22 Juin 2020 — 1er Octobre 2020

Conception d'un système embarqué permettant la capture et le stockage d'images pour un ballon sonde

---

Antton Bodin
2ème Année Électronique

Enseignant référant :
M. Yannick Bornat

Remerciements

Je tiens à remercier tout particulièrement M. Yannick Bornat pour l'aide et les connaissances qu'il m'a apportées tout au long de ce projet. Les multiples réunions ont été, pour moi, des sources d'enrichissement et de motivation.

Je remercie également M. Anthony Ghiotto de m'avoir donné l'opportunité de prendre part à un projet d'une telle ampleur.

Je voudrais aussi remercier l'ensemble de l'équipe qui a travaillé sur ce projet : Yohan Bellanger, Edgar De Oliveira Cruz, Nigel Ignatowicz, François Vanlerberghe, M. Guillaume Ferré et M. Guy Morizet. Grâce aux nombreux échanges que j'ai pu avoir avec eux, ce projet a été une véritable réussite dans une ambiance de travail conviviale.

Pour finir, je remercie M. Guillaume Durin, aérotechnicien de l'AJSEP et ingénieur chez Ariane Group, qui à suivi toute la conception de la nacelle et nous a apporté son aide et ses connaissances jusqu'au lâcher du ballon.

Table des matières

1 Introduction **3**

2 Présentation de l'IMS **4**

3 Gestion de la caméra **5**
3.1 Initialisation de la caméra 5
3.2 Prise d'une image 7
3.3 Traduction du protocole en C++ 9

4 Gestion de la carte SD **11**
4.1 Première utilisation de la carte SD 11
4.2 Utilisation de la carte SD pour stocker les images prises par la caméra 12

5 Gestion de la cadence de prise des photos **12**
5.1 Raisonnement et calculs 12
5.2 Initialisation des différents registres du timer et comparaison 13

6 Ajout d'une deuxième caméra **16**

7 Alimentation du système **17**

8 Conclusion **19**

1 Introduction

Le projet EIRBALLOON initié par M. Anthony GHIOTTO est un projet dont le but était le lancement d'un ballon sonde lors de l'évènement des 100 ans de l'ENSEIRB-MATMECA en Avril 2020. Du fait de la pandémie de la COVID-19, ce projet n'a pas pu aboutir dans les temps et nous avons eu l'opportunité de pouvoir le continuer cet été sous forme de stage, dans le but de lancer le ballon en Octobre 2020. Lorsque ce projet nous a été proposé, nous étions six étudiants, tous en deuxième année électronique, et l'objectif était de réaliser tous les systèmes électroniques qui allaient être embarqués dans la nacelle du ballon. Je devais alors réaliser, en binôme, un système permettant la capture et le stockage de photos sur une carte SD.
Lorsque nous avons repris le projet durant l'été, l'idée de la Raspberry a été abandonnée et nous avons décidé de réaliser le système avec une carte Arduino Mega 2560. Cette carte comporte un microcontrôleur ATmega2560 et quatre ports séries permettant de communiquer avec l'ordinateur pour le debug ou avec des caméras. Finalement, me retrouvant seul à travailler sur le développement de ce système, la transmission en direct des images a également été abandonnée.

Dans ce rapport, je vais détailler la gestion des différents éléments du système : prise de photos avec la caméra, écriture sur la carte SD, fréquence de prise des photos. Ce rapport est à la fois un compte rendu de stage servant à son évaluation mais il sera également utile à l'équipe d'étudiants qui participera au projet EIRBALLOON V2 afin d'améliorer le système que j'ai pu réaliser.

Les différents codes écrits au cours de ce projet ainsi que le code final sont tous disponibles sur mon GitHub (https://github.com/abodin0/eirballoon.git).

2 Présentation de l'IMS

Le laboratoire de l'Intégration du Matériau au Système est une Unité Mixte de Recherche formée en 2007 par la fusion de trois unités de recherche bordelaises (IXL, PIOM et LAPS) et dirigée par Yann Deval. Ce laboratoire est rattaché à trois institutions qui sont : le CNRS, l'Université de Bordeaux et Bordeaux Aquitaine INP.
L'IMS est composé de équipes de recherche réparties au sein de 10 groupes comme montré sur la figure 1.

10 Groupes	28 Equipes			
BIOELECTRONIQUE N. Lewis	Bio-EM Y. Percherancier	ELIBIO Y. Bornat	AS2N S. Saighi	
COGNITIQUE J-M André	CIH J-M André	ERGO J. Petit	RUDII A. Lehmans	PMH_DySCo L. Arsac
SIGNAL Y. Berthoumieu	MOTIVE J-P. Da Costa	SPECTRAL A. Giremus		
PRODUCTIQUE Y. Ducq	MEI M. Traoré	ICO C. Merlo	LOCO2 R. Dupas	
AUTOMATIQUE X. Moreau	ARIA D. Henry	CRONE P. Melchior	FFTG F. Cazaurang	
CONCEPTION D. Dallet	CAS F. Rivet	CSH N. Deltimple	CSN C. Jego	
FIABILITE G. Duchamp	PACE H. Frémont	PUISSANCE J-M. Vinassa		
NANOELECTRONIQUE P. Mounaix	LASER J-P. Guillet	MODEL C. Maneux	III-V N. Malbert	
ORGANIQUE I. Dufour	ELORGA M. Abbas	PRIMS C. Ayéla		
ONDES Y. Ousten	MDA C. Dejous	EDMINA L. Béchou	MIM F. Demontoux	

FIGURE 1: Répartition des équipes de recherche de l'IMS (www.ims-bordeaux.fr)

Les travaux de recherche effectués à l'IMS sont axés autour des priorités scientifiques suivantes :

- modélisation et mise en forme de matériaux pour l'élaboration de composants et microsystèmes,
- modélisation, conception, intégration et analyse de fiabilité des composants, circuits et assemblages,
- identification, commande, diagnostic, traitement du signal et des images,
- conduite des processus complexes et hétérogènes, ingénierie humaine et interactions avec le domaine du "vivant"

L'IMS compte aujourd'hui près de 350 membres qui ont permis, sur les 5 dernières années, la production de près de 800 publications, 1200 communications et 50 brevets.

3 Gestion de la caméra

La caméra utilisée dans ce projet est une caméra uCAM-III de chez 4D Systems. Cette caméra permet de prendre des photos en couleurs au format RAW ou JPEG dans plusieurs résolutions, qui seront détaillées plus loin dans ce rapport. Des objectifs de différentes focales sont également disponibles pour cette caméra. Dans le cadre du projet Eirballoon, nous avons utilisé un objectif grand angle (116 degrés) pour la caméra pointant à l'horizontale et un objectif standard (56 degrés) pour la caméra pointant à la verticale.
Cette caméra comporte 5 broches permettant l'alimentation et la communication : une broche d'alimentation 5V, une broche de masse, une broche de reset et deux broches de communications Rx et Tx. La communication entre la caméra et la carte Arduino se fait suivant le protocole UART.

3.1 Initialisation de la caméra

Les communications avec la caméra se font en suivant le protocole de commande détaillé dans la documentation de la caméra. Afin d'envoyer une commande à la caméra, il faut envoyer une suite de 6 octets en suivant le tableau de la figure 2 :

Command	ID Number	Parameter1	Parameter2	Parameter3	Parameter4
INITIAL	AA01h	00h	Image Format	RAW Resolution (Still Image only)	JPEG Resolution
GET PICTURE	AA04h	Picture Type	00h	00h	00h
SNAPSHOT	AA05h	Snapshot Type	Skip Frame (Low Byte)	Skip Frame (High Byte)	00h
SET PACKAGE SIZE	AA06h	08h	Package Size (Low Byte)	Package Size (High Byte)	00h
Set Baud Rate	AA07h	1st Divider	2nd Divider	00h	00h
RESET	AA08h	Reset Type	00h	00h	XXh*
DATA	AA0Ah	Data Type	Length Byte 0	Length Byte 1	Length Byte 2
SYNC	AA0Dh	00h	00h	00h	00h
ACK	AA0Eh	Command ID	ACK Counter	00h / Package ID Byte 0	00h / Package ID Byte 1
NAK	AA0Fh	00h	NAK Counter	Error Number	00h
LIGHT	AA13h	Frequency Type	00h	00h	00h
CONTRAST / BRIGHTNESS / EXPOSURE	AA14h	Contrast (0-4, 2 is Normal)	Brightness (0-4, 2 is Normal)	Exposure (0-4, 2 is '0')	00h
SLEEP	AA15h	Timeout (0-255)	00h	00h	00h
If the parameter is 0xFF, the command is a special Reset command and the module responds to it immediately.					

FIGURE 2: Tableau récapitulatif des commandes de la caméra extrait de la documentation (https://4dsystems.com.au/mwdownloads/download/link/id/420/)

Le protocole de commande indique que pour démarrer la communication avec la caméra il faut envoyer la commande SYNC. La documentation nous assure que la synchronisation se fera avant le 61ème envoi de cette commande. Lorsque la caméra sera synchronisée, elle retournera alors un signal ACK puis le signal de synchronisation. Une fois ces deux signaux reçus, il est alors nécessaire d'envoyer un signal ACK vers la caméra pour confirmer la synchronisation de cette dernière.
En observant le tableau de la figure 2, on remarquera que le signal de synchronisation à envoyer correspond aux octets "AA 0D 00 00 00 00". Le signal ACK correspondant à la synchronisation correspond quand à lui aux octets "AA 0E **0D** 00 00 00". En effet le paramètre 1 (ou troisième octet de ce signal) est l'identifiant de la commande à laquelle il fait référence, ici SYNC donc 0x0D.

La figure 3, extraite de la documentation, montre les échanges nécessaires entre le micro-contrôleur et la caméra lors de la synchronisation.

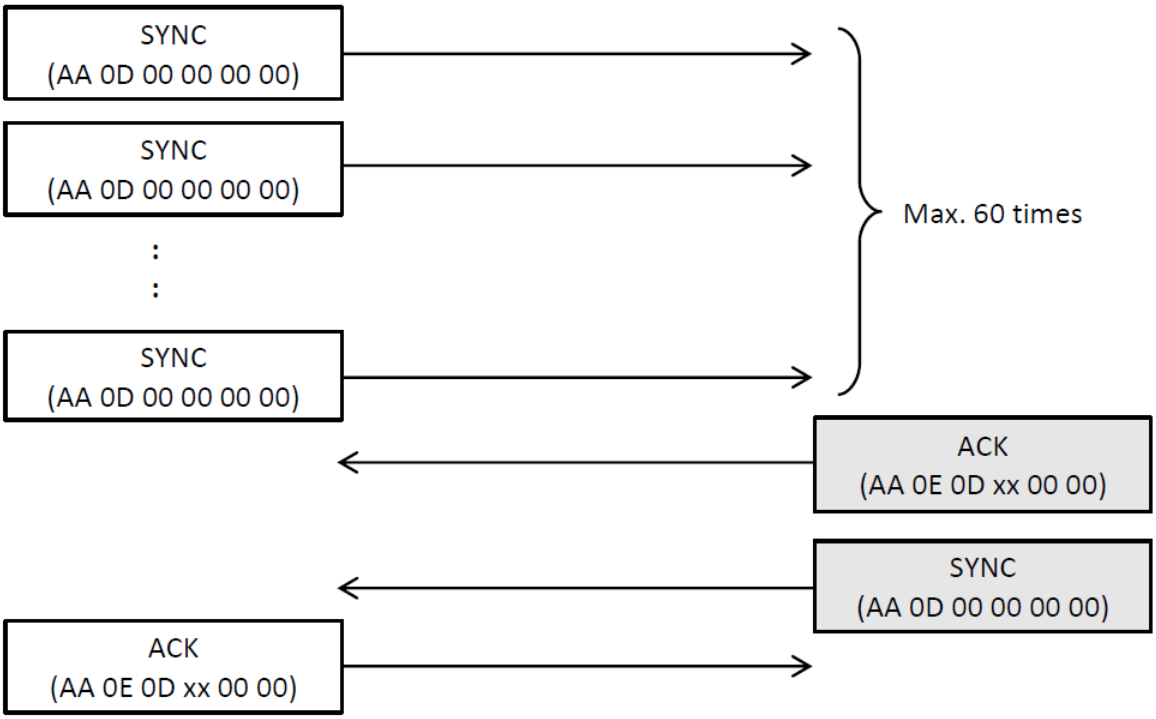


FIGURE 3: Protocole d'échange pour la synchronisation de la caméra

La suite du protocole de communication demande de sélectionner le format et la résolution des images qui seront prises par la caméra. Pour cela il faut envoyer le signal INITIAL de la figure 2 en complétant les paramètres 2, 3 et 4. Le format de l'image se choisit à l'aide du tableau de la figure 4 extrait de la documentation :

The uCAM-III can support 4 different image formats as follow:

8-bit Gray Scale (RAW, 8-bit for Y only)	03h
16-bit Colour (RAW, CrYCbY)	08h
16-bit Colour (RAW, 565(RGB))	06h
JPEG	07h

FIGURE 4: Choix du format de l'image

Le paramètre 4 est choisi grâce au tableau de la figure 5 qui permet de sélectionner une résolution pour le format JPEG :

160 x 128	03h
320 x 240	05h
640 x 480	07h

FIGURE 5: Choix de la résolution pour les images au format JPEG

On souhaite la meilleure résolution possible, nous avons donc choisi, pour ce projet, la résolution 640 × 480. Le paramètre 3 permet de sélectionner la résolution de l'image au format RAW : cela ne nous intéresse pas dans notre situation, on prendra donc l'octet 0x01 pour la plus petite résolution. Le signal qui sera envoyé par la commande INITIAL est donc "AA 01 00 07 01 07".

Pour finir l'initialisation de la caméra, il faut déterminer la taille des paquets de données transmis par la caméra. La figure 6 extraite de la datasheet nous montre comment sont décomposés les paquets de données émis par la caméra.

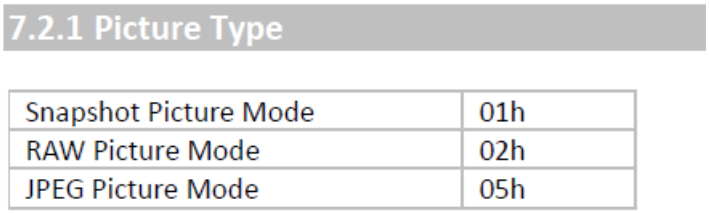
FIGURE 6: Décomposition d'un paquet de données

On remarque alors que sur un paquet de N octets, 6 ne correspondent pas à des données brutes de l'image. Afin d'avoir 256 octets de données brutes lors de la réception de l'image, on initialise la taille des paquets à 262 octets. L'initialisation se fait avec la commande SET PACKAGE SIZE (voir le tableau figure 2) où les paramètres 2 et 3 correspondent respectivement au bit de poids faible et au bit de poids fort de la taille de paquet souhaitée. On veut 262 octets soit 0x106 octets. Le bit de poids faible sera donc 0x06 et celui de poids fort 0x01.

3.2 Prise d'une image

Une fois que la caméra est correctement initialisée, nous pouvons alors prendre les premières photos. Le protocole de communication de la datasheet indique que pour prendre une photo il faut envoyer la commande SNAPSHOT, puis attendre le signal ACK de la caméra avant d'envoyer la commande GET PICTURE qui permet de demander à la caméra d'envoyer les données de l'image qui vient d'être prise. Comme on peut le voir dans le tableau 2, pour envoyer la commande SNAPSHOT, il faut envoyer les octets "AA 05" puis 4 paramètres. Le paramètre 1 correspond au format du snapshot que l'on souhaite prendre : 0x00 pour le format JPEG et 0x01 pour le format RAW. Les paramètres 2, 3 et 4 seront par défaut égaux à 0x00. Le signal complet envoyé pour réaliser la prise de l'image sera donc "AA 05 00 00 00 00". Lorsque la photo est prise, la caméra retourne un signal ACK correspondant à la commande SNAPSHOT. Le signal attendu doit donc commencer par "AA 0E 05".
Pour récupérer les données de la photo qui vient d'être prise, il faut alors envoyer la commande

GET PICTURE. Le paramètre 1 de cette commande permet de choisir le format de la photo dont on cherche à récupérer les données. Dans notre situation, les photos étant au format JPEG le paramètre 1 doit être égal à 0x05 comme le montre le tableau de la figure 7 :



7.2.1 Picture Type

Snapshot Picture Mode	01h
RAW Picture Mode	02h
JPEG Picture Mode	05h

FIGURE 7: Choix du type d'image dont on souhaite récupérer les données

La caméra retourne alors un signal ACK commençant par "AA 0E 04" pour confirmer la réception de la commande GET PICTURE. Juste après, la caméra envoie alors un signal contenant la taille de l'image qui a été prise. Ce signal commence par "AA 0A 01" et les 3 octets suivants correspondent à la taille de l'image.
Pour confirmer la bonne réception de la taille de l'image, il est nécessaire d'envoyer un signal ACK vers la caméra sous la forme "AA 0E 00 00 00 00". La caméra commence alors à transmettre les paquets de données sous la forme de la figure 6. Après la réception de chaque paquet, on envoie un signal ACK de "AA 0E 00 00 **XX** 00" ou l'octet XX correspond à l'identifiant du dernier paquet reçu.
Ce protocole est résumé sur la figure 8.

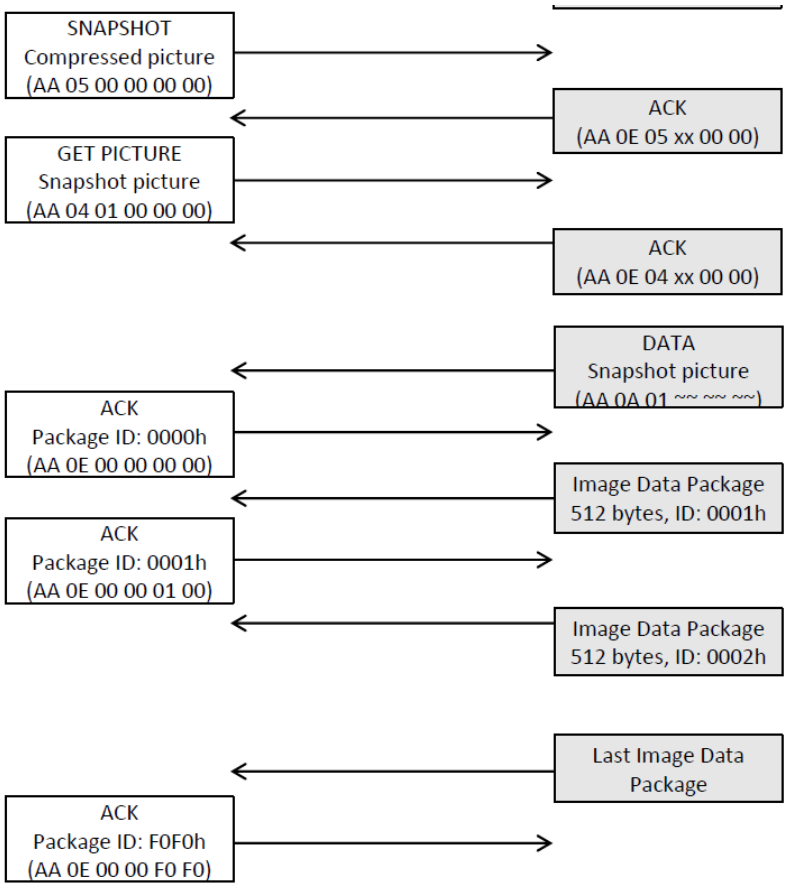


FIGURE 8: Protocole d'échange pour la prise d'une image et la récupération de ses données

3.3 Traduction du protocole en C++

Comme expliqué précédemment, la communication avec la caméra se fait via un microcontrôleur ATmega2560 intégré sur une carte Arduino. La langage de programmation utilisé est donc le C++. Un code Arduino se décompose en trois grandes parties : la déclaration des variables et des fonctions nécessaires, la fonction ***setup()*** et la fonction ***loop()***. La fonction ***setup()*** est la première fonction exécutée lorsque le programme est lancé. Elle permet d'initialiser un certain nombre de paramètres comme les communications séries ou les états de certains pins de la carte par exemple. On initialise ainsi la communication série avec l'ordinateur sur le port **Serial** à 230400 bits par seconde. La communication avec la caméra est quant à elle initialisée sur le port **Serial1** à 115200 bits par seconde. C'est dans le ***setup()*** que l'on appelle les fonctions permettant l'initialisation de la caméra.

Chaque étape du protocole de communication détaillé précédemment a été intégrée dans une fonction. Ainsi on a une fonction ***cam_sync()*** qui envoie toutes les 100ms le signal de synchronisation jusqu'à ce que la caméra réponde. Cette fonction finit par envoyer le signal ACK nécessaire comme indiqué sur la figure 3.
La commande INITIAL permettant de sélectionner le format des images est envoyé par la fonction ***cam_initiate()***.
La définition de la taille des paquets de données générés par la caméra se fait dans la fonction ***cam_set_size()***.
Pour terminer l'initialisation de la caméra, on appelle la fonction ***cam_max_baudrate()***. Cette étape permet de passer la vitesse de communication de 115200 bits par seconde à 153600 bits par seconde. Bien qu'elle ne soit pas présente dans le protocole de communication qui est en exemple dans la datasheet, cette étape est nécessaire au bon fonctionnement de notre système. Les diviseurs en paramètre 1 et 2 de cette commande sont détaillés dans le tableau de la figure 9. Le signal envoyé est donc "AA 07 07 02 00 00".

Baud Rate	1st Divider	2nd Divider
2400	31 (0x1F)	47 (0x2F)
4800	31 (0x1F)	23 (0x17)
9600	31 (0x1F)	11 (0x0B)
19200	31 (0x1F)	5
38400	31 (0x1F)	2
57600	31 (0x1F)	1
115200	31 (0x1F)	0
153600	7	2
230400	7	1
460800	7	0
921600	1	1
1228800	2	0
1843200	1	0
3686400	0	0

FIGURE 9: Choix du diviseur pour la vitesse de transmission

La fonction ***loop()*** est une boucle infinie dans laquelle on met les fonctions que l'on souhaite voir être exécutées jusqu'à la mise hors tension du système. C'est donc dans cette fonction qu'il faut prendre des photos et récupérer les données. La fonction ***snap()*** permet de prendre

un photo. On appelle en suite la fonction ***get_picture()*** qui demande alors à la caméra de transmettre les données de l'image. Il est nécessaire d'attendre 200ms entre la prise de la photo et l'instruction de demande des données, c'est pourquoi ***delay(200)*** a été ajouté.
Comme expliqué précédemment, le premier signal retourné par la caméra après la commande GET PICTURE correspond à la taille de l'image. La fonction ***data_check()*** permet ainsi de récupérer la taille de l'image sous la forme d'un *long int* puis elle envoie le signal ACK nécessaire pour recevoir les paquets de données. Dans un premier temps, les données reçues sont envoyées vers un script PYTHON permettant de reconstruire les fichiers images. Ce script, qui tourne directement sur mon ordinateur, récupère les données de la caméra via le port série et écrit les images dans un dossier. Ainsi, l'identifiant du paquet reçu est récupéré dans la variable *recv_packet_num* et la taille des données est récupérée dans la variable *data_size*. Pour récupérer ces éléments, on utilise la fonction ***get_cam_char()*** qui retourne l'octet reçu sur le port **Serial1**. On met à jour dans le même temps la valeur du checksum, qui d'après la datasheet, correspond à la somme de tous les octets du paquet, excepté les deux octets de vérification.
Dans une boucle ***for()*** de la longueur de *data_size*, on récupère chaque octet de données et on l'envoie vers le script PYTHON. Le checksum est également mis à jour à chaque itération. Lorsque tous les octets du paquet ont été reçus, on envoie alors le signal ACK correspondant à l'identifiant du paquet reçu comme montré sur la figure 8. De plus, plusieurs sources d'erreurs possibles sont vérifiées comme la valeur du checksum, le numéro du paquet reçu ou le zéro terminal à la fin du paquet.
La figure 10 présente une image prise avec la caméra et reconstruite grâce au script PYTHON.

FIGURE 10: Image prise en extérieur et reconstruite grâce au script PYTHON

4 Gestion de la carte SD

4.1 Première utilisation de la carte SD

Pour gérer la carte SD, on utilise un shield de chez Seeed Studio que l'on vient placer sur la carte Arduino. Ce shield est une carte électronique qui se branche à la carte Arduino via les différentes broches (donc sans soudure) et permet d'accueillir une carte SD. La communication entre l'Arduino et le shield SD se fait suivant le protocole de communication SPI. Ce protocole fait appel à quatre signaux :

- un signal d'horloge permettant la synchronisation de la communication entre le maître et l'esclave
- un signal MOSI (Master Output Slave Input) permettant une communication du maître vers l'esclave
- un signal MISO (Master Input Slave Output) permettant une communication de l'esclave vers le maître
- un signal CS (Chip Select) qui permet de sélectionner vers quel esclave va se faire la communication lorsque le maître possède plusieurs esclaves

L'initialisation de la communication avec la carte SD se fait comme pour la caméra en envoyant une série de commandes déterminées. Cependant, le programme ne semblant pas fonctionner correctement, il a été décidé d'utiliser une librairie développée pour l'utilisation de cartes SD depuis un système Arduino. Cette librairie est directement intégrée dans l'IDE Arduino. Il faut alors inclure <SPI.h> pour la communication SPI et <SD.h> pour pouvoir utiliser la carte SD.
L'initialisation de la carte SD se fait avec la fonction ***SD.begin(CS_PIN)*** où *CS_PIN* correspond à la broche de Chip Select. Dans notre cas, c'est la broche 4 de la carte. La figure 11 montre le code utilisé pour l'initialisation de la carte SD.

```
20  Serial.begin(230400);
21
22  Serial.print("Initializing SD card...");
23
24  if (!SD.begin(CS_PIN))
25  {
26    Serial.println("initialization failed!");
27    while (1);
28  }
29  Serial.println("initialization done.");
```

FIGURE 11: Initialisation de la carte SD en utilisant la librairie <SD.h>

On peut alors ouvrir un fichier à l'intérieur de la carte SD avec la fonction ***SD.open()***. Le fichier ouvert est un objet de type *File*. L'ouverture nécessite alors de préciser un nom et un type d'ouverture comme on peut le voir sur la figure 12.

```
33  myFile = SD.open("file.txt", FILE_WRITE);
```

FIGURE 12: Ouverture d'un fichier texte sur la carte SD

L'écriture à l'intérieur du fichier se fait en utilisant directement la classe *File*. Cette classe

possède ainsi diverses méthodes telles que ***println()*** ou ***write()***. La fermeture du fichier, lorsque toutes les modifications souhaitées ont été apportées, se fait avec la méthode ***close()***

4.2 Utilisation de la carte SD pour stocker les images prises par la caméra

Maintenant que l'on sait utiliser la librairie <SD.h> pour écrire un fichier dans la carte SD, on peut abandonner le script PYTHON qui reconstruisait les images à partir des données de la caméra. L'objectif est donc de récupérer les données pour écrire directement les images sur la carte SD. Pour cela on crée une fonction ***writing_on_sd_card()*** qui prend en argument un fichier de type *File*, le numéro de l'image et sa taille. Cette fonction se base sur le ***loop()*** de *camera.ino* qui permettait d'envoyer les données reçues de la caméra vers l'ordinateur. Ainsi, le numéro et la taille du paquet sont récupérés de la même façon que précédemment. Les octets de données ne sont alors plus envoyés vers l'ordinateur via le port **Serial** mais sont stockés dans le tableau *picdata*. Lorsque tous les octets du paquet ont été stockés dans ce tableau, on l'écrit dans le fichier image sur la carte SD de la façon suivante :

```
330        img.write(picdata, sizeof(picdata));
```

FIGURE 13: Écriture des octets d'un paquet dans un fichier image sur la carte SD

Une fois que tous les paquets de données correspondant à une image ont été écrits dans la carte SD, on ferme l'image avec ***img.close()***. Il est important d'ouvrir le fichier image avant de demander les données de l'image à la caméra (***get_picture()***). En effet, le temps nécessaire à l'ouverture d'un fichier sur la carte SD nous est inconnu et peut donc entraîner la perte de données si cette action est faite après la commande GET PICTURE.
Le numéro de l'image est ensuite incrémenté dans le ***loop()*** afin d'ouvrir un nouveau fichier et stocker une nouvelle image.

5 Gestion de la cadence de prise des photos

5.1 Raisonnement et calculs

La prise et le stockage d'une photo sur la carte SD prennent un peu plus de deux secondes. La durée du vol prévue étant d'environ 3 heures, il a été décidé de cadencer la prise de photos à une toutes les 10 secondes, ce qui donnera quand même plus de 1000 photos à la fin du vol.
Pour gérer cette cadence, nous avons utilisé un des timers internes au microcontrôleur. Les informations qui vont suivre dans cette partie sont issues de la documentation du microcontrôleur ATmega2560 disponible sur le site www.alldatasheet.fr. Le microcontrôleur ATmega2560 possède six timers différents : deux de 8 bits et quatre de 16 bits. La méthode retenue est la comparaison entre la valeur du compteur à une valeur prédéfinie en amont. Il est nécessaire de diviser la fréquence du microcontrôleur qui est par défaut de 16 MHz. On utilise alors un prédiviseur de 1024 pour créer une horloge cadencée à 15625 Hz. La valeur avec laquelle sera comparé le compteur est calculée de la manière suivante :

$$compare = \frac{16MHz}{prescaler \times interruptfrequency} - 1 = \frac{16MHz}{1024 \times 0,1Hz} - 1 = 156249$$

On dispose cependant de timers 8 et 16 bits donc une telle valeur n'est pas atteignable sur nos compteurs. On choisit tout de même un timer 16 bits (le 3 qui n'est pas utilisé par d'autres fonctions de notre projet). L'idée est alors de réaliser plusieurs comparaisons avec une valeur inférieure à 65536 et d'attendre la fin de toutes ces comparaisons pour cadencer notre système à 0,1Hz.
Nous avons décidé de faire trois comparaisons : $\frac{156249}{3} = 52083$. On réalise donc trois comparaisons avec la valeur 52082 (52083 coups d'horloge car le compteur est initialisé à 0) avant de relancer le ***loop()***.

5.2 Initialisation des différents registres du timer et comparaison

Comme dit précédemment, on choisit le timer 3 qui est sur 16 bits. On a alors besoin d'initialiser un certain nombre de registres. Le premier registre à initialiser est le registre **TCCR3A** (figure 14).

Bit	7	6	5	4	3	2	1	0	
	COM3A1	COM3A0	COM3B1	COM3B0	COM3C1	COM3C0	WGM31	WGM30	TCCR3A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

FIGURE 14: Registre TCCR3A

Les bits 7 à 2 (COM3A1, COM3A0, COM3B1, COM3B0, COM3C1 et COM3C0) permettent de paramétrer des entrées et sorties pour la comparaison. Les bits 1 et 0 (WGM31 et WGM30) servent à sélectionner le type de comparaison. L'initialisation de ce registre se fait à 0 pour tous les bits. Le registre **TCCR3B** est lui aussi initialisé à 0 (figure 15).

Bit	7	6	5	4	3	2	1	0	
	ICNC3	ICES3	–	WGM33	WGM32	CS32	CS31	CS30	TCCR3B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

FIGURE 15: Registre TCCR3B

Les bits 4 et 3 (WGM33 et WGM32) participent également à la sélection du mode de comparaison avec les bits 1 et 0 de **TCCR3A**. Les bits 2 à 0 (CS32, CS31 et CS30) correspondent à la valeur du préscalaire.
Le compteur 16 bits est initialisé grâce au registre **TCNT3**. On initialise donc à la fois les 8 bits de poids forts et les 8 bits de poids faibles à 0.

Bit	7	6	5	4	3	2	1	0	
	TCNT3[15:8]								TCNT3H
	TCNT3[7:0]								TCNT3L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

FIGURE 16: Registre TCNT3

La valeur qui servira de comparaison avec le timer est stockée dans le registre **OCR3A**. Pour stocker la valeur 52082, il faut donc initialiser les bits de poids forts à 0xCB et les bits de poids faibles à 0x72.

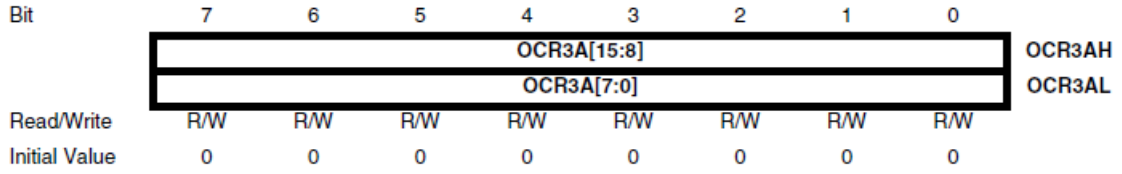


FIGURE 17: Registre OCR3A

On choisit ensuite la valeur du prédiviseur avec le tableau de la figure 18 :

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

FIGURE 18: Choix des valeurs pour les bits 2 à 0 du registre TCCR3B

Pour un prédiviseur de 1024, il faut donc passer à 1 les bits CS32 et CS30 du registre **TCCR3B**.
Le mode de comparaison choisi est le mode *Clear Timer on Compare* ou *CTC* qui réinitialise le compteur à 0 lorsque ce dernier atteint la valeur stockée dans **OCR3A**. La comparaison est ainsi périodique. Pour sélectionner ce mode de comparaison, il faut passer le bit WGMn2 (le bit 3 du registre **TCCR3B**) à 1.

Le registre **TIMSK3** comporte plusieurs *flags* qui passent à 1 lorsque le compteur atteint la valeur stockée dans un des registres **OCR**. Le bit qui nous intéresse ici est donc le bit 1 qui passera à 1 lorsque le compteur atteindra la valeur 52082.

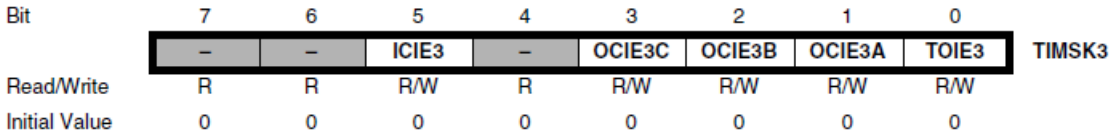


FIGURE 19: Registre TIMSK3

La figure 20 montre l'initialisation de ces différents registres dans le code.

```
void timer_init()
{
  cli();
  TCCR3A = 0x00;
  TCCR3B = 0x00; //initialize register
  TCNT3H = 0x00; //initialize counter
  TCNT3L = 0x00; //initialize counter
  OCR3AH = 0xCB; //setting high byte for compare match register
  OCR3AL = 0x72; //setting low byte for compare match register
  TCCR3B = TCCR3B | 0x05; //setting CS32 to CS30 for 1024 prescaler
  TCCR3B = TCCR3B | 0x08; //turn on CTC mode
  TIMSK3 = 0x00;
  sei();
}
```

FIGURE 20: Initialisation des registres

Dans le ***loop()*** on exécute l'extrait de code à la figure 21 qui met le système en attente tant que trois comparaisons n'ont pas été effectuées. En effet, tant que le flag du registre **TIMSK3** n'est pas à 1, il ne se passe rien, puis on redescend le flag à 0 lorsque la valeur courante du compteur atteint 52082.
Cela permet alors de cadencer notre système à une photo toutes les 10 secondes. Pour que cette cadence soit respectée, il faut cependant que la capture et le stockage d'une image soient effectués dans un délai inférieur à 10 secondes. J'ai donc mesuré le temps nécessaire à la capture et le stockage d'une photo en laissant tourner le système sans restriction. Les photos obtenues sont alors espacées d'un peu plus de deux secondes, ce qui est compatible avec notre cadence souhaitée d'une photo toutes les 10 secondes.

```
for(int i=0 ; i<3 ; i++)
{
  while((TIFR3 & 0x02) == 0){}
  TIFR3 |= 0x02;
}
```

FIGURE 21: Gestion de l'attente de 10 secondes entre chaque photo

6 Ajout d'une deuxième caméra

Afin d'obtenir à la fois des photos de l'horizon et des photos à la verticale de la nacelle lors du vol du ballon, nous avons décidé de rajouter une deuxième caméra. Cette deuxième caméra est donc connectée à la carte Arduino via le port **Serial2**. Pour y arriver, il a fallu passer le port série en argument de chacune des fonctions afin de simplifier leurs appels dans le ***setup()*** ou le ***loop()***.
Les fonctions prennent donc en argument un "*Stream *port*" qui est un pointeur vers un type *Stream* et l'utilisent comme dans l'extrait de code à la figure 22.

```
void cam_initiate(Stream *port)
{
  if (VERBOSE) Serial.print("Sending 'init' ... ");
  (*port).write((byte)0xAA);
  (*port).write((byte)0x01);
  (*port).write((byte)0x00);
  (*port).write((byte)0x07);
  (*port).write((byte)0x01);
  (*port).write((byte)0x07);
  if (ack_check(0x01, port))
  {
    if (VERBOSE) Serial.println("OK");
  }
  else
  {
    Serial.println("init Failed");
    while(1);
  }
}
```

FIGURE 22: Exemple du passage du port série en argument de la fonction ***cam_initiate()***

Cependant, la classe *Stream* ne possède pas de méthode **begin()**. Dans certains cas, il est donc nécessaire d'utiliser la classe *Hardware Serial*, qui est un héritage de la classe *Stream*, comme c'est le cas dans la fonction ***cam_max_baudrate()***.

On initialise alors chacune des deux caméras séparément dans le ***setup()***. Dans le ***loop()***, on ouvre deux fichiers images sur la carte SD (*pic.jpg* et *img.jpg*) puis on prend les photos simultanément sur les deux caméras. On récupère alors les données et on les écrit sur la carte SD de manière séparée. La figure 23 montre le ***loop()*** final du système.

```
void loop()
{
  digitalWrite(13, LOW);

  sprintf(pic_name1, "pic%d.jpg", pic_number1);
  img1 = SD.open(pic_name1, FILE_WRITE);

  sprintf(pic_name2, "img%d.jpg", pic_number2);
  img2 = SD.open(pic_name2, FILE_WRITE);

  snap(&Serial1);
  snap(&Serial2);
  delay(200);

  get_picture(&Serial1);
  long int pic_size1=data_check(&Serial1);
  writing_on_sd_card(img1, &Serial1, pic_number1, pic_size1);
  pic_number1++;

  get_picture(&Serial2);
  long int pic_size2=data_check(&Serial2);
  writing_on_sd_card(img2, &Serial2, pic_number2, pic_size2);
  pic_number2++;

  digitalWrite(13, HIGH);

  for(int i=0 ; i<3 ; i++)
  {
    while((TIFR3 & 0x02) == 0){}
    TIFR3 |= 0x02;
  }
}
```

FIGURE 23: Fonction ***loop()*** du système de prise et de stockage d'images

7 Alimentation du système

Afin de déterminer l'alimentation nécessaire à notre système, j'ai mesuré sa consommation. En fonctionnement, il ne dépasse pas les 200mA. Les piles qui ont été choisies pour leur résistance au vide et au froid peuvent délivrer jusqu'à 800mA. Bien que la broche d'alimentation de la carte précise une alimentation de 5V, la documentation de la carte Arduino Mega 2560 (https://www.robotshop.com/media/files/pdf/arduinomega2560datasheet.pdf) nous apprend que pour un fonctionnement optimal de cette carte, il faut l'alimenter avec une tension

comprise entre 7V et 12V. Il a donc été décidé d'alimenter l'Arduino avec six piles délivrant 1,5V chacune afin de rendre le système autonome pour toute la durée du vol. Un convertisseur de tension déjà intégré dans l'Arduino permet d'assurer les 5V en entrée nécessaires à l'alimentation. Après avoir réaliser près de 10 heures de tests (température ambiante, à froid ou dans le vide) sur le système sans changer les six piles constituant l'alimentation, nous pouvons alors valider l'autonomie du système.

8 Conclusion

Au moment où je rédige ce rapport, le ballon n'a toujours pas pu être lâché en raison de la crise de la COVID-19. Cependant, la nacelle est prête et tous les modules y ont été intégrés. Le système gérant la capture et le stockage des images que j'ai pu développer a été testé sous vide (dans une cloche à vide) ainsi qu'à -20 degrés Celsius. Les tests à température négatives ont été réalisés grâce à une étuve isolée dont on peut régler la température intérieure. L'étude dispose d'une fenêtre permettant au système de prendre en photo un chronomètre durant toute la durée du test (près de 3h), ce qui m'a permis d'assurer la validité du test.

Lors des derniers tests généraux de la nacelle, les caméras n'ont fourni aucune image après près de deux heures de test. Le problème a été identifié : il s'agit de la synchronisation des caméras qui ne fonctionne pas par moment. J'ai alors décidé de rajouter une LED qui clignote lorsque les caméras sont bien synchronisées. Cette même LED s'allume également à chaque fois que les deux photos (caméra verticale et horizontale) ont été prises et sauvegardées sur la carte SD. Ainsi, nous avons un témoin visuel du bon fonctionnement du système. Si la LED ne clignote pas lorsque l'on allumera tous les systèmes avant le lancement du ballon, il suffira de faire un reset de la carte Arduino jusqu'à ce que les caméras soient synchronisées. Il nous est tout de même impossible d'assurer le fonctionnement du système tout le long du vol, cette LED nous permet simplement d'être certain que le système est en route au moment du lâcher.

Ce projet aura été pour moi l'occasion d'enrichir mes connaissances de base sur le développement avec l'environnement Arduino. J'ai également pu mettre en application les connaissances sur les microcontôleurs acquises au cours de ma scolarité, notamment en terme de communication avec les caméras et gestion des timers. Je me suis aussi heurté à différents problèmes lors du développement, notamment des problèmes d'exécution de fonctions en temps réel. Cela m'a permis de prendre conscience que l'ordre dans lequel sont écrites les instructions dans le code peut s'avérer très important : par exemple l'ouverture du fichier sur la carte SD qui doit impérativement être fait avant la commande GET PICTURE.

Faire partie de cette équipe composée à la fois d'étudiants et d'enseignants-chercheurs aura également été une expérience extrêmement enrichissante. Chacun a pu apporter ses idées et sa vision du projet au cours des multiples réunions. J'ai ainsi pu me rendre compte de l'organisation que nécessite un projet d'une telle envergure et de l'importance de rédiger un cahier des charges en amont.