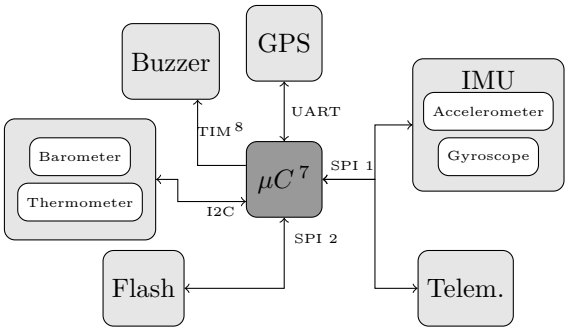


FIGURE 4.1 – Schéma des composants de l'OMEGAAA

La figure 4.1 montre un schéma des composants de l'OMEGAAA et leurs interconnexions. Les flèches indiquent les protocoles de communication utilisés entre les composants.

6. DMA : **D**irect **M**emory **A**cces. Avec la communication par DMA, le microcontrôleur peut déléguer la tâche de transfert de données à un contrôleur DMA dédié. Le contrôleur DMA est capable de transférer les données directement entre les périphériques (par exemple, entre un capteur) et la mémoire de l'ordinateur sans passer par le CPU. Cela libère le temps CPU pour qu'il puisse effectuer d'autres tâches.

8. μC : **M**icro**C**ontroleur.

8. TIM : **T**imer **I**nterface **M**odule sert dans ce contexte à générer des signaux PWM.

4.2 Communication entre composants

La programmation des protocoles de communications entre les différents composants électroniques dans un système temps réel se révèle être un éléments clé pour le bon fonctionnement de ce dernier mais aussi ardu à mettre en place. Il n'est par exemple pas souhaitable de bloquer certaines tâches du système lors de la transmission ou de la réception de données entre les composants. C'est notamment pour cela que le choix d'un microcontrôleur contre un microprocesseur a été fait. En effet, un microcontrôleur est capable de gérer des interruptions matérielles bien plus facilement et efficacement qu'un microprocesseur seul grâce aux périphériques intégrés de ce premier. Néanmoins une vigilent plus importante comparé à un système séquentiel est nécessaire. En effet, il est possible de se retrouver avec des problèmes de synchronisation entre les différents composants électroniques. Il faut ainsi être capable de ne pas permettre à deux tâches d'intéragir avec des composants utilisant le même bus de communication en même temps.

4.2.1 Présentation du concepte

Prenons l'exemple d'une communication SPI. Il est possible de partager un bus SPI entre plusieurs composants électroniques. Cependant, il est nécessaire de s'assurer que le bus SPI soit bien libre avant de pouvoir transmettre ou recevoir des données. Voici l'implémentation de quelques outils permettant de gérer la communication SPI de manière asynchrone.

Code 4.1 – Définition des outlis spi suplémentaires (tools.h)

```c
#include <string.h>

typedef struct SPI_HandleTypeDef_flag {
    SPI_HandleTypeDef *hspi;
    bool               is_used;
} SPI_HandleTypeDef_flag;

typedef struct ASYNC_SPI_TxRx_DMA_CONTEXT {
    SPI_HandleTypeDef_flag *hspi_flag;
    GPIO_TypeDef           *csPinBank;
    uint16_t                csPin;
    bool                    has_started;
    uint8_t                *tx_buf;
    uint8_t                *rx_buf;
    uint16_t                tx_size;
    uint16_t                rx_size;
} ASYNC_SPI_TxRx_DMA_CONTEXT;

// Initialise la structure SPI_HandleTypeDef_flag
void SPI_HandleTypeDef_flag_init(SPI_HandleTypeDef_flag *hspi_flag,
                                 SPI_HandleTypeDef* hspi);

// Initialise le contexte de la tâche ASYNC_SPI_TxRx_DMA
void ASYNC_SPI_TxRx_DMA_init(...);

// Tâche permettant de transmettre et recevoir des données en SPI en asynchrone
void ASYNC_SPI_TxRx_DMA(SCHEDULER* scheduler, TASK* self);
```

Code 4.2 – Implémentation des outlis spi suplémentaires (tools.c)

```c
void SPI_HandleTypeDef_flag_init(SPI_HandleTypeDef_flag *hspi_flag,
                                 SPI_HandleTypeDef      *hspi) {
    hspi_flag->hspi = hspi;
    hspi_flag->is_used = false;
}

void ASYNC_SPI_TxRx_DMA_init(TASK                       *task,
                             SPI_HandleTypeDef_flag     *hspi_flag,
                             GPIO_TypeDef               *csPinBank,
                             uint16_t                    csPin,
                             uint8_t                    *tx_buf,
                             uint8_t                    *rx_buf,
                             uint16_t                    tx_size,
                             uint16_t                    rx_size) {
    ASYNC_SPI_TxRx_DMA_CONTEXT *context=(ASYNC_SPI_TxRx_DMA_CONTEXT*)task->context;

    context->hspi_flag = hspi_flag;
    context->csPinBank = csPinBank;
    context->csPin = csPin;

    context->owner_task = owner_task;

    context->has_started = false;

    context->tx_buf = tx_buf;
    context->rx_buf = rx_buf;
    context->tx_size = tx_size;
    context->rx_size = rx_size;

    uint16_t size = tx_size + rx_size;

    context->__tx_buf_full = (uint8_t*)malloc(sizeof(uint8_t) * size);
    context->__rx_buf_full = (uint8_t*)malloc(sizeof(uint8_t) * size);

    memcpy(context->__tx_buf_full,
           context->tx_buf,
           context->tx_size);
}

void ASYNC_SPI_TxRx_DMA(SCHEDULER *scheduler, TASK *self) {
    ASYNC_SPI_TxRx_DMA_CONTEXT* context=(ASYNC_SPI_TxRx_DMA_CONTEXT*)self->context;

    uint16_t size = context->tx_size + context->rx_size;

    if (context->hspi_flag->hspi->State == HAL_SPI_STATE_READY) {
        if ((!context->has_started) && (!context->hspi_flag->is_used)) {
            context->has_started = true;
            context->hspi_flag->is_used = true;      // Prend le controle du bus SPI
            HAL_GPIO_WritePin(context->csPinBank, context->csPin, GPIO_PIN_RESET);
            HAL_SPI_TransmitReceive_DMA(context->hspi_flag->hspi,
                                        context->__tx_buf_full,
                                        context->__rx_buf_full,
                                        size);
        } else if ((context->has_started) && (context->hspi_flag->is_used)) {
            HAL_GPIO_WritePin(context->csPinBank, context->csPin, GPIO_PIN_SET);
            context->hspi_flag->is_used = false;     // Realease SPI bus
            if (context->rx_buf) {                   // rx_buf peut être NULL
                memcpy(context->rx_buf,
```

```
59                          context->__rx_buf_full + context->tx_size,
60                          context->rx_size);
61              }
62              free(context->__tx_buf_full);
63              free(context->__rx_buf_full);
64              kill_task(scheduler, self);
65          }
66      }
67  }
68
```

Bien que la couche d'abstraction matérielle (HAL) du microcontrôleur STM32 propose le type de structure `SPI_HandleTypeDef` pour gérer les bus SPI, il a été jugé plus pertinent de créer une structure supplémentaire `SPI_HandleTypeDef_flag` notamment pour gérer l'état du bus SPI. En effet, bien qu'un `SPI_HandleTypeDef` ait un attribut `State` permettant de connaître l'état du bus SPI et donc de savoir s'il est prêt à être utilisé, il est possible de se retrouver dans une situation où une tâche B souhaite utiliser le bus SPI alors qu'une tâche A n'ait pas fini de cloturer complètement son utilisation du bus SPI, en changeant notamment l'état du pin `CS` (Chip Select).

Aussi, la fonction `HAL_SPI_TransmitReceive_DMA`, servant à démarrer une transmission et réception de données en SPI en asynchrone, a un comportement qui permet d'envoyer et de recevoir des données en même temps. Cependant, le comportement souhaité dans la majeur partie des composants électroniques est de transmettre puis de recevoir des données. Il a été donc été décidé d'ajouter des buffers supplémentaires `__tx_buf_full` et `__rx_buf_full` afin de simuler un comportement plus attendu de la fonction.

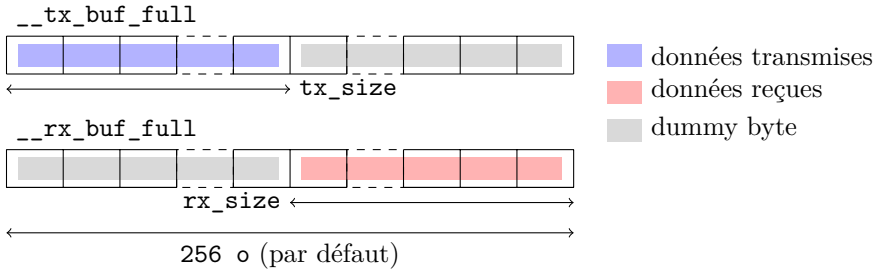


FIGURE 4.2 – Représentation des buffers `__tx_buf_full` et `__rx_buf_full`

Pour résumer la routine d'une communication SPI en asynchrone : une fois que la tâche est ajoutée au `scheduler` et que son context est initialisé, elle attend que le bus SPI soit prêt à être utilisé. Une fois que le périphérique SPI est prêt, trois cas de figure peuvent se présenter :

- **La tâche n'a pas encore démarrer une communication SPI et le bus est libre.**
  La tâche peut donc prendre le contrôle du bus SPI et initier une communication SPI en activant le pin `CS` donné en paramètre et en appelant la fonction `HAL_SPI_TransmitReceive_DMA`.
- **La tâche n'a pas encore démarrer une communication SPI mais le bus n'est pas libéré.**
  Cela signifie qu'une autre tâche avait déjà entamé une communication SPI et cette dernière n'a pas cloturer complètement son utilisation du bus SPI et un pin `CS` est actif. La tâche doit donc attendre que le bus SPI soit libéré.
- **La tâche a déjà démarré une communication SPI et le bus SPI est occupé.**
  Dans ce cas, la tâche qui avait déjà entamé une communication SPI est la tâche en cours. Elle peut donc cloturer sa communication SPI en désactivant le pin `CS`, en copiant les données reçues du buffer `__rx_buf_full` vers le buffer `rx_buf` (sans prendre en compte les dummy bit) et en libérant le bus SPI. La tâche peut alors se terminer.

Avec ce fonctionnement, tous processuces peut appeler plusieurs tâches `ASYNC_SPI_TxRx_DMA` simultanément et les requêtes seront traitées séquentiellement sans qu'il n'y ait de problème de synchronisation et sans bloquer le système. Une tâche voulant utiliser le bus SPI sera la plupart du temps construite comme une machine à 3 états dont les états sont définies par l'`enum ASYNC_DMA_STATE` :

- `ASYNC_DMA_START` : la tâche n'a pas encore démarré une communication SPI et appelle la sous-tâche `ASYNC_SPI_TxRx_DMA`.
- `ASYNC_DMA_WAIT` : la tâche a démarré une communication SPI et attend que la sous-tâche précédament appelée ait terminé complètement son exécution (utilisation de la fonction `kill_task(...)` qui met à jour un drapeau partagé avec la tâche).
- `ASYNC_DMA_END` : la tâche a démarré une communication SPI et la sous-tâche précédament appelée a terminé son exécution. La tâche peut alors continuer son exécution en traitant notamment les données reçues si elle en attendait.

Le même principe peut être appliqué pour d'autres protocoles de communication comme l'I2C ou l'UART.

4.2.2 Les tâches de communications

Chaques tâches de communication se construisent sur le même modèle.

Code 4.3 – Patron d'une tâche de communication

```
void ASYNC____TxRx_DMA_init(TASK      *task,
                            ...,
                            uint8_t   *tx_buf,
                            uint8_t   *rx_buf,
                            uint16_t   tx_size,
                            uint16_t   rx_size);

void ASYNC____TxRx_DMA(SCHEDULER *scheduler, TASK *self);
```

Elles prennent alors toutes en paramètre un tableau de données à transmettre `tx_buf`, un tableau de données à recevoir `rx_buf`, la taille de ces tableaux `tx_size` et `rx_size`. Si il n'est question que de transmettre ou que de recevoir des données, il est possible de mettre `NULL` pour le tableau de données. Cependant, il est nécessaire de préciser la taille du tableau à 0. A noter que le tableau `tx_buf` est copié dans un buffer interne lors de l'appel de la fonction `ASYNC____TxRx_DMA_init`. Il n'est donc pas nécessaire de conserver le tableau `tx_buf`.

La tâche se termine une fois que la communication SPI est terminée et que les données reçues ont été copiées dans le tableau `rx_buf`. Après la détection de la fin de la tâche il est possible de traiter les données reçues présentent dans le tableau `rx_buf`.

4.3 Mémoire Flash W25Q_XXX

La mémoire flash W25Q_XXX est un composant de stockage de données non volatiles de la marque Winbond. Elle est utilisée dans le projet OMEGAAA pour stocker les données acquises par les capteurs. La mémoire flash W25Q_XXX utilise une interface SPI pour communiquer avec le microcontrôleur. Elle est composé de plusieurs blocs de mémoire appelés secteurs. Chaque secteur est composé de 16 pages de 256 octets, soit 4 Ko par secteur. La mémoire flash W25Q_XXX dispose de plusieurs commandes permettant nottamant de lire, écrire, effacer des données, de connaitre et modifier l'état du composant.

Les fichiers correspondant à la gestion de la mémoire flash W25Q_XXX sont les suivants :
— `w25q_mem.h` : Déclaration des structures et des fonctions
— `w25q_mem.c` : Implémentation des fonctions

4.3.1 Déclaration des structures et des fonctions

`W25Q_Chip`

La structure contient les informations nécessaires pour communiquer avec la mémoire flash et pour gérer son état.

```
typedef struct W25Q_Chip {
        SPI_HandleTypeDef_flag *hspi_flag;
        GPIO_TypeDef           *csPinBank;
        uint16_t                csPin;
        bool                    status_bits[24];
        bool                    ASYNC_busy;
} W25Q_Chip;

void W25Q_Init(W25Q_Chip               *w25q_chip,
               SPI_HandleTypeDef_flag  *hspi_flag,
               GPIO_TypeDef            *csPinBank,
               uint16_t                 csPin,
               uint32_t                 id);
```

Les trois premiers champs de la structure `W25Q_Chip` sont des informations nécessaires pour la communication SPI.

Le tableau `status_bits` contient les bits d'état de la mémoire flash. Ces bits servent par exemple à indiquer si la mémoire flash est occupée par une opération interne. En effet, lorsque la mémoir flash est en train d'écrire ou d'effacer des données, elle ne peut pas effectuer d'autres opérations et ainsi bons nombre de commandes envoyer ne seront pas pris en compte par le composant.

Le champ `ASYNC_busy` est un booléen qui indique si la mémoire flash est occupée par une opération asynchrone. Cela permet d'assurer que certaines tâches asynchrones, puissent s'executer une par une et non pas en parallèle.

La fonction `W25Q_Init` permet d'initialiser la structure `W25Q_Chip` avec les informations nécessaires pour communiquer avec la mémoire flash. Elle prend en paramètre un pointeur vers la structure `W25Q_Chip`, les informations nécessaires pour la communication SPI ainsi que l'identifiant de la mémoire flash W25Q_XXX. L'identifiant est utilisé pour vérifier le bon fonctionnement de la communication SPI avec la mémoire flash.

ASYNC_W25Q_ReadStatusReg :

La tâche `ASYNC_W25Q_ReadStatusReg` permet de lire les bits d'état de la mémoire flash. La mémoire flash W25Q_XXX disposant de trois registres d'état, cette tâche envoie trois commandes de lecture pour récupérer les bits d'état de chacun des registres. Elle mettera ensuite à jour le tableau `status_bits` de la structure `W25Q_Chip` passée au préalable en paramètre.

```
void ASYNC_W25Q_ReadStatusReg_init(TASK *task, W25Q_Chip *w25q_chip);
void ASYNC_W25Q_ReadStatusReg(SCHEDULER *scheduler, TASK *self);
```

La tâche se termine lorsque tous les bits d'état ont été lus et stockés dans le tableau `status_bits`.

ASYNC_W25Q_WriteEnable :

La tâche `ASYNC_W25Q_WriteEnable` permet d'activer l'écriture dans la mémoire flash. En effet, avant de pouvoir écrire ou d'effacer des données, il est nécessaire qu'un certain bit d'état se trouvant dans un registre de la mémoire flash soit activé. Cette tâche envoie la commande d'activation de l'écriture à la mémoire flash.

```
void ASYNC_W25Q_WriteEnable_init(TASK *task, W25Q_Chip *w25q_chip);
void ASYNC_W25Q_WriteEnable(SCHEDULER *scheduler, TASK *self);
```

La tâche se termine lorsque l'envoie de la commande d'activation a été programmé dans le scheduler. Cependant, pendant l'exécution de la tâche, cette dernière transfert le pointeur `self->is_done` à la tâche se chargeant de l'envoie de la commande. Ainsi, lorsque l'entité génératrice de la tâche `ASYNC_W25Q_WriteEnable` détecte la fin de son exécution, cela revient à détecter que la commande s'est rélement bien envoyé. (Cette technique est utilisé afin d'optimiser le nombre de tâche en cours d'exécution.)

ASYNC_W25Q_WaitForReady :

La tâche `ASYNC_W25Q_WaitForReady` permet d'attendre que la mémoire flash soit prête à écrire ou à effacer des données. Elle appelle la tâche `ASYNC_W25Q_ReadStatusReg` pour lire les bits d'état de la mémoire flash jusqu'à ce que le bit d'état indiquant que la mémoire flash est occupée soit à 0.

```
void ASYNC_W25Q_WaitForReady_init(TASK *task, W25Q_Chip *w25q_chip);
void ASYNC_W25Q_WaitForReady(SCHEDULER *scheduler, TASK *self);
```

La tâche se termine lorsque le bit d'état indiquant que la mémoire flash est occupée est à 0.

ASYNC_W25Q_EraseSectore :

La tâche `ASYNC_W25Q_EraseSectore` permet d'effacer un secteur de la mémoire flash. Elle envoie la commande d'effacement d'un secteur à la mémoire flash. A noter que la tâche attend d'abord que le booléen `ASYNC_busy` de la structure `W25Q_Chip` soit faux avant de s'executer proprement et le met ensuite à vrai.

```
void ASYNC_W25Q_EraseSectore_init(TASK *task, W25Q_Chip *w25q_chip, uint32_t addr);
void ASYNC_W25Q_EraseSectore(SCHEDULER *scheduler, TASK *self);
```

La tâche se termine lorsque l'envoie de la commande d'effacement s'est bien déroulé. Elle mettra alors le booléen `ASYNC_busy` à faux avant de disparaitre.

ASYNC_W25Q_EraseAll :

La tâche `ASYNC_W25Q_EraseAll` permet d'effacer toute la mémoire flash. Elle envoie la commande d'effacement de toute la mémoire flash à la mémoire flash. A noter que la tâche attend d'abord que le booléen `ASYNC_busy` de la structure `W25Q_Chip` soit faux avant de s'executer proprement et le met ensuite à vrai. L'opération d'effacement de toute la mémoire flash est une opération longue qui peut dépendant de la taille de la mémoire flash prendre plusieurs dizaines de secondes.

```
void ASYNC_W25Q_EraseAll_init(TASK *task, W25Q_Chip *w25q_chip);
void ASYNC_W25Q_EraseAll(SCHEDULER *scheduler, TASK *self);
```

La tâche se termine lorsque l'envoie de la commande d'effacement s'est bien déroulé. Elle mettra alors le booléen `ASYNC_busy` à faux avant de disparaitre.

ASYNC_W25Q_ReadData :

La tâche `ASYNC_W25Q_ReadData` permet de lire un certain nombre d'octets dans la mémoire flash à partir d'une adresse donnée. Elle envoie la commande de lecture de données à la mémoire flash. A noter que la tâche attend d'abord que le booléen `ASYNC_busy` de la structure `W25Q_Chip` soit faux avant de s'executer proprement et le met ensuite à vrai.

```
void ASYNC_W25Q_ReadData_init(TASK      *task,
                              W25Q_Chip *w25q_chip,
                              uint8_t   *data,
                              uint32_t   addr,
                              uint32_t   data_size);
void ASYNC_W25Q_ReadData(SCHEDULER *scheduler, TASK *self);
```

Le paramètre `data` est un pointeur vers un tableau d'octets qui contiendra les données lues. Le paramètre `addr` est l'adresse à partir de laquelle les données seront lues. Le paramètre `data_size` est le nombre d'octets à lire. La tâche requiert que le tableau `data` soit alloué, que sa taille soit suffisante pour contenir les données lues et qu'il reste en mémoire jusqu'à la fin de l'exécution de la tâche. La tâche se termine lorsque l'envoie de la commande de lecture s'est bien déroulé. Elle mettra alors le booléen `ASYNC_busy` à faux avant de disparaitre.

ASYNC_W25Q_PageProgram :

La tâche `ASYNC_W25Q_PageProgram` permet d'écrire un certain nombre d'octets dans la mémoire flash à partir d'une adresse donnée. L'opération d'écriture ne permet d'écrire que sur des données préalablement effacées et seulement sur une page de 256 octets. Si l'utilisateur tente d'écrire plus de données que la taille restante de la page, la tâche écrira uniquement les données qui peuvent tenir dans la page. A noter que la tâche attend d'abord que le booléen `ASYNC_busy` de la structure `W25Q_Chip` soit faux avant de s'executer proprement et le met ensuite à vrai.

```
void ASYNC_W25Q_PageProgram_init(TASK      *task,
                                 W25Q_Chip  *w25q_chip,
                                 uint8_t    *data,
                                 uint32_t    addr,
                                 uint16_t    data_size);
void ASYNC_W25Q_PageProgram(SCHEDULER *scheduler, TASK *self);
```

Le paramètre `data` est un pointeur vers un tableau d'octets qui contient les données à écrire. Le paramètre `addr` est l'adresse à partir de laquelle les données seront écrite. Le paramètre `data_size` est le nombre d'octets à écrire. La tâche effectue une copie des données à écrire dans un tableau interne au moment de son initialisation. Il n'est donc pas nécessaire de conserver le tableau `data` en mémoire jusqu'à la fin de l'exécution de la tâche.

La tâche se termine lorsque l'envoie de la commande d'écriture s'est bien déroulé. Elle mettra alors le booléen `ASYNC_busy` à faux avant de disparaitre.

`ASYNC_W25Q_WriteData :`

La tâche `ASYNC_W25Q_WriteData` permet d'écrire un certain nombre d'octets dans la mémoire flash à partir d'une adresse donnée en permettant d'écrire sur plusieurs pages.

```
void ASYNC_W25Q_WriteData_init(TASK       *task,
                               W25Q_Chip *w25q_chip,
                               uint8_t    *data_buf,
                               uint32_t    addr,
                               uint32_t    data_size);
void ASYNC_W25Q_WriteData(SCHEDULER *scheduler, TASK *self);
```

Le paramètre `data_buf` est un pointeur vers un tableau d'octets qui contient les données à écrire. Le paramètre `addr` est l'adresse à partir de laquelle les données seront écrite. Le paramètre `data_size` est le nombre d'octets à écrire. La tâche effectue une copie des données à écrire dans un tableau interne au moment de son initialisation. Il n'est donc pas nécessaire de conserver le tableau `data_buf` en mémoire jusqu'à la fin de l'exécution de la tâche. La tâche découpe les données à écrire enplusieurs pages de 256 octets et appelle la tâche `ASYNC_W25Q_PageProgram` pour écrire dans chacune des pages. La tâche se termine lorsque toutes les pages ont été écrites. [SCHEMA NECESSAIRE ?]

4.3.2 Exemple d'utilisation

Illustrons l'utilisation de la mémoire flash W25Q_XXX avec un exemple simple. Dans cet exemple, nous allons simplement écrire et lire des données dans la mémoire flash.

Code 4.4 – Exemple d'utilisation de la mémoire flash W25Q_XXX

```
#include "w25q_mem.h"
#include "scheduler.h"


typedef enum STATUS {
    INIT,
    WRITE,
    READ,
    STOP
} STATUS;


void main() {
    W25Q_Chip w25q_chip;
    W25Q_Init(&w25q_chip, &hspi1_flag, GPIOA, GPIO_PIN_4, W25Q_ID);

    SCHEDULER scheduler;
    init_scheduler(&scheduler);

    STATUS status = INIT;

    bool is_done = false;

    uint8_t data_write[256*4] = {...};
    uint8_t data_read[256*5] = {0};


```

```
28      while (1) {
29          switch (status) {
30          case INIT: {
31              TASK *task_write = add_task(&scheduler, ASYNC_W25Q_WriteData);
32              ASYNC_W25Q_WriteData_init(task_write, &w25q_chip,
33                                        data_write, 0, 256*4);
34              task_write->is_done = &is_done;
35              status = WRITE;
36              break;}
37          case WRITE: {
38              if (is_done) {
39                  is_done = false;
40                  TASK *task_read = add_task(&scheduler, ASYNC_W25Q_ReadData);
41                  ASYNC_W25Q_ReadData_init(task_read, &w25q_chip,
42                                           data_read, 0, 256*5);
43                  task_read->is_done = &is_done;
44                  status = READ;
45              }
46              break;}
47          case READ: {
48              if (task_read->is_done) {
49                  // Can do something with data_read here
50                  status = STOP;
51              }
52              break;}
53          case STOP: { break; }
54          }
55          run_scheduler(&scheduler);
56      }
57  }
```

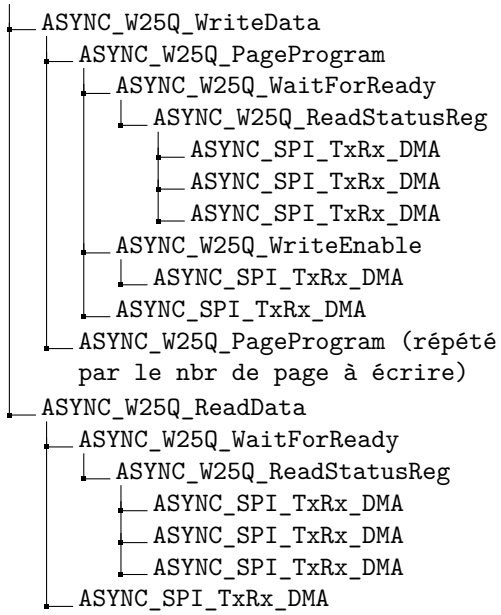


FIGURE 4.3 – Arboressance des tâches pour l'exemple 4.4

Ce programme est une machine à état qui écrit des données dans la mémoire flash, puis les lit et enfin s'arrête. La fonction `main` initialise la mémoire flash et le scheduler. Elle crée ensuite une tâche pour écrire des données dans la mémoire flash. Lorsque la tâche est terminée, elle crée une tâche pour lire les données écrites. Lorsque la tâche de lecture est terminée, le programme ne fait plus rien.

L'arboressance des tâches pour cet exemple est représentée dans le schéma ci-contre. On peut voir alors que l'écriture de données dans la mémoire flash est une opération complexe qui nécessite plusieurs tâches pour s'executer. En effet, l'appelle à la tâche `ASYNC_W25Q_PageProgram` entraine la génération de 8 sous-tâches. On peut donc voir l'importance de la gestion des tâches asynchrones et la nécessité d'avoir une pleine conscience de l'arboressance des tâches pour éviter d'être à cours de ressources.

4.4 Flash Stream

Le module `flash_stream` est un composant logiciel permettant de gérer la lecture et l'écriture de données dans une mémoire flash tel que la W25Q tout en gérant les pointeurs de lecture et d'écriture.

Les fichiers correspondants à ce module sont les suivants :
— `flash_stream.h`
— `flash_stream.c`

4.4.1 Déclaration des structures et des fonctions

`FLASH_STREAM`

La structure `FLASH_STREAM` contient un pointeur vers un composant W25Q ainsi que les pointeurs de lecture et d'écriture.

```
typedef struct FLASH_STREAM {
    W25Q_Chip   *flash_chip;

    uint32_t     write_ptr;
    uint32_t     read_ptr;
} FLASH_STREAM;

void flash_stream_init(FLASH_STREAM* stream, W25Q_Chip* flash_chip);
```

Les pointeurs de lecture et d'écriture sont initialisés à 0 lors de l'appel de la fonction `flash_stream_init`.

`ASYNC_fs_read` et `ASYNC_fs_write`

Les tâches `ASYNC_fs_read` et `ASYNC_fs_write` permettent respectivement de lire et d'écrire des données dans une mémoire flash. Leur contexte étant similaire, ils sont initialisés par la même fonction `ASYNC_fs_read_write_init`.

```
void ASYNC_fs_read_write_init(TASK *task, FLASH_STREAM *stream,
                              uint8_t *data, uint16_t len);
void ASYNC_fs_read(SCHEDULER *scheduler, TASK *self);
void ASYNC_fs_write(SCHEDULER *scheduler, TASK *self);
```

Une fois appelées pour la première fois, les tâches `ASYNC_fs_read` et `ASYNC_fs_write` incrémentent respectivement les pointeurs de lecture et d'écriture de la taille des données lues ou écrites. Elle appellent ensuite la tâche `ASYNC_W25Q_ReadData` ou `ASYNC_W25Q_WriteData` puis se terminent sans attendre la fin de l'opération. Cependant, elles transmettent à la tâche précédament générée le pointeur vers le booléen `self->is_done` ce qui permet à la tâche appelante de détecter la fin de l'opération. Puisque aucune copie de données n'est effectuée, le tableau `data` doit rester valide jusqu'à la fin de l'opération.

`ASYNC_fs_read_floats` et `ASYNC_fs_write_floats`

Les tâches `ASYNC_fs_read_floats` et `ASYNC_fs_write_floats` sont des versions des tâches `ASYNC_fs_read` et `ASYNC_fs_write` spécialisées pour la lecture et l'écriture de données de type `float`. Tous comme ces dernières, elles sont initialisées par la même fonction `ASYNC_fs_read_write_floats_init`.

```
void ASYNC_fs_read_write_floats_init(TASK *task, FLASH_STREAM *stream,
                                     float *data, uint16_t len);
void ASYNC_fs_read_floats(SCHEDULER *scheduler, TASK *self);
void ASYNC_fs_write_floats(SCHEDULER *scheduler, TASK *self);
```

Du fait de la conversion des données de type `float` en tableau de type `uint8_t`, les tâches `ASYNC_fs_read_floats` et `ASYNC_fs_write_floats` nécessitent l'utilisation de buffers internes pour stocker les données converties. Ces buffers sont alloués dynamiquement lors de l'appel de la fonction d'initialisation et libérés à la fin de l'opération. Contrairement aux tâches `ASYNC_fs_read` et `ASYNC_fs_write`, les tâches `ASYNC_fs_read_floats` et `ASYNC_fs_write_floats` restent actives jusqu'à la fin de l'opération (nécessité de maintenir les buffers internes existant).

[TO BE CONTINUED...]

A Projet Unknown

A.1 Rapport du projet

Le domcument suivant peut être retrouver en annexe du rapport du projet SP-01, projet de fusée amateur dans lequel le module Unknown a été intégré. (cf. https ://www.aeroipsa.com/sp01)

Unknown

Abbreviations

VF : Vincent FAUQUEMBERGUE
MCG : Mathieu COQUELLE-GRANDSIMON
AP : Alexis PAILLARD

Préface

Suite au projet HANAMI lancé au C'Space 2023, VF a eu l'idée de réaliser un module de « tracker » afin de retrouver les fusées en cas de vol balistique ou de vol mystère. L'objectif secondaire est d'enregistrer des données de vol qui permettrait de comprendre pourquoi le vol a échoué. Le projet est initié en Août 2023 pour être placé dans une fusée expérimentale de l'association AEROIPSA. La fusée acceptant d'inclure le module Unknown à son bord est SP-01 dirigé par MCG.

En septembre 2023, le projet est présenté à l'occasion du forum des associations comme un projet unique. Environ 5 personnes rejoignent le projet mais seuls AP et VF s'investisse dans celui-ci, la quantité de travail n'étant pas colossal et les autres membres ayants d'autres projets à réaliser. VF étant en Allemagne au premier semestre, le projet démarre réellement à partir de Décembre 2023 – Janvier 2024.

Remerciements

Je tiens à remercier le Bureau d'AEROIPSA pour m'avoir fait confiance pour mener à bien ce projet ainsi que Mathieu pour avoir accepté d'intégrer le module dans sa fusée SP-01.

Je remercie également Julien DENAT qui nous a beaucoup conseillé sur le choix des composants ainsi que sur certaines notions techniques.

Un grand merci à Planète Sciences et plus particulièrement Florent pour son aide en télémesure. Merci aussi à Paul pour ses conseils d'amélioration.

Enfin, un immense merci à mon camarade Alexis PAILLARD sans qui le projet n'aurait jamais vu le jour.

Organisation du Projet

Pour ce projet de module, nous avons réparti les tâches de cette façon :

Vincent FAUQUEMBERGUE : Conception HARDWARD
Alexis PAILLARD : Réalisation SOFTWARE

Document rédigé par Vincent FAUQUEMBERGUE

Expériences du Projet

Les expériences du projet Unknown sont les suivantes :

Expérience principale : Relocaliser une fusée après sont lancement grâce à un module de télémesure LoRa renvoyant les données GNSS tout au long du vol.

Expérience secondaire : Réalisation d'une collecte de données provenant de nombreux capteurs, barométrique et centrale inertielle, afin de reconstituer le vol après récupération des données stockées sur une mémoire flash.

Expérience tertiaire : Réaliser une intégration suffisamment petite pour pouvoir être intégrée dans n'importe quelle fusée expérimentale (FusEx).

Composants électroniques intégrés

Microcontrôleur : **STM32F411CEU6** de STMicroelectronics. Consommation max de 1.6 mA à 16MHz.

Télémesure : Module **LoRa RFM96W** permettant de communiquer en 433MHz (433.365 MHz alloué durant le C'Space 2024) à une puissance de 10 mW. Consommation max de 100 mA

Centrale inertielle : **BMI088** de Bosch, 6 axes (3 axes d'accélération et 3 axes gyroscope) fréquence d'acquisition choisis de 80 Hz. Consommation max de 5.15 mA

Accéléromètre (Haute accélération) : **ADXL375BCCZ-RL7** de Analog Devices. La particularité de cet accéléromètre est de pouvoir une valeur de l'accélération jusqu'à 200g. Consommation max de 0.1 mA

Baromètre : **BMP388** de Bosch avec une précision de 0.5 hPa, très bon rapport qualité/prix pour une mesure d'altitude barométrique. Consommation max de 5mA

Mémoire Flash : **W25Q128JVPIQTR** de Winbond capacité de 128 Mbit soit 16Mo. Consommation max de 25 mA.

GNSS : **SAM-M10Q-00B** de ublox avec un cold start de 29s. Consommation max 100mA.

Buzzer : **MLT-5020** de JIANGSU HUAWHA ELECTRONICS CO, 75 dB. Consommation max 100mA.

Régulateur de tension : **TPS62172DSGR** de Texas Instruments. Consommation négligeable (10 µA)

Au total, la consommation, avec l'hypothèse que tous les composants soient à leur consommation maximale, est de 336.85 mA. Cela revient à une puissance de $P_{tot} = 3.3V \times 336.85mA = 1111.6\ mW$

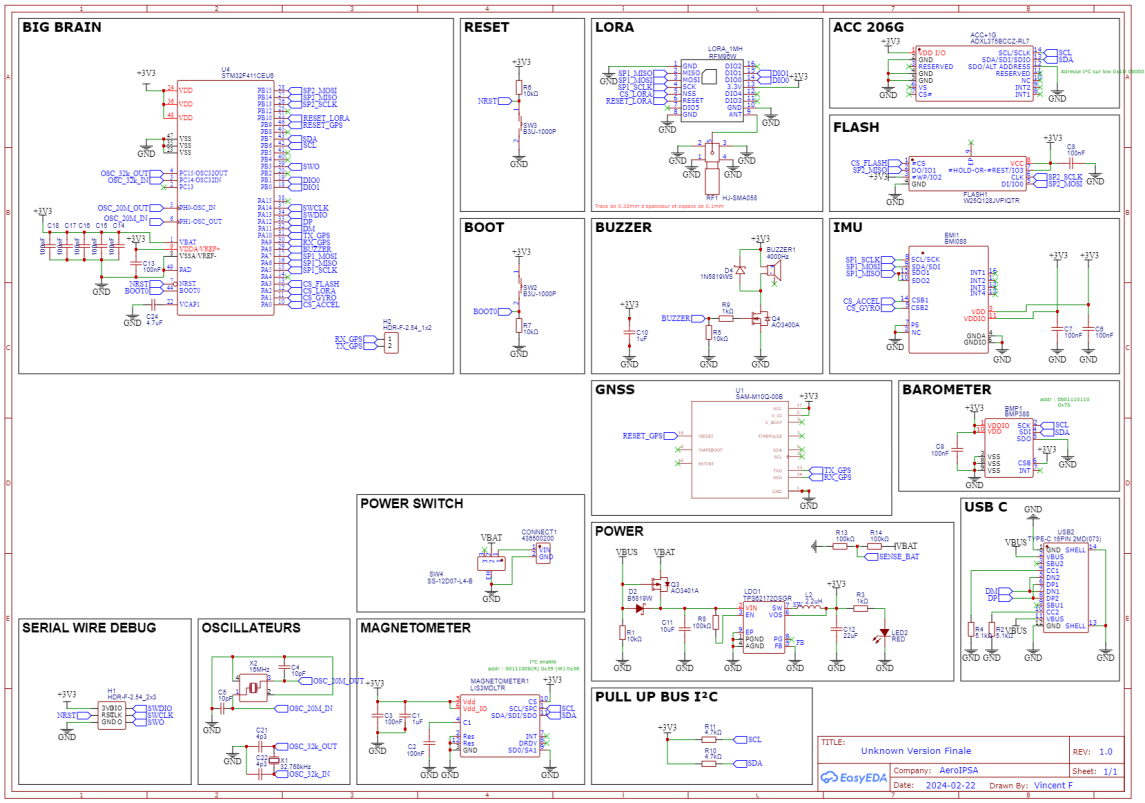
Document rédigé par Vincent FAUQUEMBERGUE

Nous voulons opérer le module pour une durée de 3h minimum conformément au cahier des charges FusEx. De plus, nous utiliserons une batterie 1S LiPo d'une tension de 3.7V. Cherchons maintenant la puissance nécessaire pour pouvoir subvenir au module :

$$Q = \frac{3h \times 1.1116\,W}{3.7V} = 904\,mAh$$

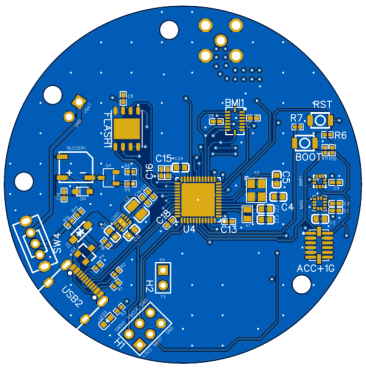
Ainsi, sachant que l'hypothèse que tous les composant soit à leur consommation maximale durant toute la phase de vol est impossible, nous avons choisis d'utiliser une batterie **LiPo 1S 3.7V 900 mAh.**

Schéma électronique

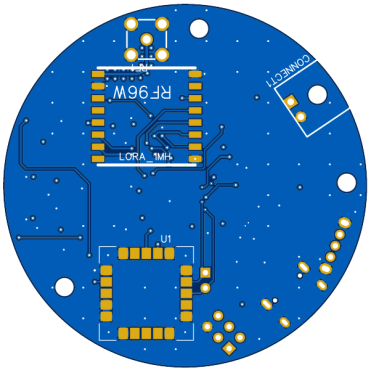


Document rédigé par Vincent FAUQUEMBERGUE

Circuit imprimé

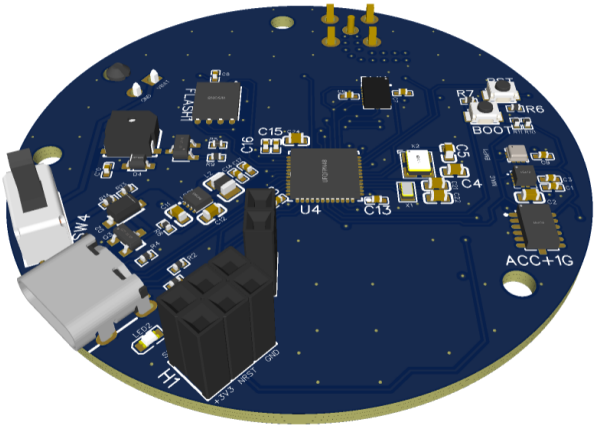


Face avant du module Unknown



Face arrière du module Unknown

Vue 3D du circuit imprimé



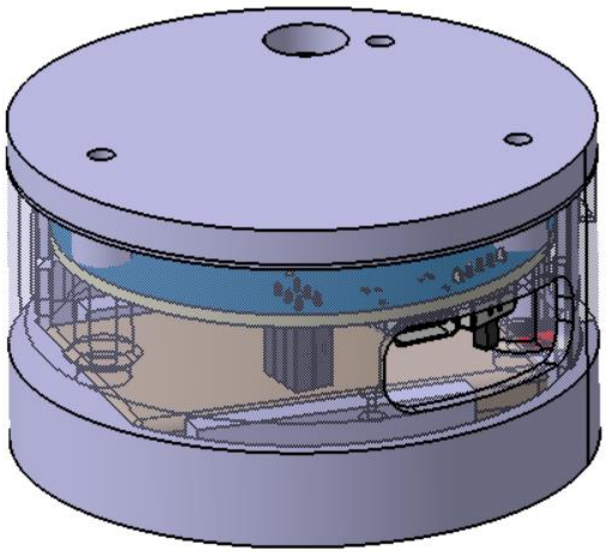
Face avant 3D du module Unknown

Intégration mécanique du module Unknown

Le module est intégré dans la coiffe. Celle-ci est réalisée en fibre de verre permettant de laisser passer les ondes RF indispensable pour la réception GNSS et LoRa. La fixation de la carte est de la batterie se fait grâce à 3 pièces en PLA.

$D_e = 70\ mm$
$H = 41\ mm$

Document rédigé par Vincent FAUQUEMBERGUE



La bague supérieure permet de fixer le module à une autre bague fixée dans la coiffe. La bague centrale permet d'accéder à l'interrupteur d'allumage du module ainsi qu'au branchement de la batterie. Enfin, la bague inférieure permettant de fixer la batterie LiPo pour que ne puisse bouger et ainsi éviter tous risques de dégâts.

Paramètres du Software

Télémesure :

- Puissance d'émission : 10 mW (cahier des charges)
- Spreading factor : 7

Paramètre généraux

- Fréquence d'acquisition de données : 80Hz
- Temps possible prise de données : 1h15

Après demande auprès de Planètes Sciences, il nous a été accordé d'envoyer un « Go » au module pour démarrer la prise de données.

Document rédigé par Vincent FAUQUEMBERGUE

Déroulement d'un vol simulé du module

- Allumage du module
- Réception sur la station au sol d'un « Wait for GPS »
- Lorsque le module à réussi à se synchroniser avec une constellation, celui-ci envoie un « Go ? » à la station au sol
- La station au sol envoie alors le « Go ! » de confirmation. La prise de données commence pour une durée de 1h15 = 75 minutes.

Résultats du vol

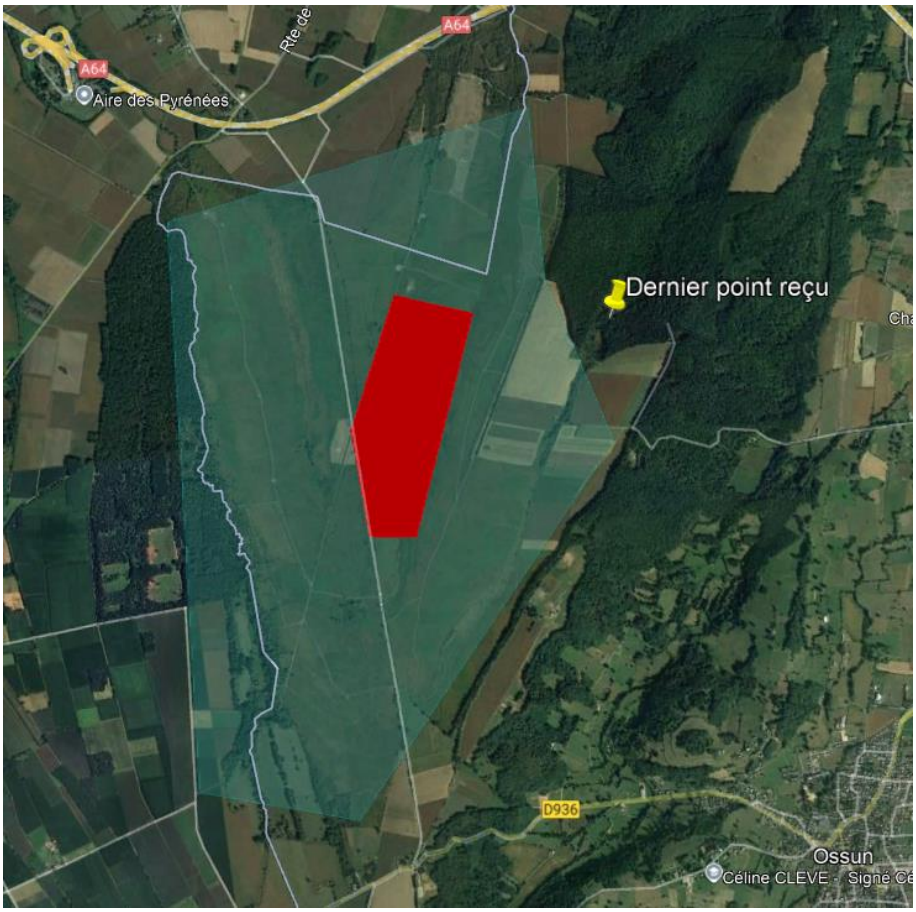
Le vol du module Unknown a eu lieu sur la fusée Sp-01 le jeudi 11 juillet 2024 à 10h55. Sur la chronologie, une procédure de secours était écrite en cas de soucis d'initialisation du module dû à la distance ainsi que les obstacles présent entre la station au sol et le module. De plus, nos antennes avaient de faibles gains (2 – 3 dBi). Malheureusement, nous avons dû déclencher cette procédure de secours, AP ne recevant pas de « Wait for GPS ». Après correction de ce souci, nous avons bien reçu les données GNSS durant toute la phase de vol.

Malgré son vol nominal, la fusée SP01 étant très petite et rapide n'a pas était bien aperçue par les équipes aux jumelles. Cependant, une jumelle a réussi à la repérer durant sa phase de descente. Nous avons ensuite donné notre dernière position GNSS connue à 228 m d'altitude à la localisation qui se concordait avec les données fournies par la jumelle. La fusée à était récupérer à une trentaine de mètres du dernier point GNSS permettant de conclure avec succès la mission principale du module Unknown.

Dernière ligne envoyée par le module Unknown à la station au sol. Pour rappel, le Camp de Ger est à une altitude d'environ 450 mètres.

10:56:25:794 -> GPS -> long: -0.041453, lat: +43.21362, alt: +678.0

Document rédigé par Vincent FAUQUEMBERGUE



Traitement des données post vol

Après avoir récupéré le module le vendredi 12 juillet 2024 à 15h30, nous avons procédé à l'analyse des données. Nous avons vite aperçu un problème. Dans un premier temps, l'ensemble de nos données étaient fixes, on ne voyait donc aucune accélération, ni déplacement GNSS. Un second problème constitue le nombre de lignes qui aurait dû être de 380 000, or nous en avons obtenu 95 000 soit exactement 4 fois moins. Ce chiffre de 4 nous a tout de suite mis la puce à l'oreille.

En effet, nous avons remarqué que la DATA_SIZE que nous avons définie était de 44 au lieu de 11. Ainsi, au lieu d'écrire 44 octets de données, nous en écrivions 176, ce qui explique la perte de ¾ des données (44 valeurs et 132 « Zéros »).

Document rédigé par Vincent FAUQUEMBERGUE

```
41
42  void save_data_to_flash(FLASH_STREAM *stream, DATAS datas) {
43      float datas_buff[DATA_SIZE] = {datas.accel[0],
44                                     datas.accel[1],
45                                     datas.accel[2],
46                                     datas.gyro[0],
47                                     datas.gyro[1],
48                                     datas.gyro[2],
49                                     datas.pressure,
50                                     datas.temp,
51                                     datas.longitude,
52                                     datas.latitude,
53                                     datas.time};
54
55      flash_stream_write_floats(stream, datas_buff, DATA_SIZE);
56
57  >   // flash_stream_write_float(stream, datas.accel[0]);…
68  }
```

Ainsi, au lieu d'avoir 75 minutes de prise de données, nous n'en avions que 19. Nous pouvons vérifier que la fusée n'a pas bougé sur cette plage en regardant la différence de temps entre le moment du « Go ! » et le moment du décollage. En regardant les données de télémesures, on remarque que le « Go ! » a été envoyé à 10h22min19s tandis que le décollage a eu lieu à 10h54min44s ce qui correspond à une durée de 32 minutes environ dépassant largement la durée de 19 minutes. Nous n'avons donc malheureusement aucune donnée à recueillir.

10:22:59:596 -> [Arduino] go !
10:54:44:073 -> G56%}8#long: (0~86761, lQrº :2®3ò|>4²y nD9j -> Décollage

Perspectives d'améliorations

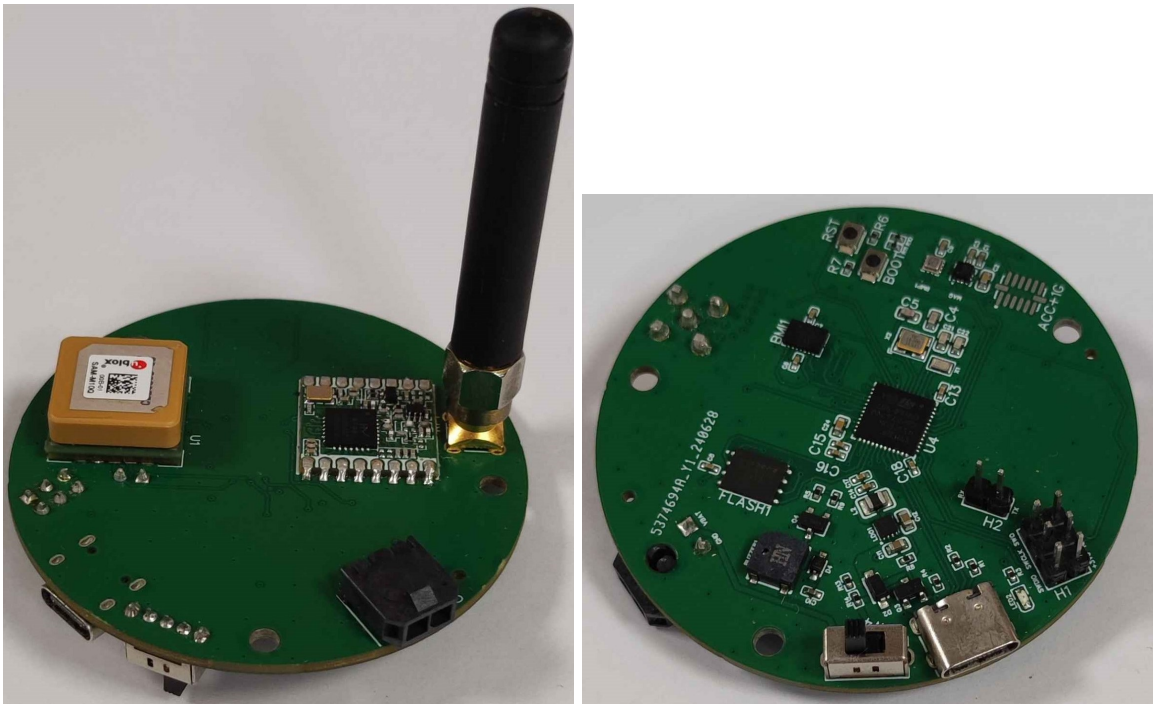
Afin d'améliorer ce module, nous comptons corriger les soucis concernant l'écriture de donnée ainsi qu'améliorer la capacité de stockage en passant d'une flash de 16 Mo à 32 voir 64 Mo afin de garantir une plus longue plage de prise de données.

Nous comptons également changer la fréquence de LoRa en passant sur la plage 869.4 – 869.65 MHz afin de pouvoir monter jusque 500 mW de puissance. Nous comptons également réaliser une vraie station au sol avec des antennes ayant un meilleur gain ainsi que placer quelqu'un avec une antenne Yagi (11 – 13 dBi) en zone public.

Ce projet nous a néanmoins appris beaucoup notamment sur l'électronique et la télémesure. Ce fut une expérience enrichissante dont nous espérons tirer profit pour le C'Space 2025.

Document rédigé par Vincent FAUQUEMBERGUE

A.2 Photos du projet Unknown



(a) Vu de dessus (b) Vu de dessous

FIGURE A.1 – Photos du projet Unknown