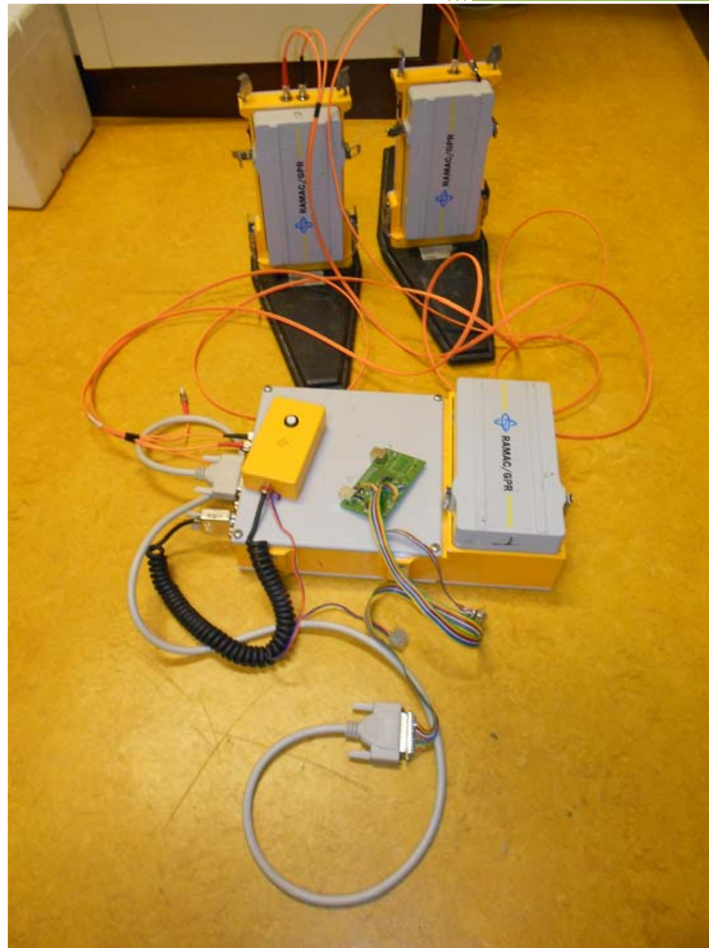
2011/2012

Acquisition automatique par RADAR de sol pour un suivi de l'évolution de la profondeur de nappe phréatique



VINOT Nicolas

BONNOT Aurélien

Responsable de projet

Jean-Michel FRIEDT

Table des matières

1 Remerciements 3

2 Introduction 3

3 Objectif du projet 4
3.1 Cahier des charges . . . 4
3.2 Diagramme de Gant . . . 4

4 Principes de fonctionnement 5

5 Compréhension du protocole 6
5.1 Introduction . . . 6
5.2 Sondage des signaux de communication . . . 6
5.2.1 Câblage . . . 6
5.2.2 Logiciels OLS et GTKWave . . . 6
5.3 Commandes . . . 8
5.3.1 Commande Reset_P . . . 8
5.3.2 Commande S_Sample . . . 9
5.3.3 Commande S_Freq . . . 9
5.3.4 Commande S_Sigpos . . . 9
5.3.5 Commande pour récupérer une trame . . . 10
5.4 Réponse de l'unité de contrôle . . . 11

6 Implémentation du protocole sur microcontrôleur 11
6.1 Introduction . . . 11
6.2 Explication du programme . . . 13
6.2.1 Fonction Write_command . . . 14
6.2.2 Fonction Read_command . . . 14
6.3 Problèmes rencontrés . . . 15

7 Stockage sur carte Secure Digital (SD) 15
7.1 Introduction . . . 15
7.2 Branchement . . . 15
7.3 Explication du programme . . . 15
7.3.1 Ouverture de la communication . . . 16
7.3.2 Stockage de la trame . . . 16
7.3.3 Fermeture de la communication . . . 16
7.4 Test . . . 16

8 Gestion de l'énergie 17
8.1 Introduction . . . 17
8.2 Mise en veille . . . 17
8.3 Réveil du microcontrôleur . . . 18
8.4 Stockage des paramètres sur la carte . . . 19

9 Interface graphique 20
9.1 Cahier des charges . . . 20
9.2 Programme . . . 20
9.3 Utilisation de l'interface graphique . . . 21

10 Nouveau RADAR : l'unité de contrôle CUII **22**
10.1 Introduction 22
10.2 Définition d'une trame 24
10.3 Envoi de la commande 24
10.4 Réponse du RADAR 25
10.5 Réglage des différents paramètres 26
10.6 Réception de la trame 26

11 Conclusion **29**

12 Annexes **29**
12.1 Tableau de commandes 29
12.2 Programme pour un fonctionnement autonome (stockage sur carte SD et veille) 30
12.3 Gestion de l'énergie 39
12.4 Programme Interface graphique 39

1 Remerciements

Nous tenons à remercier Jean-Michel Friedt pour sa disponibilité, le matériel fourni, et le temps qu'il nous a consacré pour le bon déroulement de ce projet, mais aussi toute l'équipe des étudiants en thèse et plus précisément Gwenhaël Goavec-Merou qui nous ont également apporté toutes leurs connaissances. Julien Garcia a écrit l'interface graphique en Qt permettant d'afficher les traces acquises.

2 Introduction

Le but de notre projet est la commande d'un RADAR. Celui-ci est composé d'une unité de contrôle reliée à 2 antennes : il s'agit d'un RADAR impulsionnel bistatique fonctionnant autour de fréquences allant de 50 à 200 MHz. Une antenne permet de générer des ondes électromagnétiques et la seconde de les réceptionner. Cet appareil est exploité pour détecter les structures sous-terraines, ou, dans le cas qui nous intéresse ici, de mesurer une épaisseur de glace.

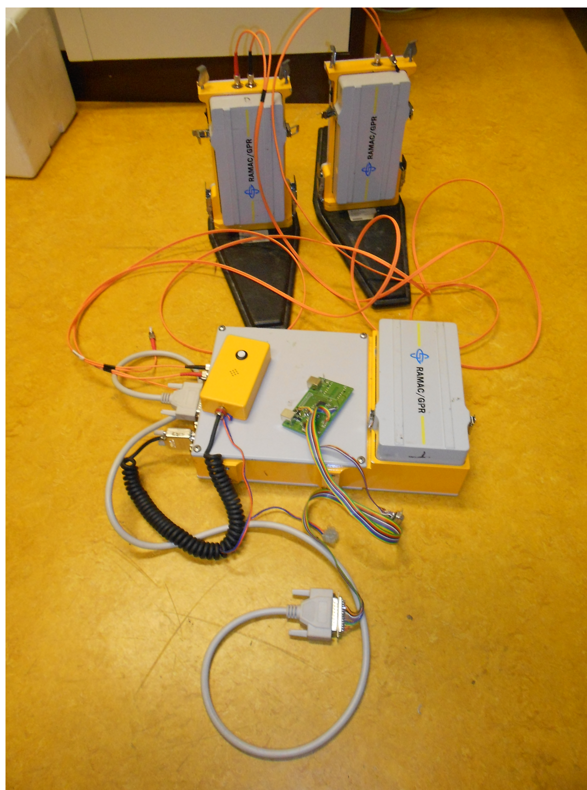


Figure 1 – RADAR Malå RAMAC de première génération, unité de contrôle CU.

Les unités de contrôle de RADAR à notre disposition sont relativement anciennes et exploitent des technologies considérées comme obsolètes sur les ordinateurs modernes : logiciel de contrôle sous DOS faisant un appel direct aux couches matériel d'un ordinateur et exploitation du port parallèle pour la communication. Nous nous sommes proposés, grâce à la compréhension ou l'identification du protocole de communication, de fournir une interface compatible avec la majorité des ordinateurs actuellement disponibles – port USB – et ce en vue soit d'une exploitation autonome du RADAR, soit sous le contrôle d'une interface graphique fonctionnant sous Unix.

3 Objectif du projet

3.1 Cahier des charges

Le RADAR de sol (*Ground Penetrating RADAR*, GPR) [1, 2] est un outil classique de caractérisation des structures souterraines en géophysique. Dans une configuration bistatique, un émetteur – généralement formé d'une antenne dipôle alimentée par une impulsion radiofréquence de l'ordre de ±350 V – génère un signal électromagnétique se propageant dans le sol. Toute rupture d'impédance électrique (variation de permittivité ou de conductivité du sol) se traduit par la réflexion aux interfaces d'une fraction de l'énergie électromagnétique. Le récepteur – lui aussi formé d'une antenne dipôle alimentant un amplificateur radiofréquence faible bruit – met en forme le signal reçu du sol, le numérise et envoie l'information à l'unité de contrôle. La bande de fréquences exploitée va de 25 à plus de 1000 MHz, avec le choix d'une fréquence d'autant plus basse que la profondeur de pénétration de l'onde électromagnétique dans le sol doit être importante, au détriment de la résolution puisqu'une fréquence plus basse se traduit par une longueur d'onde plus importante.

Un GPR est classiquement exploité pour une mesure ponctuelle de la configuration du sous-sol [3]. Une extension de ce mode de travail est un suivi à long terme de propriétés du sous-sol susceptibles d'évoluer dans le temps : pour l'application qui nous intéresse en environnement polaire, la position du toit du permafrost et l'épaisseur de la couche active sont deux grandeurs que nous désirons suivre sur une durée d'une année hydrologique (avril-novembre). Un GPR n'est pas un outil approprié pour une telle application : sans mode veille, sa consommation électrique importante (1,35 A sous 7 V, ou 10 W, pour l'unité de contrôle) ne fournit, lors d'une alimentation sur batterie, qu'une autonomie de quelques heures. Par ailleurs, commandé par un logiciel propriétaire fonctionnant sur un ordinateur exécutant MS-DOS ou une version inférieure à 98 de MS-Windows, son utilisation en autonomie pour un réveil périodique n'est pas possible.

L'objectif de ce travail consiste donc en

1. maîtriser le protocole de communication entre le GPR et l'ordinateur : effectué au travers du port parallèle d'un PC, ce protocole est identifié en partie au moyen d'une documentation partielle fournie par le constructeur, et en partie par une écoute des transactions,
2. porter ce protocole de communication à un microcontrôleur émulant le port parallèle par ses GPIO,
3. stocker l'information obtenue du RADAR sur une carte SD pour exploitation ultérieure, et notamment par un utilisateur n'ayant pas une expertise avancée des systèmes embarqués. Nous verrons que la compatibilité du format de stockage avec les ordinateurs personnels modernes implique l'utilisation du format FAT,
4. gestion de l'énergie du microcontrôleur et de l'alimentation du RADAR pour permettre une opération pendant au moins 1 an sur batterie.

Un résultat annexe à ces travaux est de fournir une interface souple – convertissant l'interface parallèle du RADAR en liaison série sur port USB – compatible avec les ordinateurs modernes. Une interface graphique sur ordinateur personnel permet alors de visualiser les trames acquises, fournissant donc une interface libre de commande du GPR.

3.2 Diagramme de Gant

Dans le cadre d'un projet hébergé par un laboratoire de recherche, les objectifs s'ajustent en fonction des ambitions et des résultats obtenus. Initialement, il avait été envisagé de comparer la vitesse de développement d'un logiciel de contrôle du RADAR en vue d'un fonctionnement autonome en C et exploitant une bibliothèque de gestion du stockage formaté sur carte SD d'une part, et sous environnement exécutif TinyOS d'autre part. Compte tenu du temps nécessaire à prendre en main TinyOS avec, en particulier, la maîtrise d'un nouveau langage s'apparentant à la programmation objet (NesC), et des succès rencontrés pour maîtriser le protocole de communication entre le PC et le GPR d'autre part, les orientations ont été revues au cours du projet pour s'ajuster au mieux au délai court de ce projet et en extraire le maximum de résultats exploitables à long terme par l'équipe d'accueil.

Les diagrammes des Fig. 1 et 3 présentent le déroulement de notre projet. Les tâches en bleu représentent les travaux qui devaient être effectuées en commun, les tâches vertes sont celle que M.Vinot devait réaliser, et pour finir les tâches rouges sont celles que devait réaliser M.Bonnot.

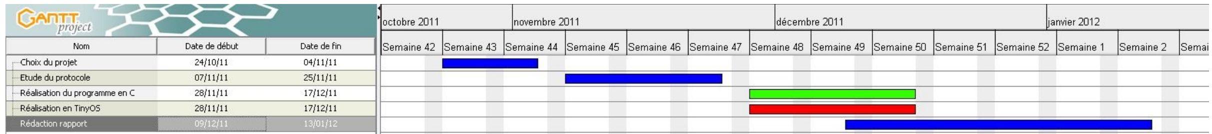


FIGURE 2 – Estimation des tâches

Les objectifs et les tâches à réaliser ont été modifiés en cour de projet (Fig. 3). La rédaction s'est déroulée tout au long du projet du fait que nous faisions un rapport régulier de l'avancée de nos travaux à notre responsable.

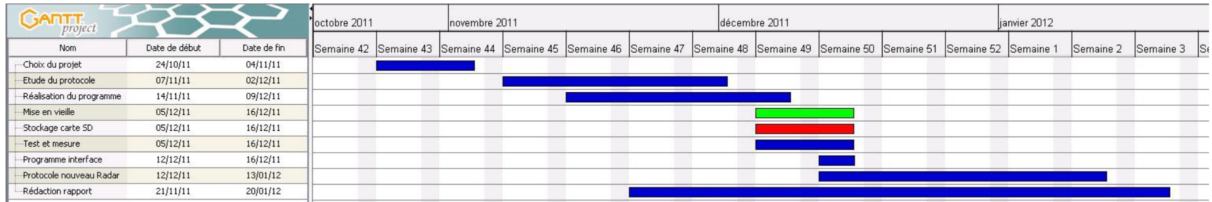


FIGURE 3 – Déroulement du projet

4 Principes de fonctionnement

Un GPR se résume en un générateur d'impulsions radiofréquences et d'un échantilloneur du signal radiofréquence reçu apès amplification. La durée de l'impulsion émise – inversement proportionnelle à la fréquence annoncée de fonctionnement – est déterminée par l'impédance de l'antenne [1], et varie donc en fonction de la permittivité du milieu sondé. Tout comme un microscope à sonde locale qui balaie une unique sonde sur la surface à analyser, le GPR est déplacé en divers points de la zone à sonder et une mesure prise toutes les 0,5 ou 1 s.

Plusieurs paramètres définissent les conditions de fonctionnement du GPR, et en particulier

- le taux de répétition des impulsions émises (commandées par l'utilisateur sur un critère d'intervalle de temps ou de distance parcourue),
- la fréquence d'échantillonnage f_e du signal reçu,
- le nombre d'échantillons N acquis pour chaque trace issue de l'émission d'une impulsion électromagnétique,
- le retard entre l'émission de l'impulsion électromagnétique (qui sature temporairement le récepteur par l'onde transmise dans l'air) et le début des mesures,
- le nombre de moyennes accumulées par le GPR (*stacks*) avant de renvoyer l'information à l'utilisateur.

Les second et troisième paramètres sont fondamentaux car du fait de la mémoire limitée embarquée par le GPR – il semblerait qu'il puisse stocker au plus 8192 valeurs de 16 bits – la durée de la mesure est $N \times f_e$. La durée de cette mesure détermine la profondeur des structures observées par le GPR : alors qu'en milieu tempéré l'eau imbibant le sol limite la profondeur de mesure à une dizaine de mètres au plus, un milieu sec tel qu'un glacier polaire permet une mesure d'interfaces situées à plus de 150 m de la surface. La durée de propagation d'une onde électromagnétique se propageant dans un milieu de permittivité ε_r vers une interface située à une distance d est $d\sqrt{\varepsilon_r}/c$ avec $c = 3.10^8$ m/s la célérité d'une onde électromagnétique dans le vide. Pour ε_r de l'ordre de 5 à 25 selon les milieux (`http://www.geo-sense.com/GPRmore.htm`) une interface située à 10 mètres est observée après un retard de 150 à 300 ns, tandis qu'une interface à 150 m est observée pour des retards de 2,2 à 5 μs. Étant donné qu'il est de bon goût de sélectionner une fréquence d'échantillonnage f_e de 5 à 10 fois la fréquence centrale de fonctionnement d'une antenne, nous constatons que le choix de N et f_e est un compromis entre mémoire disponible, temps de transfert des informations du GPR au microcontrôleur, et résolution de la mesure.

1. pp. 14 et 15 de `http://jmfriedt.free.fr/slides_ieeesensors2011.pdf`

À chaque émission du signal de déclenchement d'une impulsion par le module *transmit* (fibre optique T), ce même signal est émis vers un signal de déclenchement sur le module de réception (fibre optique D). Le module de réception est par ailleurs équipé d'une transmission par une seconde fibre optique des niveaux radiofréquences reçus numérisés (signal nommé R). En liant T et D, nous pouvons exploiter l'unité de contrôle du RADAR sans nécessiter les antennes et leur alimentation associée.

5 Compréhension du protocole

5.1 Introduction

Pour comprendre le fonctionnement du RADAR, nous avons utilisé le logiciel constructeur fourni par Malå. Nous avons pu voir les paramètres modifiables pour définir les conditions de fonctionnement (fréquence d'échantillonnage, nombre d'échantillons, nombre de moyennes accumulées avant de communiquer la mesure à l'utilisateur, origine temporelle des mesures).

Nous avons ensuite étudié un programme réalisé par notre responsable à partir d'un *sniffage* de ce logiciel, qui faisait fonctionner le RADAR en mode simulation. Ce programme visait à exploiter un composant convertissant une interface USB vers un port numérique générique (GPIO) susceptible d'émuler un port parallèle au moyen du composant FT2232. Cependant, la définition bit à bit de l'état du port au travers d'une liaison USB est une opération lente, et il s'est avéré que le RADAR atteint une limite de temps de communication (*timeout*) lors de la réception des messages les plus longs. Cette approche n'est donc pas viable : une interface autonome, capable de gérer la communication avec le RADAR avant de transmettre le résultat de la mesure au PC, permettra de résoudre ces problèmes de latence, au détriment d'une complexité logicielle et matérielle accrue puisqu'un microcontrôleur est en charge de ces opérations de communication.

En comparant le programme avec la documentation constructeur, nous avons relevé des incohérences : il semble que cette documentation soit générique à plusieurs RADARs de la marque. Nous avons donc décidé de "sniffer" à nouveau les signaux entre le RADAR et le logiciel d'origine.

5.2 Sondage des signaux de communication

La carte nous permettant de sonder les communications numériques est une Open Bench Logic Sniffer (Fig. 4), développé lors d'une collaboration entre Dangerous Prototypes et Gadget Factory [2], qui fonctionne avec un FPGA et un microcontrôleur PIC. La carte Logic Sniffer [3] peut relever jusqu'à 16 signaux, mais le taux d'échantillonnage et la profondeur mémoire sont réduits. Dans notre cas, ce problème ne se pose pas car nous ne relevons que 11 signaux. Le logiciel d'acquisition des données associées à ce matériel est OLS [4], un client Java fonctionnant sur de multiples plateformes dont GNU/Linux.

5.2.1 Câblage

Nous avons observé que dans la documentation constructeur, les broches INIT, STROBE et SCLTIN permettent de faire fonctionner le RADAR : nous avons donc décidé d'observer ces trois paramètres ainsi que les 8 bits de données. Pour cela, nous devons connaître le brochage du port parallèle (Fig. 5).

5.2.2 Logiciels OLS et GTKWave

OLS acquiert les signaux numériques qui sont ensuite analysés au moyen de GTKWave, capable de regrouper les signaux individuels en bus. Il faut définir au logiciel OLS (Fig. 6) :
- le port sur lequel est branchée le carte permettant de sonder les signaux,
- la fréquence d'échantillonnage : plus elle est élevée et plus la résolution temporelle est bonne mais moins la durée d'acquisition est longue,
- la source de déclenchement (*trigger*) : dans notre cas la broche reliée à *strobe*,
- choisir les noms des voies pour plus de clareté par la suite.

2. `gadgetfactory.net/logicsniffer`
3. `dangerousprototypes.com/docs/Open_Bench_Logic_Sniffer`
4. `ols.lxtreme.nl`

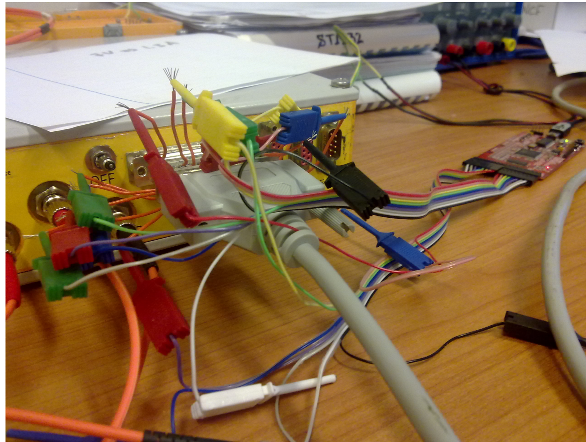


FIGURE 4 – Montage de sondage des signaux de communication sur le port parallèle du PC.

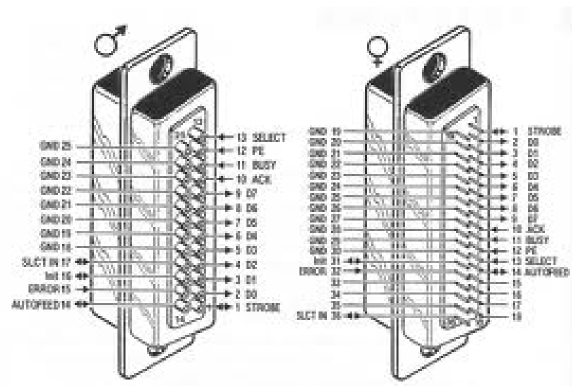


FIGURE 5 – Brochage port parallèle

Pour pouvoir exploiter les donnée des relevés au moyen de GTKWave, il faut les exporter au format "vcd".

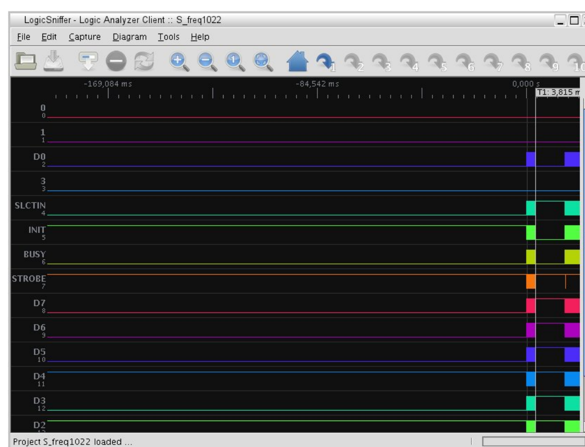


FIGURE 6 – Logiciel OLS

Le logiciel GTKWave [5] permet d'interpréter les signaux acquis (Fig. 7). Il faut regrouper les signaux de données en un bus pour qu'ils soient plus facilement interprétables puisque faisant partie d'un mot comprenant plusieurs bits dans un ordre connu. Nous demandons également que les mots (contenu du bus) soient affichés en décimal pour pouvoir faire plus facilement le lien avec les valeurs de commandes de la documentation constructeur. Ce logiciel permet également de zoomer sur les données pour les analyser : il est donc aisé de voir les signaux qui évoluent en même temps.

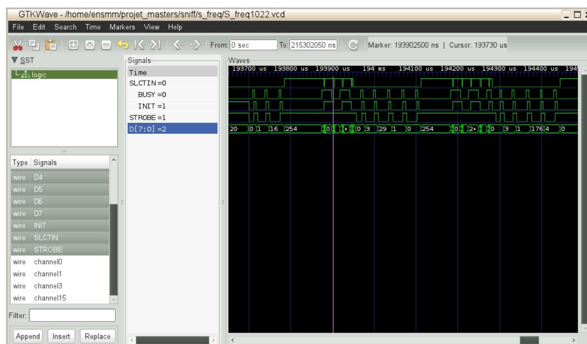


Figure 7 – Logiciel GTKWave

5.3 Commandes

Les commandes pour piloter le RADAR sont composées d'un nombre variable de caractères, en fonction du type de commande. Les trois premiers caractères se retrouvent pour toutes les commandes :

1. le premier caractère à envoyer est un 0 : il s'agit de l'octet de poids fort d'un mot de 16 bits explicitant le nombre d'informations transmises dans la trame,
2. nous envoyons ensuite le nombre de caractères restant jusqu'à la fin de la commande (octet de poids faible du mot explicitant le nombre d'informations transmises dans la trame),
3. le troisième caractère est le numéro de la commande, qui est soit issu de la documentation du constructeur (section. 12.1) ou de nos propres investigations du protocole de communication.

Les deux derniers caractères représentent les octets de poids faible et fort d'un mot contenant les arguments de la commande transmis au RADAR (fréquence, position de courbe,...). Ces deux caractères ne sont utilisés que par les commandes où il faut définir des paramètres.

Dans cette partie, nous ne définissons que les commandes essentielles : les autres commandes utilisées seront présentées en annexe.

5.3.1 Commande Reset_P

Pour paramétrer l'unité de contrôle du RADAR, il faut envoyer en premier une commande Reset après la mise sous tension. Cette commande de réinitialisation permet de définir le protocole de communication et les paramètres de connexion, notamment l'exploitation unidirectionnelle (SPP) ou bidirectionnelle (ECP) des 8 bits de données du port parallèle.

Il existe 3 types de commande Reset :

- Reset_C : initie les paramètres de connexion et définit le protocole de communication pour Centronics (port respectant la norme initiale avec un bus de données unidirectionnel communiquant de l'ordinateur vers le RADAR, la liaison du RADAR vers l'ordinateur nécessitant l'exploitation des signaux de status du port),
- Reset_P : initialise les paramètres de connexion et configure le RADAR pour une communication bi-directionnelle du bus de données du port parallèle,
- Reset_I : définit les paramètres de connexion et configure le RADAR pour une communication série asynchrone (RS232).

5. `gtkwave.sourceforge.net`

Dans notre cas, nous utilisons un Reset_P, afin de réaliser une communication parallèle bi-directionnelle en vue de maximiser le débit de communication. Toutes les transactions portent sur des mots de 16 bits et nécessiteront donc deux séquences d'échanges d'informations sur le port parallèle (8 bits). Dans un port parallèle classique, le bus de données ne communique que du PC vers l'imprimante, et ne permet pas de recevoir des informations du GPR : il faut pour cela exploiter les 5 bits de status, et par conséquent communiquer par quartet (deux fois plus lent).

Lors des différentes manipulations sur le programme de test, nous avons constaté que l'instruction `Reset_P` ne se réalisait pas. La communication étant mal initialisée, la plupart des commandes suivantes ne fonctionnaient pas. Nous avons donc décidé de répéter cette étape importante, qui a permis d'initialiser correctement la liaison, et ainsi de supprimer un nombre d'erreurs important sur les commandes envoyées dans la suite du programme.

Pour cette commande il faut envoyer 3 paramètres (0,1,16) : le numéro 16 est le numéro de commande indiqué dans la documentation constructeur et qui est celle comprise par le radar pour `Reset_P`.

5.3.2 Commande S_Sample

Cette commande permet de définir le nombre d'échantillons que l'on veut mesurer. La valeur de sa commande est 1. Nous définissons ensuite 2 paramètres pour choisir le nombre d'échantillons. Le relevé des différents sondages du protocole nous a permis de trouver une équation permettant de relier les deux paramètres (bh octet de poids fort et bl octet de poids faible). Le tableau est présenté en Fig. 8.

Nombre d'échantillons	BL	BH	Résultat
500	244	1	500
800	32	3	800
1000	232	3	1000
1200	176	4	1200
1500	220	5	1500
2000	208	7	2000
4000	160	15	4000

Figure 8 – Tableau de relevés du nombre d'échantillons.

La relation entre le nombre d'échantillons et les différents paramètres est :

$$N = BH \times 256 + BL$$

Exemple : pour un nombre d'échantillons de 1300, il faut un BH de 5 et un BL de 20.

5.3.3 Commande S_Freq

Cette commande permet de définir la fréquence d'échantillonnage des signaux acquis par le RADAR. Pour trouver les différentes valeurs des arguments à cette commande, nous avons sondé les paramètres envoyés par le logiciel d'origine à des valeurs de fréquences choisies. Nous avons relevé le tableau de la Fig. 9.

La relation est la suivante :

$$fe = 30666/BL$$

5.3.4 Commande S_Sigpos

La commande `S_Sigpos` permet de définir l'origine de la mesure. Cette commande est composée de cinq caractères. Le premier caractère à envoyer est un 0, suivi du nombre de caractères restant (ici 3). Le troisième caractère est le numéro de la commande fourni par la documentation du constructeur (3 pour SIGPOS) et les deux derniers caractères représentent les octets de poids faible et fort d'un mot de 16 bits permettant de modifier la position de la courbe.

Nous avons observé, en comparant une courbe acquise avec la valeur par défaut (inconnue mais positive) de SIGPOS (Fig. 10, courbe bleue) et pour SIGPOS=0 (Fig. 10, courbe rouge), que incrémenter SIGPOS décale la courbe vers la gauche sur l'axe des temps. Incrémenter SIGPOS décale donc la courbe

Fréquence d'échantillonnage (MHz)	BL	BH
502,73	61	0
601,31	51	0
807,02	38	0
901,96	34	0
1022,23	30	0
1533,34	20	0
2044,45	15	0

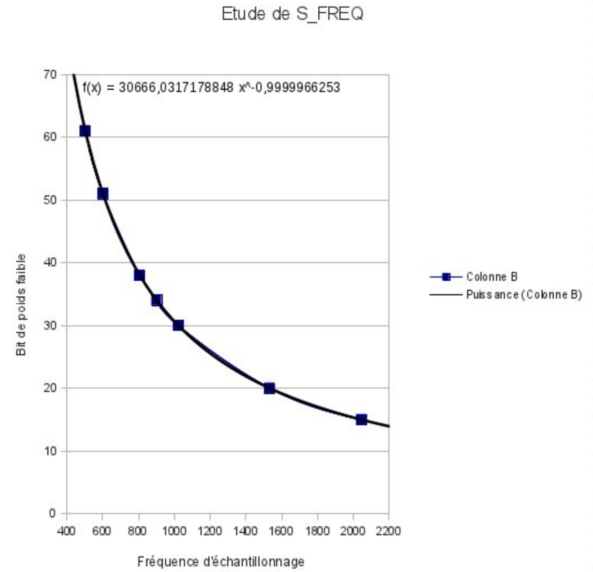


Figure 9 – Tableau de relevés du nombre d'échantillons, et tracé du relevé de fréquences en vue d'identifier graphiquement la relation entre mot transmis et fréquence d'échantillonnage.

vers la droite. L'octet de poids faible de l'argument de SIGPOS ne semble pas avoir d'effet visible sur la position de la courbe : seul l'octet de poids fort est significatif. Une valeur de SIGPOS négative n'a pas encore été testée : nous ne savons pas s'il s'agit d'un argument signé ou non.

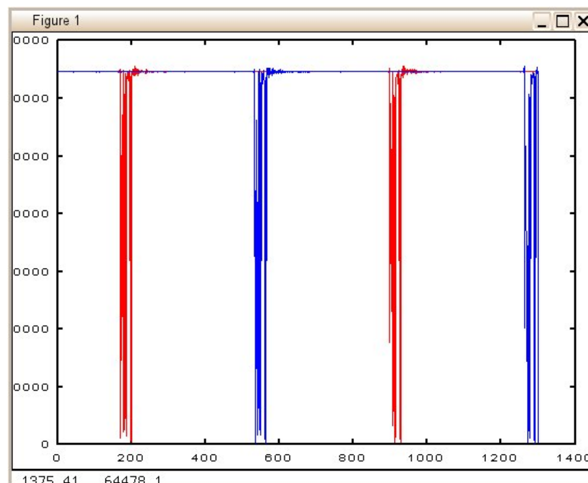


Figure 10 – Relevé des positions

5.3.5 Commande pour récupérer une trame

Afin de récupérer une trame, nous devons envoyer plusieurs commandes :

- la première est START qui permet de passer l'unité de contrôle du mode veille au mode acquisition. Le RADAR attend ensuite la commande TRIG pour déclencher une acquisition,
- la seconde commande est TRIG qui déclenche l'acquisition et stocke les valeurs dans le tampon,
- la commande suivante est GMT qui renvoie les valeurs du tampon,
- finalement, la dernière commande de la séquence est STOP qui arrête l'acquisition.

5.4 Réponse de l'unité de contrôle

Lorsque nous envoyons une commande à l'unité de contrôle, celle-ci acquitte une trame (Fig. 11) composée d'une chaîne de plusieurs caractères :

- le premier caractère qui est renvoyé a une valeur nulle,
- le second caractère correspond au nombre d'octets à envoyer (généralement deux),
- le troisième octet correspond au numéro de la commande qu'on vient d'envoyer,
- le dernier octet est l'octet d'acquittement (ACK). L'acquittement correspond à une valeur décimale de 254 (documentation constructeur) soit 0xFE en hexadécimal.

Lorsque l'unité de contrôle ne comprend pas la trame de commande transmise, une trame de non acquittement est communiquée. Elle est composée de plusieurs éléments :

- le premier caractère qui est renvoyé est une valeur nulle,
- le second caractère qui nous est envoyé est un trois,
- le troisième octet est l'octet de non acquittement (NACK), qui correspond à une valeur décimale de 255 (documentation constructeur) soit 0xFF en hexadécimal,
- l'octet suivant est le type d'erreur qui s'est produit. Il existe trois erreurs possibles (0x82 pour une mauvaise implémentation, 0x85 pour un timeout, et 0x86 pour une commande invalide).

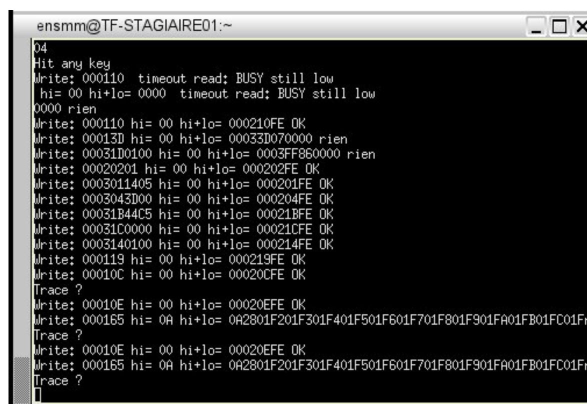


Figure 11 – Envoi et réception des trames de commande

6 Implémentation du protocole sur microcontrôleur

6.1 Introduction

Les unités de contrôle des anciens modèles de GPR Malå exploitent une liaison sur le port parallèle programmé comme GPIO. Ce bus de communication n'est plus disponible sur les ordinateurs récents. Une première tentative de mise en œuvre de ces RADARs a consisté en l'utilisation de modules de conversion USB-Centronic. Il semblerait que soit le matériel, soit les *drivers* fournis pour GNU/Linux de ces interfaces, ne permettent pas une programmation bit par bit en dehors du contexte de communication avec une imprimante. Une seconde approche a consisté en l'utilisation d'un convertisseur USB-GPIO du type FT2232 : dans le cas le plus simple, chaque nouvel état de la sortie GPIO nécessite une communication sur bus USB, opération lente lorsqu'elle n'exploite pas pleinement les fonctionnalités de communication par bloc sur ce bus. La conséquence est un délai trop long (*timeout*) lors de la réception des trames de mesures du RADAR. Une stratégie plus pertinente de cette approche serait probablement de type DMA avec pré-programmation des séquences de manipulation des bits du GPIO. Cependant, cette approche est complexe et ne répondait pas à une exigence d'autonomie à long terme lors d'un fonctionnement en suivi temporel de l'évolution des propriétés du sous-sol. Nous avons donc choisi de développer une approche alternative consistant en l'exploitation d'un microcontrôleur dédié chargé de la communication avec le RADAR et la réception des trames en vue de leur stockage ou communication à un ordinateur de contrôle (Figs. 12 et 13).

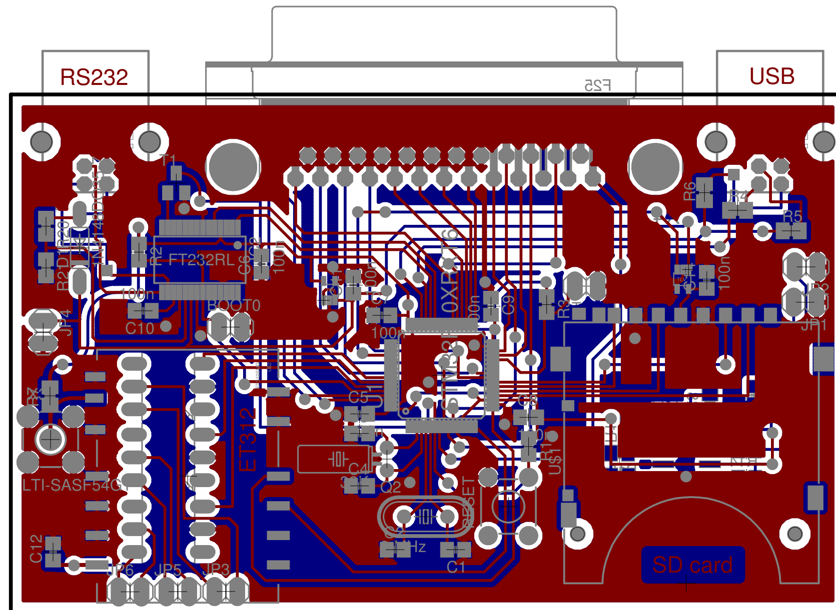


FIGURE 12 – Placement des composants autour d'un microcontrôleur STM32 chargé d'émuler un port parallèle (connecte DB25) et stocker des informations sur carte SD. La programmation et communication se font par un port série virtuel créé sur bus USB au moyen d'un FTDI FT232, mais à terme le support USB natif du STM32 sera exploité.

Le département temps-fréquence de FEMTO-ST a décidé de passer à l'utilisation d'un microcontrôleur basé sur un cœur ARM Cortex-M3 depuis l'été 2010, et en particulier sur la gamme STM32 de ST Microelectronics. Cette série de microcontrôleurs décline un cœur commun avec une vaste gamme de périphériques, allant de versions faible coût n'incorporant que peu de mémoire volatile (RAM) et non-volatile, aux versions haut de gamme proposant des interfaces ethernet et USB. Dans le cas qui nous intéresse, nous ne faisons appel qu'aux fonctionnalités d'entrées-sorties numériques généralistes (GPIO) ainsi qu'aux capacités de mise en veille pour une gestion efficace de l'énergie lors d'un fonctionnement autonome sur pile.

Historiquement, lorsque le département temps-fréquence a commencé à exploiter ce microcontrôleur, seule une bibliothèque ouverte mais propriétaire de ST était accessible pour fournir un niveau d'abstraction lors de l'accès aux périphériques : `libstm32`. Une bibliothèque libre, `libopenstm32`, n'était alors qu'à l'état d'embryon et ne fournissait que peu de fonctionnalités. Cependant, la bibliothèque `libstm32` évolue constamment pour tenir compte des évolutions de la gamme des microcontrôleurs, et la dépendance de nos développements sur une bibliothèque propriétaire qui peut devenir fermée à tout moment n'est pas satisfaisante.

Nous nous sommes par conséquent imposés d'utiliser la nouvelle bibliothèque `libopencm3`[6], une implémentation libre d'une bibliothèque portable sur plusieurs architectures autour du cœur Cortex-M3 qui commence à se stabiliser, sucesseur de `libopenstm32`. Durant notre projet, nous avons donc été confrontés aux problèmes de portage des codes écrits avec `libstm32` vers `libopencm3`. Pour certaines fonctions qui n'existaient pas, il nous a fallu les réécrire en étudiant les opérations qu'elles réalisaient.

La chaîne de compilation croisée, basée sur le compilateur `gcc`, est compilée au moyen du script[7] `summon-arm-toolchain` chargé de récuperer l'ensemble des outils nécessaires, compiler dans le bon ordre chaque élément, et les installer dans les répertoires appropriés. À l'issue de cette compilation, l'outil `arm-none-eabi-gcc` permettra de générer des binaires à destination du processeur ARM Cortex-M3.

6. `http://sourceforge.net/projects/libopencm3/`
7. `github.com/esden/summon-arm-toolchain`

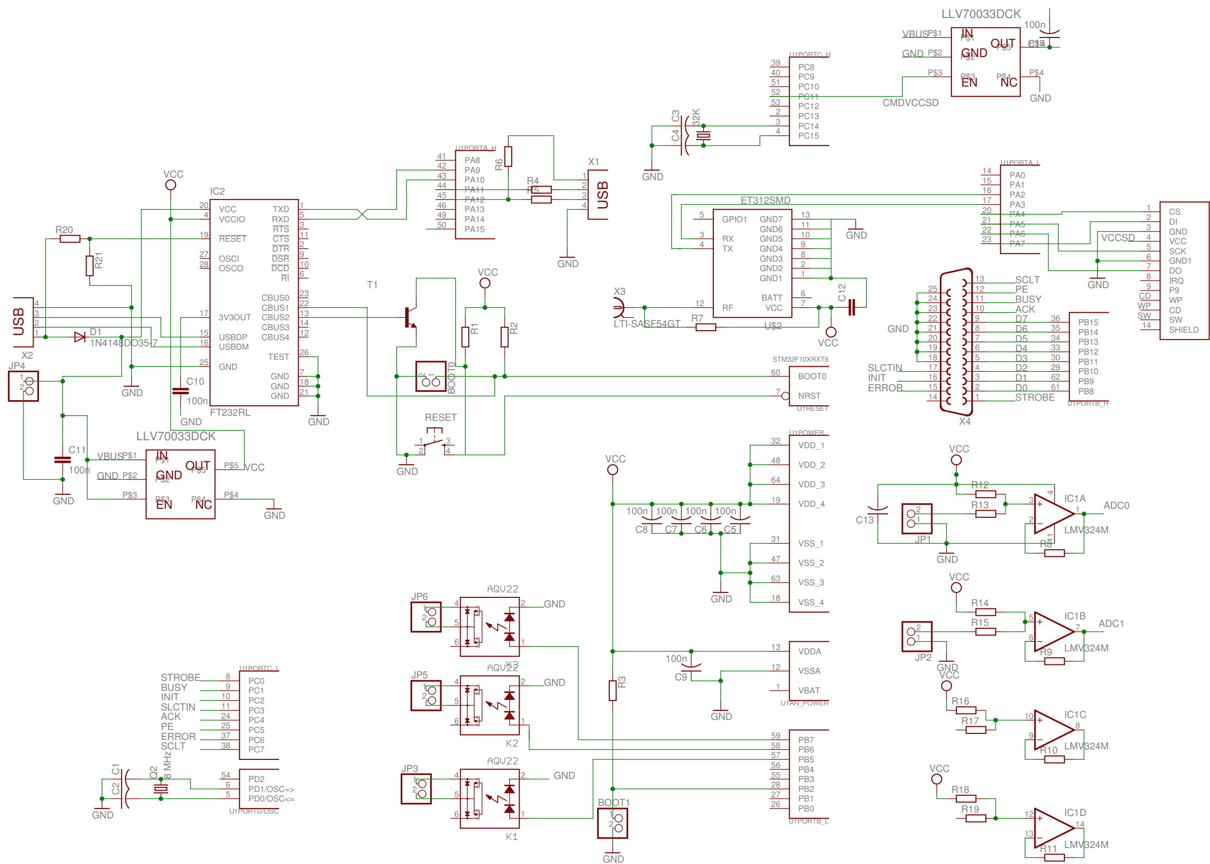


Figure 13 – Schéma du circuit interfaçant un microcontrôleur STM32 au RADAR : les broches PB8 à PB15 du STM32 correspondent au bus de données D0-D7 du port parallèle (broches 2 à 9), PC0 à STROBE (broche), PC1 à BUSY (broche 11), PC2 à INIT (broche 16), PC3 à SLCTIN (broche 17), PC4 à ACK (broche 10), PC5 à PE (broche 12), PC6 à ERROR (broche 15) et PC7 à SCLT (broche 13). Noter que seuls deux quartz (32 kHz et 8 MHz optionnel) et des condensateurs de découplage sont nécessaires pour le fonctionnement du microcontrôleur.

6.2 Explication du programme

Le programme principal se déroule en plusieurs sous-programmes. Nous allons décrire son fonctionnement en suivant séquentiellement le programme principal :

– Nous appellons, pour commencer, la fonction `clock_setup()`. Nous activons tout d'abord l'horloge du système en la définissant comme exploitant le quartz haute fréquence à 8 MHz pour cadence le processeur à 72 MHz par multiplication par PLL, puis nous activons les horloges des différents registres et ports que nous utilisons : cela est obligatoire avec ce microcontrôleur.
– Nous utilisons la fonction `gpio_setup()` qui permet de définir en entrée/sortie les ports du microcontrôleur que l'on utilise,
– puis nous activons et définissons les paramètres de la liaison USART (liaison par un port de communication virtuel sur port USB), par la fonction `USART_setup()`.
– Nous définissons ensuite les états initiaux des bit de contrôle `init` (va changer d'état tous les 1.57 ms) et `strobe` (l'état haut). Ces états ont été trouvés en sondant l'état des bits de contrôle.
– Ensuite, nous lisons la valeur des données du bus, et la comparons au bit `busy`, afin d'identifier si l'unité de contrôle du RADAR est occupée.
– Nous définissons un tableau `cmd`, comprenant les différentes commandes que nous allons utiliser,
– puis nous passons les commandes au RADAR à l'aide d'une fonction `write()` que nous expliquerons par la suite.
– Puis nous lisons se que nous renvoie le radar à l'aide de la fonction `read()` (voir partie suivante),

– Pour la dernière fonction `read()`, nous lisons la longueurs de la trame et la comparons a une valeur globale. Cela permet de vérifier et d'adapter la taille du tampon en fonction de la longueur de la trame, pour que lors du prochain relevé, l'ordinateur de contrôle ne remplisse pas à moitié le tableau contenant la trame de données ou interprète le reste d'une trame qui a été mal stockée.
– Pour finir, nous remplissons le tableau avec une boucle qui a comme valeur maximum l'indice trouvé précédemment.

6.2.1 Fonction Write_command

Dans la fonction `write()`, nous envoyons les paramètres au RADAR pour le configurer. Pour cela il faut respecter un ordre bien précis entre les bits de commande et les bits de données (Fig. 14).

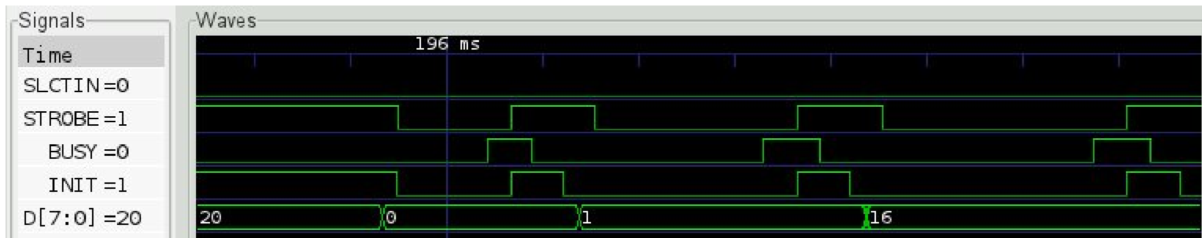


FIGURE 14 – Trame d'envoie de commande

– Nous envoyons tout d'abord la première valeur commande contenu dans le tableau `cmd`,
– puis passons le bit `strobe` à l'état bas.
– Le bit `busy` passe à l'état bas, pour indiquer que le RADAR n'est pas occupé.
– Ensuite, il faut repasser le bit `strobe` et `init` à l'état haut,
– enfin, le bit `busy` repasse à l'état haut, pour indiquer que le RADAR est occupé.
– Pour finir, nous mettons le bit `init` à l'état bas et incrémentons l'indice du tableau pour passer à la commande suivante.

Cette fonction va tourner tant que le RADAR n'a pas reçu la totalité du tableau correspondant à la commande en cours.

6.2.2 Fonction Read_command

Cette fonction permet de détecter si la commande envoyée au RADAR a bien été prise en compte.

Avant tout chose, dans cette partie il faut définir les bits de données en entrée pour pouvoir interpréter la réponse du RADAR. Le bit de donnée `SELECTIN` doit être placé à l'état haut. Puis comme pour la fonction `write()`, il faut respecter un ordre bien précis entre les bits de commande et la réception des bits de donnée provenant du RADAR.

Le RADAR nous renvoie deux types de trames (Fig. 15) :
– Une trame d'aquittement de la commande envoyée. Elle est composée de trois bits, 0, 2, la commande et 254 (acknowledge).
– Une trame de non acquitement de la commande envoyée. Elle est composée de trois bits, 0, 3, la commande, 255 (non acknowledge) et le code d'erreur.

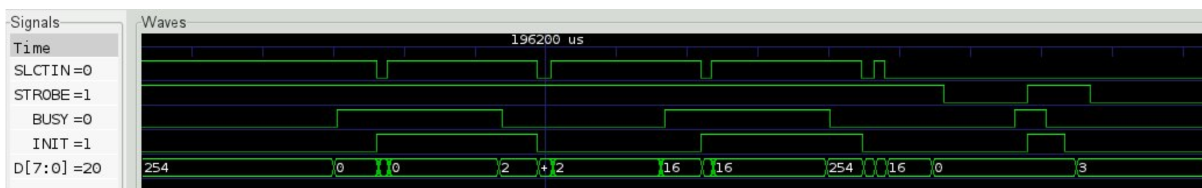


FIGURE 15 – Trame de réception de commande.

À chaque nouveau paramétre envoyé par le RADAR, le bit de commande `SLCTIN` doit passer à l'état bas. Le bit `INIT` doit être placé à l'état haut : il va changer d'état à chaque envoi d'un nouveau caractère. Le bit `SLCTIN` repasse à l'état haut après le changement d'état de `INIT`.

Si la commande est égale á `GMT` (101) nous remplissons un tableau de données avec les valeurs que nous envoie le radar.

6.3 Problèmes rencontrés

Lors du test du programme sur la carte, nous avons constaté que le premier caractère correspondant à la taille de la trame ne correspondait pas à la taille réelle de celle-ci (400 caratères annoncés et 60 relevés).

L'étude du code assembleur ainsi que les espaces mémoire occupés, nous avons constaté que la gestion de la mémoire du microcontrôleur était configurée de façon peu judicieuse : afin que le code soit portable sur tout les types de microcontrôleurs de la gamme STM32 (tailles de mémoires différentes), la position de la pile se situe, par défaut, en position 2048, au milieu de la RAM.

Les valeurs relevées de la trame sont stockées dans un tableau, qui venait écraser la pile. En replaçant la pile a l'adresse la plus élevée de la mémoire, le problème est corrigé.

7 Stockage sur carte Secure Digital (SD)

7.1 Introduction

Afin de pouvoir faire plusieurs relevés au cours du temps, il nous est demandé d'implémenter un programme permettant de stocker les trames sur une carte SD. Ces trames seront interprétées en post-traitement afin d'analyser l'évolution de la composition d'un sol au cours du temps.

Le stockage sur une carte SD se réalise en plusieurs étapes, avec toutes les transactions au travers d'un bus synchrone SPI :

- Dans un premier temps, notre programme testera la présence ou non d'une carte SD dans le lecteur.
- Puis, nous créons un dossier associé à une structure condition permettant de vérifier l'existence du dossier à chaque mesure.
- Ensuite, nous ouvrons ce dossier et créons un fichier .txt pour stocker nos trames.
- Après relevé de la trame, nous la stockons dans le fichier.
- Pour finir, le fichier est fermé et le système de fichier synchronisé physiquement sur la carte.

7.2 Branchement

Une carte SD (Secure Digital) est une carte mémoire amovible de stockage de données numériques, ne nécessitant que peu de signaux de commande (Fig. 16), s'associant chacun à un signal d'un bus SPI (Fig. 13).

L'horloge est connectée sur PA5, CS est branché sur PA4, que les données d'entrées sont reliées sur PA7 et enfin les données de sortie sur PA6.

7.3 Explication du programme

Nous avons utilisé un programme qui a été développé par Mr Friedt qui permettait de communiquer avec une carte SD. Nous avons donc étudié ce programme afin de nous en inspirer pour l'utiliser dans notre cas.

Nous effectuons une écriture sur la carte SD au format FAT, selon une implémentation proposée par la bibliothèque EFSL [8] portée au STM32, afin de stocker sur la carte SD toute les trames relevées. Une variable mémorise la présence de la carte SD dans le lecteur (inutile de tenter d'écrire s'il n'y a pas de carte). La première opération consiste à initialiser la communication avec la carte SD, puis de stocker la trame et enfin de fermer la communication.

8. `efsl.be`

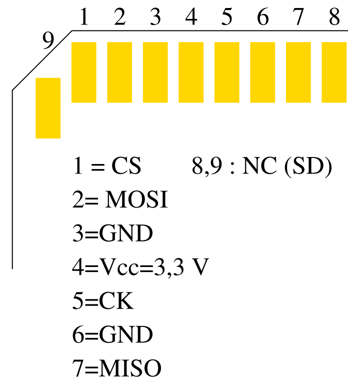


Figure 16 – Brochage d'une carte SD : MOSI est la liaison du microcontrôleur vers la carte, MOSI de la carte vers le microcontrôleur, CK l'horloge du bus synchrone imposée par le microcontrôleur, CS le signal d'activation du périphérique. La carte est alimentée sous une tension de 3,3 V.

7.3.1 Ouverture de la communication

Tous d'abord, on regarde la valeur retournée par la fonction `efs_init()` (initialisation de EFSL). Si cette valeur est différente de 0, cela signifie qu'une erreur s'est produite sur la communication avec la carte SD. Sinon, on met la variable `SD_present` à 1. De plus, on vérifie si le dossier où l'on souhaite stocker le fichier est créé. Si il n'est pas créé, nous le créons grâce à la fonction `mkdir()`.

7.3.2 Stockage de la trame

Lorsque que nous avons vérifié que la carte SD est présente et qu'une trame est relevée, nous devons la stocker. Pour pouvoir la stocker, nous devons dans un premier ouvrir le fichier texte dans lequel on souhaite écrire la trame avec l'option d'ajout de données en fin de fichier (*append*, option d'ouverture "a"). Cette étape permet de ne pas effacer les caractères qui ont déjà été stockés auparavant sur la carte. En cas d'échec de cette ouverture, le fichier est inexistant et il faut le créer : la même opération d'ouverture de fichier est réitérée mais avec l'option de création de fichier *write* (option "w"). Si une de ces deux opération réussit, la variable `SD_present` est à 1 et nous pouvons stocker l'information en effectuant une boucle qui permet de lire toutes les valeurs de la trame issue de l'unité de contrôle et les stocker sur la carte en caractères ASCII. L'instruction qui permet d'écrire est `file_write()`. Cette instruction est composée de trois paramètres :

- le premier est le descripteur du fichier dans lequel on souhaite écrire,
- le second est le nombre d'octets à écrire,
- le dernier est le pointeur vers le tableau contenant les informations à écrire.

7.3.3 Fermeture de la communication

Lorsque que l'écriture de la trame est finie, nous devons fermer le fichier avec l'instruction `file_fclose()`. De plus, nous devons arrêter la communication avec la carte SD grâce à `fs_umount()`. Cette instruction est essentielle car c'est elle qui permet d'écrire physiquement la valeur de la trame dans le fichier et de stocker sur la carte SD : l'instruction `file_write()` écrit la trame dans un tampon.

7.4 Test

Nous avons vu qu'on stockait octet par octet sur la carte SD. Nous possédions une carte de 64 MB. En effectuant un relevé de trames avec 1200 échantillons, nous pouvons stocker 53300 trames, ce qui permet d'effectuer 146 relevés par jour pour un fonctionnement en autonomie pendant un an. Nous avons validé sur une nuit le bon fonctionnement de notre implémentation de ces algorithmes.

8 Gestion de l'énergie

8.1 Introduction

La mise en place de mesure de sol sur une année avec un RADAR ne peut se faire sans une gestion de l'alimentation. Le RADAR, le microcontrôleur et les antennes seront alimentés par des batteries, qui permettront de ne pas intervenir sur le systèmes durant l'année de mesure. Nous avons donc étudié les possibilités du microcontrôleur sur sa mise en veille et sur son réveil de façon périodique pour effectuer des mesures qui seront stockées sur une carte SD.

8.2 Mise en veille

Pour économiser de l'énergie, le microcontrôleur peut entrer dans différents modes de veille :

- Sleep WFI : le microcontrôleur rentre en mode veille lors d'une interruption, mais la gestion de l'alimentation n'est pas gérée dans ce mode,
- Sleep WFE : dans ce mode le microcontrôleur sort du mode veille avec une interruption, mais la gestion de l'alimentation n'est également pas gérée dans ce mode,
- Stop : ce mode veille se réveille par une interruption qu'il faut définir à l'avance,
- Standby : ce mode de veille est celui que nous allons utiliser car il sort du mode veille lors d'une interruption (`RTC_alarm`) et que la gestion d'alimentation est prise en compte. Dans ce mode, il n'y a que le registre backup qui est sauvegardé.

Les configurations des différents états du mode Standby se retrouvent dans le tableau en annexe (Fig. 12.3).

Pour configurer le mode standby, il nous est demandé de configurer plusieurs registres `SCB_SCR` et `PWR_CR`, qui sont détaillés en annexes (Fig. 12.3 et 12.3). Le premier registre concerne l'alimentation du microcontrôleur et le deuxième registre gère le système de contrôle du Cortex. Pour sortir du mode veille, on utilise une boucle "while" qui permet le réveil du microcontrôleur lors d'une interruption (Wait for interrupt).

Ces instructions se retrouvent dans la fonction ci-dessous :

```
void PWR_EnterSTANDBYMode(void)
{  jmf_printf("bienvenue dans le mode veille\r\n");

/* Set SLEEPDEEP bit of Cortex System Control Register */
SCB_SCR |= SCB_SCR_SLEEPDEEP;
/* Select STANDBY mode */
PWR_CR \Iota= PWR_CR_PDDS;

/* Clear Wake-up flag */
PWR_CR |= PWR_CR_CWUF;

while ((PWR_CSR & 1) == 0x00) ;
__asm volatile ("WFI");
}
```

Une partie très importante du projet concerne la gestion de l'énergie. En effet, notre système doit être capable de fonctionner sur batterie sur une année complète sans intervention extérieure. C'est pour cette raison qu'il ne doit pas consommer de l'énergie inutilement. Nous avons donc réalisé plusieurs mesures afin d'observer la consommation de chaque éléments. Pour effectuer la mesure, nous branchons un ampèremètre en série avant le régulateur de tension (LM117). Nous relevons pendant le mode veille que la carte consomme environ 4.1 mA, qu'un courant de 13.5 mA est consommé lorsque qu'une interruption est présente et enfin qu'elle consomme 23 mA pour effectuer un relevé de trame. Le courant le plus gênant se situe dans le mode veille. En effet, les batteries ne seront pas assez puissantes pour tenir une année complète. Nous avons décidé de changer de régulateur car nous pensions que son courant de fuite était important. Mais en effectuant à nouveau une série de mesure, nous avons obtenu des résultats identiques. Comme la carte est développée pour plusieurs applications, un amplificateur opérationel que

nous n'utilisons pas consommait du courant. Lorsque nous avons résolu ce problème nous avons relevé les nouvelles consommations :
- en mode veille, le courant est d'environ 0 mA après quelque secondes (temps de stabilisation après le relevé d'une trame),
- lorsque qu'une interruption est présente, le courant consommé est de 9 mA,
- 20 mA sont consommé pour relever une trame du RADAR.

Ces mesures ont été effectuées sans la carte SD. En ajoutant la carte SD, nous avons remarqué que le courant était à peine plus élevé et avait beaucoup plus de mal à ce stabiliser.

On retrouve les différentes valeurs relevées dans le tableau ci-dessous (Fig. 17) :

Mode de fonctionnement	Consommation avec AOP	Consommation du système
Mode veille	4.1 mA	0 mA
Interruption	13.5 mA	9 mA
Mesure	23 mA	20 mA

FIGURE 17 – Tableau des consommations de courant.

Nous avons exécuté le programme pour relever une trame toutes les demi-heures (Fig. 18). L'acquisition de données a cessé suite au déchargement des batteries alimentant les modules d'émission et de réception des antennes. De plus, l'unité de contrôle du RADAR consomme **1.3 A** sous **7 volts** en permanence. Pour ces éléments, il faudra trouver une solution pour couper l'alimentation en mode veille.

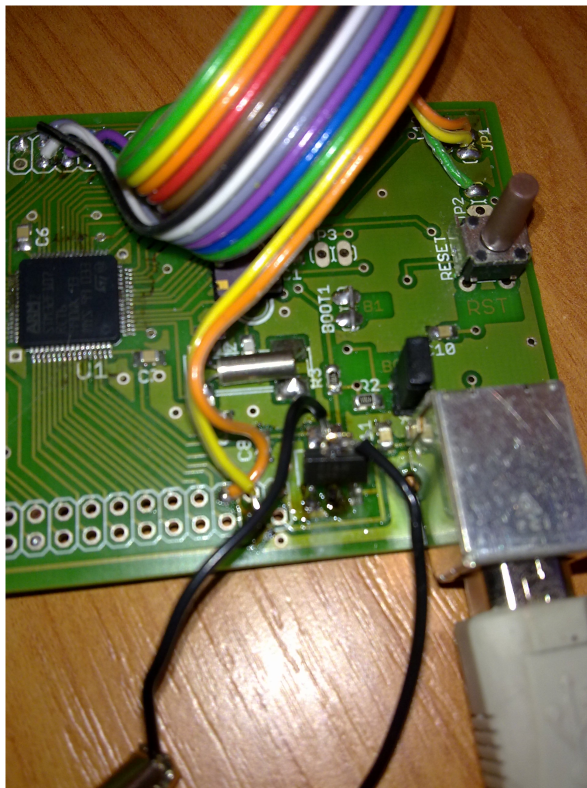


FIGURE 18 – Mesure du courant

8.3 Réveil du microcontrôleur

Pour réveiller le microcontrôleur, il faut réaliser plusieurs actions dans différents sous programmes :
- La procédure `nvic_setup()` permet de gérer le contrôle des vecteurs interruptions, il active donc les interruptions : il fait appel au sous programme `rtc_isr()`.

- Dans le programme `rtc_isr`, on choisit quel type d'interruption nous voulons faire, dans notre cas une interruption sur l'alarme et sur le compteur. Pour faire des interruptions, nous jouons sur l'état des drapeaux et registres, en les plaçant à l'état bas. L'interruption `RTC_ALARM` permet de réveiller le microcontrôleur lorsque le compteur d'horloge temps rél atteint un certain seuil. Ce seuil est défini dans l'interruption `RTC_SECF` qui est utilisée pour le compteur. Dans celle-ci, nous allons lancer le compteur et définir une variable (`temps_mesure`). Cette variable permet de choisir l'intervalle de temps entre deux réveils, elle est définie en variable globale pour pouvoir la changer dans toute fonction, incluant les gestionnaires d'interruptions. Dans cette interruption nous plaçons également à '1' une variable `my_val` afin d'indiquer que nous avons fini de traiter l'interruption et effectuer les actions en conséquent dans le programme principal.
- Dans la fonction `main()`, nous plaçons tout d'abord la variable `my_val` à '0' afin de définir que nous ne somme pas dans une interruption. Puis nous utilisons la fonction `rtc_auto_awake` qui permet de configurer le mode *standby*. Il faut préciser le type d'horloge utilisée et son diviseur de fréquence (*prescale*). Celui-ci va définir la fréquence de cadencement du compteur, dans notre cas 1 seconde. Par la suite, nous faisons appel la fonction `nvic_setup`, pour définir les vecteurs d'interruptions. Ensuite nous lançons les interruptions pour l'alarme et pour le compteur, en plaçant les drapeaux et les registres correspondant à l'état haut. Nous rappelons également la fonction `rtc_set_alarm_time(rtc_get_counter_val() + temps_mesure);` afin de lancer le compteur. Lancer le compteur une fois dans le programme principal (`main()`) peut être considéré comme une initialisation du compteur.
- Pour finir nous réalisons le programme de mesure et de stockage sur la carte sd. À la fin de celui-ci nous remettons la variable `my_val` à '0' afin de dire que nous avons fini l'interruption, et nous faisons appel au sous-programme de mise en veille du microcontrôleur.

8.4 Stockage des paramètres sur la carte

Notre programme étant destiné à réaliser des relevés pendant une année, il est important de savoir à quel moment la mesure de trace a été effectuée, en cas de raté de mesures. Nous voulions donc avant chaque relevé de trames préciser une indication de temps du moment de la mesure ainsi qu'une variable correspondant à l'indice du relevé. Le problème est qu'entre chaque mesure nous mettons en veille le microcontrôleur, induisant une perte de toute les variables conservés dans la RAM. Nous avons constaté dans le documentation du STM32 que les registres "Backup domain (BKP)" et "RTC" sont conservés.

Nous avons donc décidé d'utiliser le registre BKP pour stocker notre valeurs de comptage du nombre de relevés et RTC pour l'horloge (comme pour le réveil).

Cependant, pour pouvoir utiliser ces deux registres, il faut activer leurs horloges avec la fonction ci-dessous : (nous la plaçons dans la sous partie `clock_setup`)

```
rcc_peripheral_enable_clock(&RCC_APB1ENR, RCC_APB1ENR_BKPEN | RCC_APB1ENR_PWREN);
```

Pour réaliser une horloge qui est activée depuis la première mesure, nous utilisons une fonction `temps()` (expliquée ci-dessous).

Le but de cette fonction est de convertir au bon format les données de temps en heure, minute, seconde et de les stocker dans un tableau de 9 éléments. Nous insérons également dans ce tableau les séparateurs " :" entre les heures, les minutes et les secondes.

Les données d'heure, de minute et de seconde sont générées par le sous programme `RTC_GetTime()`, qui va récupérer les valeurs de la fonction qui permet de compter (`rtc_get_couter_val`) et les convertir en heure, minute et seconde en divisant la variable générée `rtc_get_couter_val` (division par 3600 pour les heures et de 60 pour les minutes).

Nous exploitons à nouveau le tableau de données dans le programme `sd_ouvre()`, où nous écrivons les valeur sur la SD avant chaque trame.

Pour la variable stockée dans le registre BKP, nous avons fait un sous programme dans lequel :
- il faut activer le registre de contrôle pour pouvoir écrire dans le registre BKP (`PWR_CR |= PWR_CR_DBP;`)
- Nous lisons la valeur du premier registre de BKP et nous stockons sa valeur dans une variable `nv = BKP_DR1;`
- nous incrémentons cette variable (`nv++;`)

– puis nous réalisons une fonction pour convertir cette valeur en hexadécimal que nous stockons dans un tableau de 4 octets.
– Pour finir nous écrivons la nouvelle valeur de cette variable dans le registre 1 de BKP (`BKP_DR1 = nv;`)

Pour prendre en compte cette variable nous lançons ce sous programme dans notre fonctionne `main`, et nous écrivons la valeur sur la carte SD dans la fonction `sd_ouvre` (`file_write(&file, 4, l);`) à la suite du temps de mesure.

Un exemple de trame enregistrée sur la carte SD RECORD est proposée Fig. 19.

```
00:00:00  0001
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
00:10:01  0002
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
00:15:02  0003
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
00:20:03  0004
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
00:25:03  0005
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
00:30:04  0006
0A28007C007C007C007C007C007C007C007C007C007C007C007C007
```

Figure 19 – Trame enregistrée sur la carte SD

9 Interface graphique

Afin de démontrer la capacité du microcontrôleur d'acquérir des données et de les transmettre à un PC, une interface graphique libre et portable (langage Qt) permettant à un utiisateur de visualiser en temps réel les trames acquises est développée. La communication entre le microcontrôleur et un PC nécessite de respecter scrupuleusement un protocole de communication défini avec le développeur de l'interface graphique.

9.1 Cahier des charges

Nous avons défini avec le développeur de l'interface graphique un cahier des charges :
– l'interface doit permettre de régler les paramètres de fréquence, du nombre d'échantillons et de la position initiale de l'onde dans l'air,
– l'interface graphique doit permettre d'effectuer plusieurs mesures avec les même paramètres sans avoir à les renvoyer au RADAR à chaque mesure,
– l'interface graphique doit envoyer un caractère " !" avant chaque séquence de paramétrisation du RADAR, et " ?" pour requérir une trace,
– en réponse à la requête, le microcontrôleur envoie " ?" quand il a fini d'initialiser les paramètres du RADAR et un "#" en fin de trame,
– l'interface graphique doit fournir 6 paramètres correspondant aux octets de poids faible et fort de la fréquence, du nombre d'échantillons et de la position initiale de l'onde dans l'air. Ces paramètres sont définis par les formules vues précédemment dans la partie 5.3.

9.2 Programme

Pour le programme dialoguant avec l'interface graphique, nous avons repris les bases de la structure du programme vu précédemment. Pour simplifier le code et son utilisation nous avons regroupé les

commandes d'initialisation dans une fonction (`init()`) afin de pouvoir les appeler plus facilement dans le programme principale.

Nous avons créé une fonction (`setting()`), qui contient tous les paramètres de réglage (`sample`, `S_freq` et `Sig_pos`), ainsi que la fonction `stop()`. Le fait de faire un `stop()` permet de sortir du mode aquisition et de prendre en compte des nouveaux paramètres.

Le microtronôleur attend que l'interface graphique envoie un "!" suivi des paramètres afin de les transmettre au RADAR. Nous envoyons un "?" pour annoncer que le radar est prêt pour envoyer une trame. Une fois que nous recevons un "?", nous relevons une trace avec le RADAR, et ajoutons un "#" en fin de trace. Pour finir nous transférons à l'interface utilisateur une trace et attendons soit de nouvelles commandes, soit une demande de trace.

9.3 Utilisation de l'interface graphique

Une fois l'interface graphique lancée et la carte branchée en USB, il faut définir les paramètres de la liaison. Pour cela il faut cliquer sur le logo en haut à gauche (rond rouge), puis il faut régler les paramètres suivant de la liaison asynchrone (Fig. 20) :

1. le type de liaison, pour cela il faut cliquer sur "scan",
2. la vitesse de connexion (Baude Rate),
3. le nombre de bits de données,
4. l'utilisation ou non du bit de parité,
5. l'utilisation ou non du bit d'arrêt,
6. l'utilisation ou non du contrôle de flux,

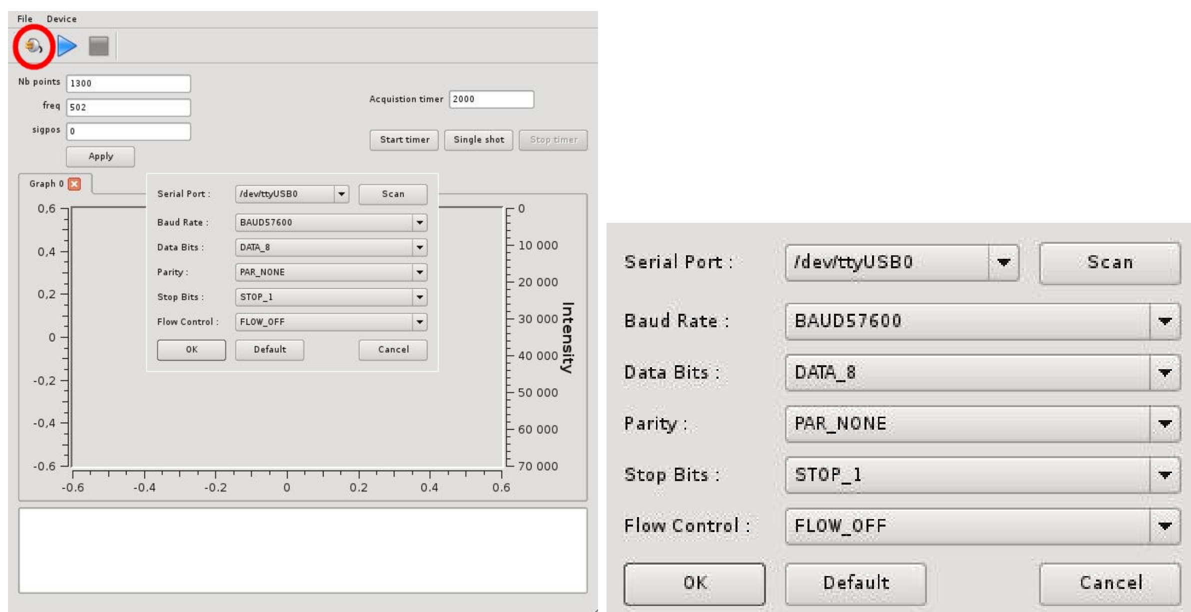


FIGURE 20 – Réglage de la liaison

Une fois la liaison établie, il faut définir le nombre de points, la fréquence et la position de départ de l'onde dans l'air en remplissant les cases définies à cet effet. Pour appliquer les valeurs, il faut cliquer sur la touche "Apply". On peut constater que les paramètres ont bien été pris en compte lorsque l'on reçoit "receptions : ?" dans la boîte de dialogue en bas de l'interface.

Pour relever une trace, il y a plusieurs possibilités :

1. un relevé simple, en cliquant sur la case "Single shot",

2. un relevé en continu périodique, avec un intervalle de temps à définir dans la case "Acquisition timer". Une fois l'intervalle défini, il faut cliquer sur "start timer" pour lancer les acquisitions et sur "stop timer" pour les arrêter.

Les relevés s'affichent en bas dans le graphique : nous visualisons l'intensité de l'onde électromagnétique reçue par le RADAR par une variation de couleur en fonction du temps (Fig. 21).

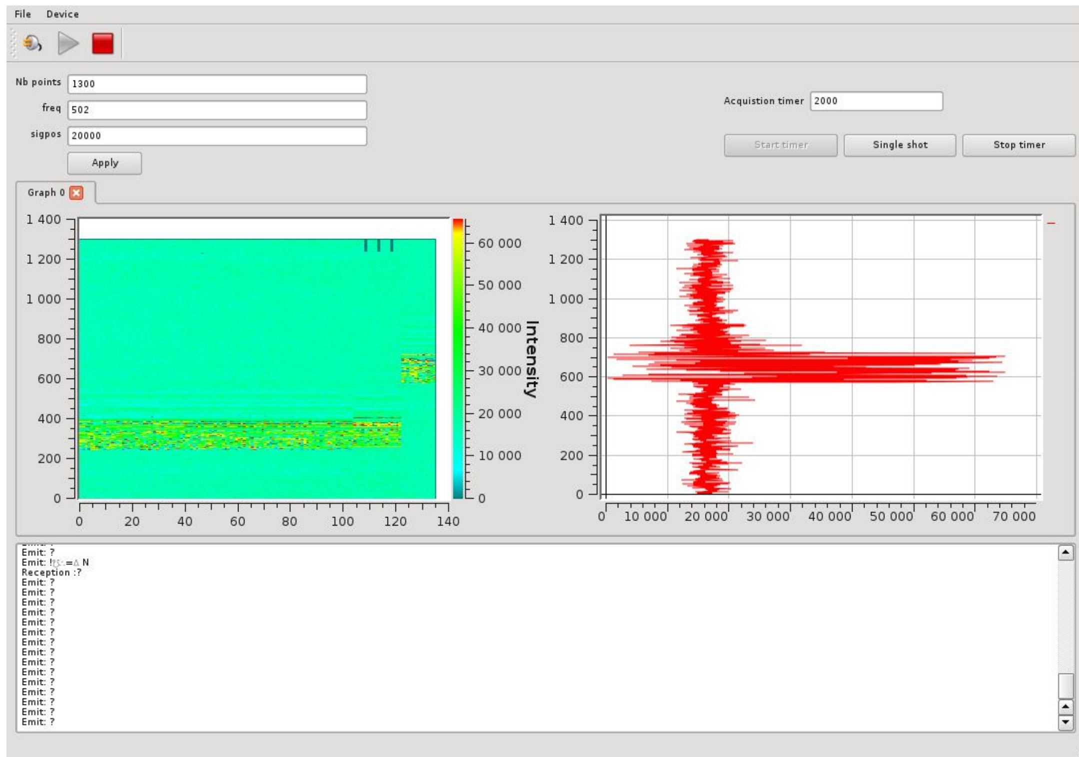


FIGURE 21 – Interface graphique permettant l'affichage en temps réel des trames acquises par une unité de contrôle CU et transitant vers le PC par l'intermédiaire d'un microcontrôleur STM32 et une liaison USB. À droite la dernière trace acquise (A scan), à gauche la séquence de toutes les traces acquises dont l'intensité du signal reçu est codé par la couleur du pixel (B scan). Noter qu'autour de la date 120, une commande de modification de l'origine de l'acquisition a été émise (`sigpos`), se traduisant bien par un décalage de l'origine des traces affichées. Les trois paramètres fondamentaux (fréquence d'échantillonnage, nombre déchantillons et origine de l'acquisition) sont accssibles en haut à gauche, l'intervalle de temps entre deux acquisitions en haut à droite.

Pour le moment, les traces sont sauvegardées soit sous forme d'une séquence de valeurs binaires, soit sous forme d'une image codant la couleur de chaque pixel. Ces formats ne sont pas standard et ne permettent pas une exploitation ultérieure des données acquises : la sauvegarde dans un format standard ouvert, compatible avec les outils classiques de la communauté [4] – SEG Y par exemple [9] – doit encore être implémentée. L'objectif serait au mieux de pouvoir en temps réel exploiter les algorithmes de traitement fournis par la bibliothèque Seismic Unix pour améliorer la qualité de l'affichage, ou au pire faciliter le post-traitement en fournissant au moins un fichier reconnu par ces outils.

10 Nouveau RADAR : l'unité de contrôle CUII

10.1 Introduction

Nous avons essayé de comprendre le protocole d'une unité de contrôle plus récente Malå CUII (Fig. 22). Pour cela, nous avons utilisé un analyseur logique (Tektronix TLA5201) qui permet de sonder les

9. `http://en.wikipedia.org/wiki/SEG_Y`

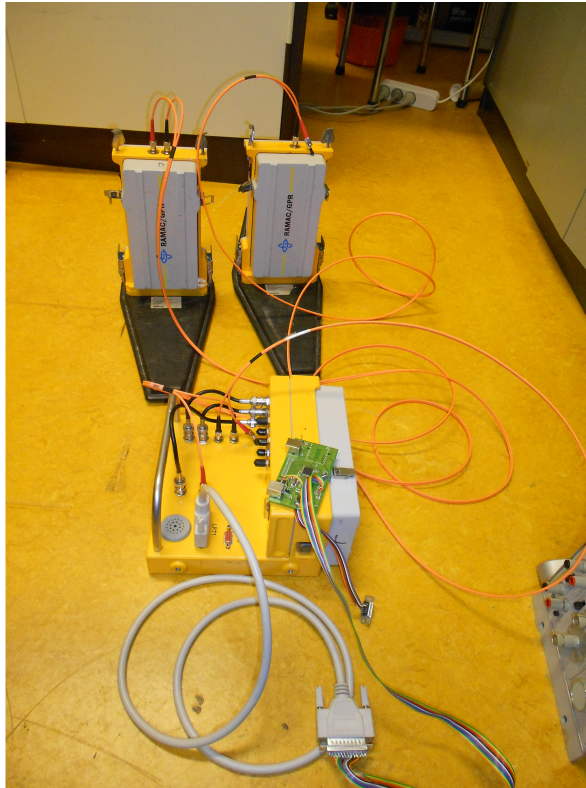


Figure 22 – Unitde controlôle de RADAR CUII.

signaux présents sur le port parallèle : cet instrument permet une mesure sur plus de voies et surtout avec une profondeur de mémoire bien plus importante que le Logic Sniffer. En effet, les trames du CUII sont longues et une compréhension du protocole initialement inconnu nécessitait une durée de mesure supérieure (Fig. 23) à ce que permet le Logic Sniffer (limité à 24 kmesures).

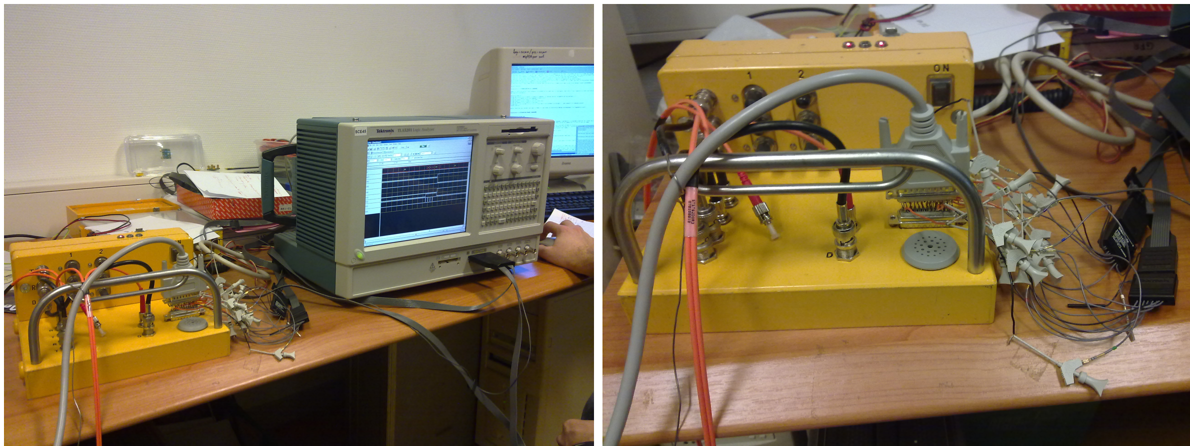


Figure 23 – Sondage du protocole de communication de l'unité de contrôle CUII.

Nous utilisons le logiciel GroundVision fourni par le constructeur Malå pour initier les communications avec l'unité de contrôle CUII. Nous avons donc sondé les signaux lorsque nous envoyons des commandes depuis ce logiciel au RADAR. La grande différence entre les deux logiciels se situe sur l'envoi des paramètres. En effet, avec le logiciel commandant l'ancien RADAR (unité de contrôle CU), nous mod-

ifions les paramètres et il faut appuyer sur la fonction "test" pour que le RADAR les prenne en compte. Avec le CUII, lorsque que l'on règle un paramètre, celui-ci est envoyé immédiatement au RADAR.

10.2 Définition d'une trame

Comme le port parallèle est exploité en mode bidirectionnel, nous avons eu du mal à identifier quels octets étaient reçus par le RADAR et quels octets étaient envoyés. Cependant nous avons pu remarquer que pour la première trame, les bits de commandes qui évoluaient étaient `STROBE` et `BUSY`. Pour la deuxième trame, c'était `ACK`. Pour la troisième trame, les bits étaient `strobe` et `busy` et ainsi de suite. Nous en avons conclu qu'une commande sur deux était l'envoi des paramètres et l'autre, la réponse du RADAR (Fig. 24).

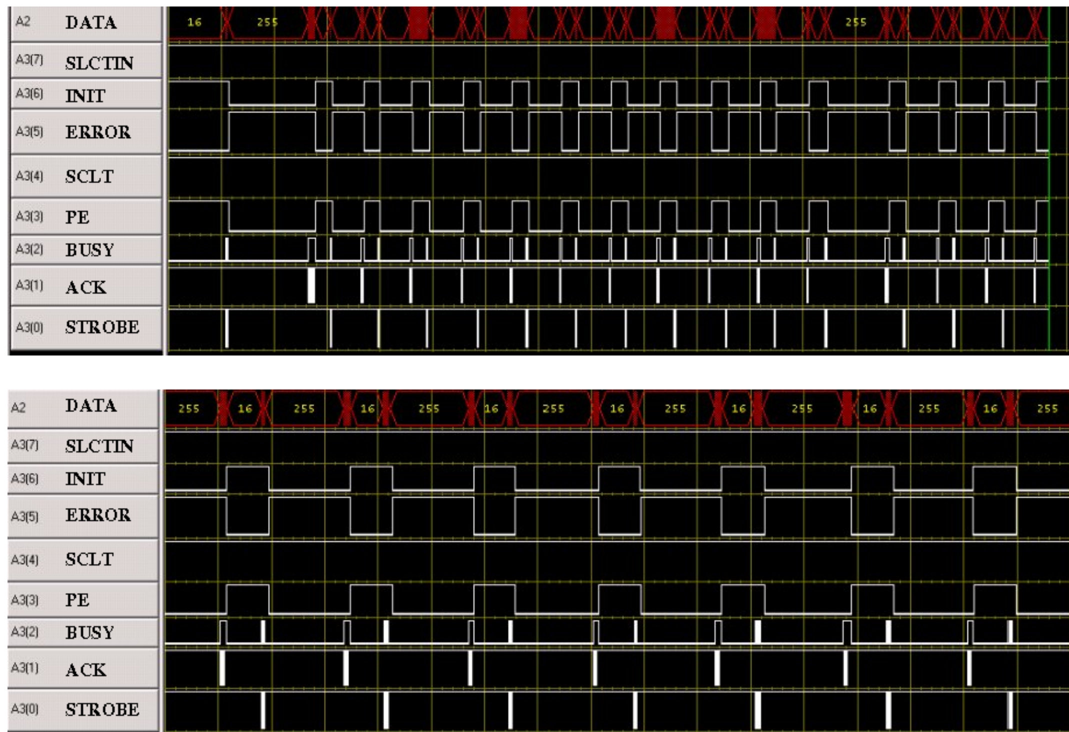


FIGURE 24 – Trame complète lors d'une transaction avec l'unité de contrôle CUII (en haut), et zoom sur une partie de la transaction (bas). A2 correspond au bus de données, A3(0) correspond au signal STROBE, A3(1) à ACK, A3(2) à BUSY, A3(4) à PE, A3(4) à SCLT, A3(5) à ERROR, A3(6) à INIT, et A3(7) à SCLTIN.

10.3 Envoi de la commande

Au départ, lorsque le RADAR reçoit un octet provenant de l'ordinateur, le bit de commande `STROBE` passe à zéro. Puis quelque instant après, le bit de commande `BUSY` passe à l'état haut. Puis quand l'octet est correctement reçu, le bit `STROBE` passe à 1 et `BUSY` à zéro. Cette étape est répétée pour tous les octets d'une même commande. De plus, à la fin d'envoi de chaque commande, `INIT` passe à l'état bas, puis `ERROR` à l'état haut et enfin `PE` passe à l'état bas (Fig 25).

La trame correspondant à l'envoi d'une commande est identique à celle de l'ancien RADAR, c'est à dire qu'en premier est envoyée la valeur 0 (octet de poids fort de la taille de la trame), puis le nombre restant d'octets à transmettre, suivi du numéro de la commande et enfin les différents paramètres associés à la commande. De plus, nous avons pu voir que les numéros des commandes étaient sensiblement les mêmes pour les deux unités de contrôle.

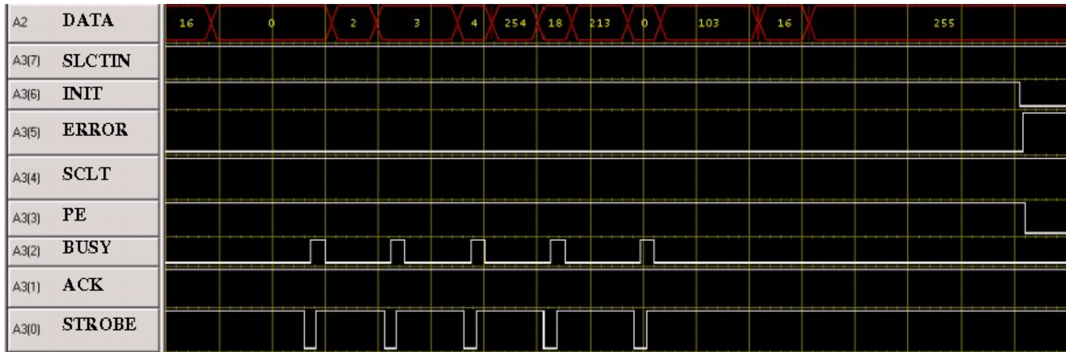


FIGURE 25 – trame d'envoie : A2 correspond au bus de données, A3(0) correspond au signal STROBE, A3(1) à ACK, A3(2) à BUSY, A3(4) à PE, A3(4) à SCLT, A3(5) à ERROR, A3(6) à INIT, et A3(7) à SCLTIN.

10.4 Réponse du RADAR

Lorsque qu'une commande est correctement envoyée, le RADAR envoie une trame composée de plusieurs octets. Le bit de commande `BUSY` passe à l'état haut pendant toute la durée de la réponse. De plus, `ACK` change d'état à chaque octet reçu (passe de 0 à 1). Lorsque la trame est finie, le bit `INIT` passe à 1, puis le bit `ERROR` passe à 0, ensuite le bit `BUSY` est à 0 et enfin `PE` passe à l'état haut.

On distingue deux cas de figure dans la trame renvoyée par le RADAR. Le premier est une trame pour signifier que l'envoi de commande s'est déroulé correctement. La trame renvoyée est composée de quatre octets. Tous d'abord, on trouve la valeur 0, puis le nombre d'octets restant à transmettre, ensuite le numéro de la commande et enfin la valeur 254 (qui correspond à un acquittement (*acknowledge*), Fig. 26).

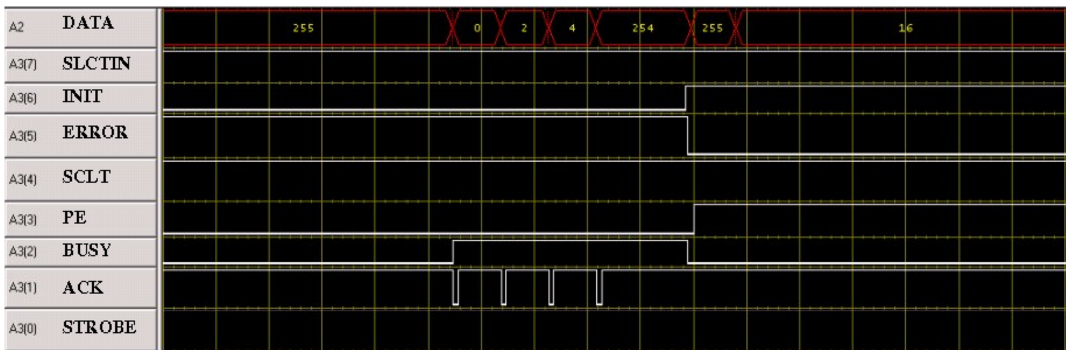


FIGURE 26 – Trame d'acquitement : A2 correspond au bus de données, A3(0) correspond au signal STROBE, A3(1) à ACK, A3(2) à BUSY, A3(4) à PE, A3(4) à SCLT, A3(5) à ERROR, A3(6) à INIT, et A3(7) à SCLTIN.

Le second cas de figure indique qu'un problème est survenu lors de l'envoi de la commande. La trame renvoyée est donc composée de cinq octets. Tous d'abord, on trouve la valeur 0, puis le nombre d'octets restant à transmettre, ensuite le numéro de la commande et enfin un code d'erreur composé de deux octets (Fig. 27).

Il semble surprenant que le RADAR impose la cadence des communications en n'abaissant que ACK lors du positionnement de chaque nouvelle valeur sur le bus de données, alors que tous les protocoles obserbés jusqu'ici attendaient un acquittement de l'interlocuteur. Sonder un signal additionel – broche 14 du port parallèle, initialement déconnectée sur l'unité de contrôle plus ancienne (et dans la documentation constructeur) – confirme cette intuition. La broche 14, commandée par le PC pour informer le CUII de la réception de la trame, est manipulée en alternance avec ACK (Fig. 28).

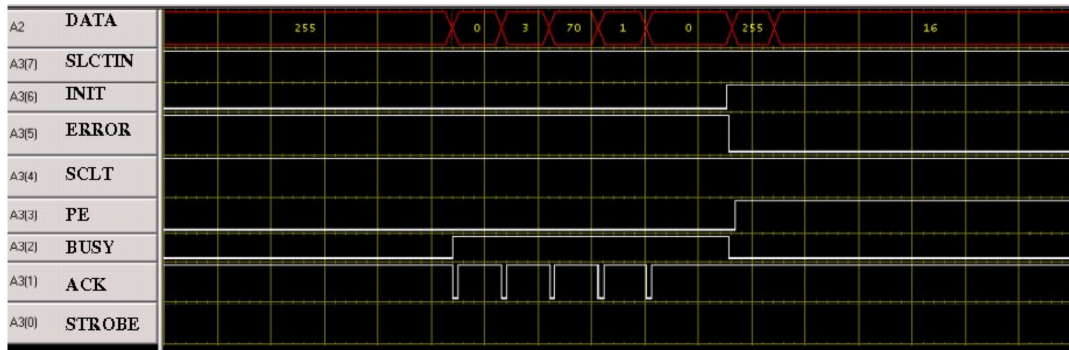


FIGURE 27 – trame de non acquitement : A2 correspond au bus de données, A3(0) correspond au signal STROBE, A3(1) à ACK, A3(2) à BUSY, A3(4) à PE, A3(4) à SCLT, A3(5) à ERROR, A3(6) à INIT, et A3(7) à SCLTIN.

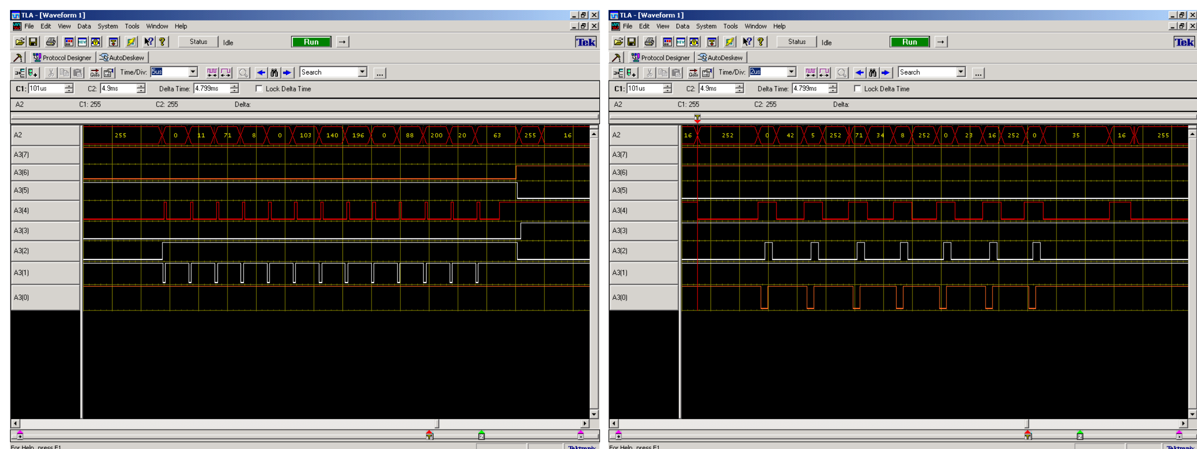


FIGURE 28 – Trames de communication, avec sondage de la broche 14 (en rouge) commandée par le PC.

10.5 Réglage des différents paramètres

Nous avons pu constater que les réglages des paramètres s'effectuaient de la même façon entre les deux RADARs (Fig. 29). Les octets correspondant aux nombre d'échantillons sont toujours $BH \times 256 + BL$ (exemple, pour définir un nombre d'échantillons de 1221, les paramètres envoyés sont 4 et 197 puisque $4{\times}256{+}197{=}1221$).

L'octet correspondant à `S_stack` est toujours un nombre en fonction de la puissance de 2 (exemple, pour un nombre de stacks à 2, le paramètre envoyé est 1).

Les octets correspondant à la position initiale sont $BH \times 256 + BL$ (exemple, pour définir une position de 50852, les paramètres envoyés sont 164 et 198).

Enfin, les octets correspondant à `S_freq` forment toujours une hyperbole (cependant, nous n'avons pas réalisé assez de relevés pour en déduire une équation).

10.6 Réception de la trame

Nous avons cherché a relever la trame envoyée par le RADAR à l'ordinateur. Cette trame est décrite Fig. 30.

Afin de pouvoir observer la trame complète, nous avons dû augmenter le temps d'échantillonnage et par conséquent certaines informations ont été perdues. On distingue deux parties : la première composée d'un petit paquet et la seconde de la trame mesurée.

Premiere partie : la première commande qui est envoyée au RADAR est décrite Fig. 31.

Trame	paramètre1	nbre de params.	num. cmde	BL	BH	param. 6	param. 7	commande
1	0	5	71	8	0	16	0	inconnue
3	0	1	70					Start
5	0	2	32	0				inconnue
7	0	3	1	197	4			S_Sample
9	0	2	2	1				S_Stack
11	0	3	3	164	198			S_Sigpos
13	0	3	4	18	0			S_Freq
15	0	2	5	87				S_triggs
17	0	5	6	0	0	0	0	S_triggc
19	0	3	30	0	1			S_Pdir
21	0	3	8	0	1			S_Mdir
23	0	5	2	0	28	0	0	inconnue
25	0	3	27	244	181			S_Tadj
27	0	3	20	0	0			S_Summer (bip)
29	0	1	41					inconnue
31	0	2	32	0				inconnue
33	0	1	65					inconnue

FIGURE 29 – Tableau des commandes

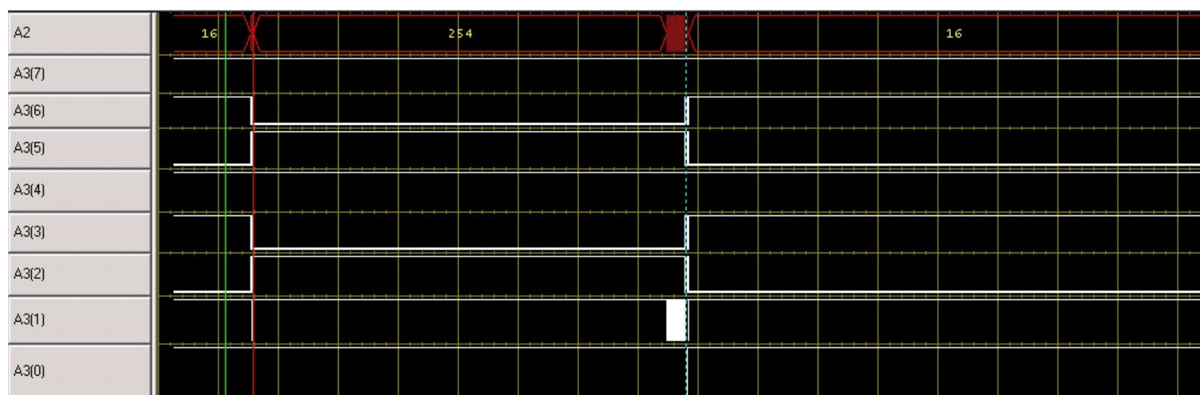


FIGURE 30 – Relevé complet

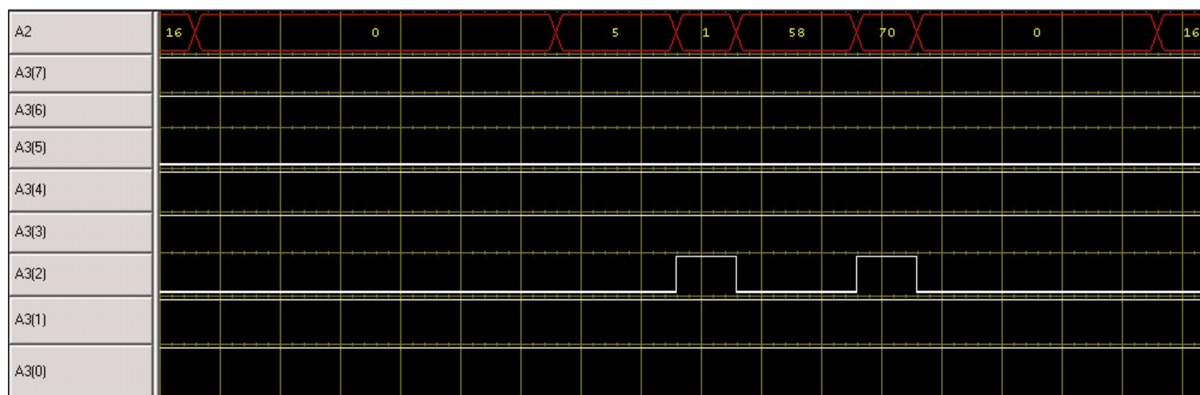


FIGURE 31 – Première commande envoyée.

On remarque que la trame ressemble aux trames relevées précédemment. Elle est composée d'un 0, puis d'un 1 et enfin de la valeur 70 (supposée être un START). Nous observons la réponse de la Fig. 32.

Puis nous étudions la deuxième commande (Fig. 33).

L'allure de cette commande est identique aux précédentes. Nous obtenons la valeur 0, 1 et 14 (qui

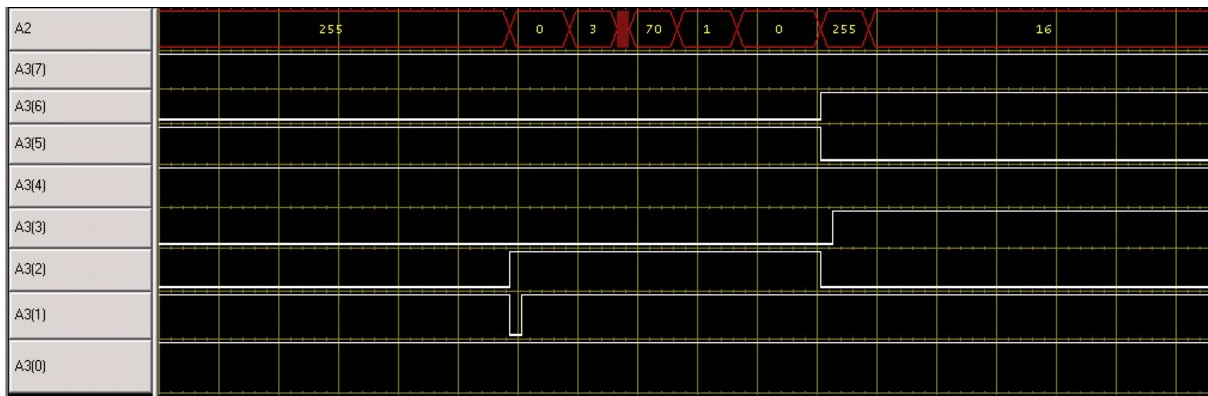


FIGURE 32 – Réponse à la première commande (Fig. 31).

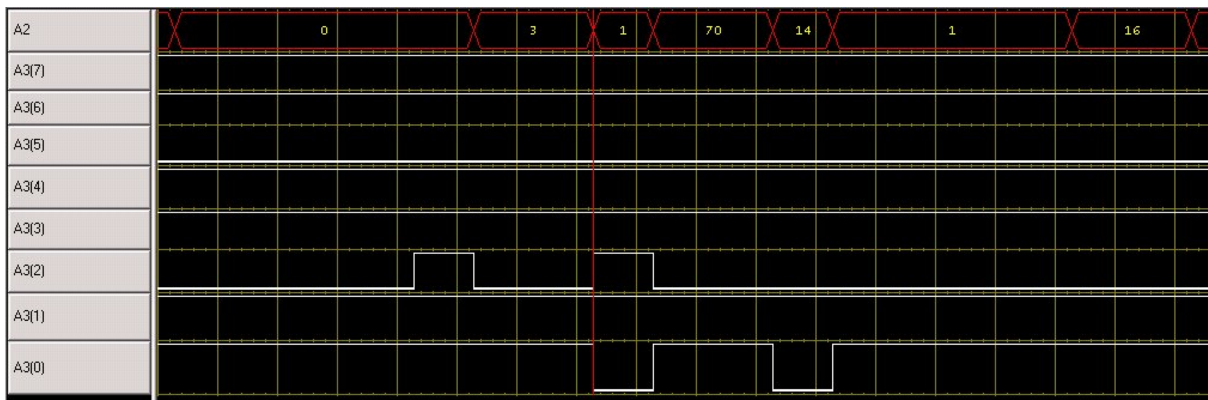


FIGURE 33 – Deuxième commande

correspondait à la commande TRIGG de l'ancien RADAR).

Deuxième partie, la trame mesurée est proposée sur Fig. 34.

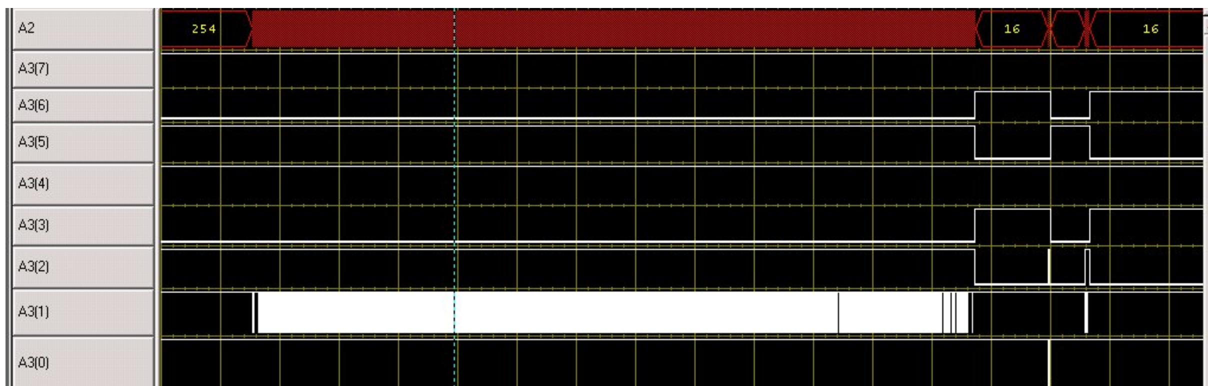


FIGURE 34 – Trame complète

On y observe bien la trâme mesurée et une courte trâme à la fin, dont un grossissement est fourni Fig. 35.

On remarque que sa composition est identique aux précédentes. Nous supposons que la valeur 58 est un STOP.

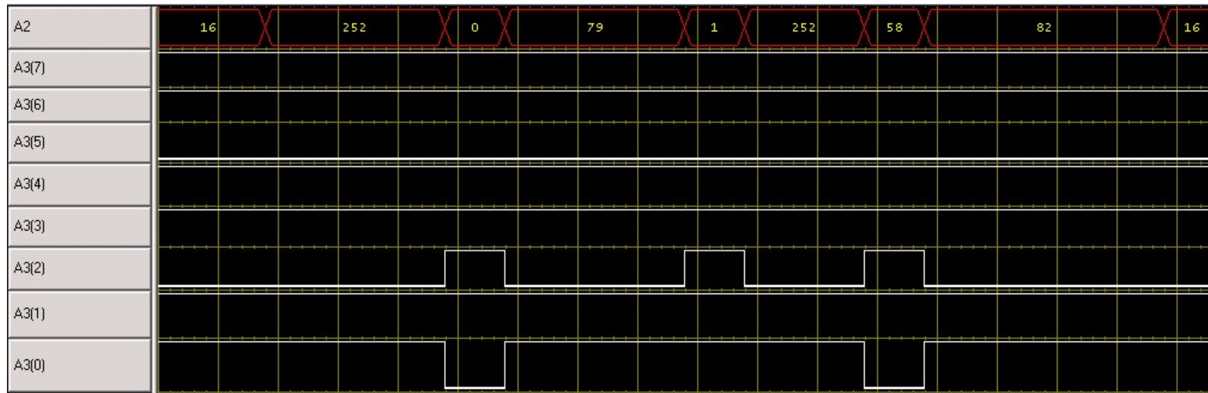


Figure 35 – Caractère de fin de trame

11 Conclusion

Au cours de ce travail, nous avons analysé en détail les protocoles de communication des RADARs de sol développés par Malå en vue d'une part d'en automatiser le fonctionnement pour une application autonome d'acquisition de données à long terme, et accessoirement pour fournir un environnement (interface graphique) d'utilisation libre implémentant les divers protocoles des communication entre système embarqué et PC.

Nous avons apprécié travailler sur ce projet et ce pour différentes raisons

- Tous d'abord nous avons pu comprendre un protocole inconnu en utilisant un analyseur logique afin d'observer l'envoi des paramètres de l'ordinateur au RADAR.
- En nous aidant des documents fournis et lorsque le protocole fut assimilé, nous avons implémenté ce dernier sur un microcontrôleur afin de pouvoir envoyer et recevoir des informations du RADAR et ainsi relever la trame de mesure.
- Ensuite, nous avons réalisé un programme permettant de stocker la trame sur une carte SD. Cela permet d'effectuer des mesures sans qu'un utilisateur intervienne pour enregistrer la mesure.
- Puis, nous avons travaillé sur la gestion de l'énergie du microcontroleur et avons configuré les registres afin de le mettre en veille lorsque qu'aucune mesure est effectuée.
- Ensuite, nous avons réalisé un second programme, sensiblement identique au premier, qui permet de communiquer avec une interface graphique, permettant d'observer en temps réel les relevés du RADAR.
- Enfin, nous avons repris la même étude sur un nouveau radar plus recent (CUII).

Perspectives :

- gérer l'énergie des modules de transmission et de réception qui démontrent un courant de fuite excessif. Cette étape passe par l'ajout au niveau matériel d'optocoupleurs commandés par fibre optique depuis le microcontrôleur.
- remplacer la liaison par port série virtuel sur bus USB par une liaison sur bus USB native proposant un débit au moins 40 fois plus élevé. Cette étape passe probablement par le développement d'un *driver* dédié sur l'ordinateur de contrôle,
- finaliser l'interface graphique utilisateur, et notamment en sauvegardant les fichiers dans un format aisément accessibles par les outils de traitements de la communauté (SEG Y),
- finaliser l'implémentation du protocole de l'unité de contrôle CUII.

12 Annexes

12.1 Tableau de commandes

Liste des commandes décrites dans une documentation gracieusement fournie par Malå. Les commandes que nous avons utilisé sont indiquées en gras. Les commandes présentant un acronyme mais non-documentées ne sont pas mentionées ici.

Menemonique	Numéro	Description
S_SAMP	1	nombre d'échantillons
S_STACKS	2	nombe d'empilement(s)
S_SIGPOS	3	position du signal
S_FREQ	4	fréquence d'échantillonnage
S_TRIGGS	5	source de déclenchement
S_TRIGGC	6	condition de déclenchement
S_MDIR	8	définit la direction de mesure
S_WPOS	9	définit la position de la roue codeuse
START	12	commence une mesure
STOP	13	arrête une mesure
TRIGG	14	déclenchement
RESET_C	15	réinitialise l'unité de contrôle et passe en communication parallèle Centronics
RESET_P	16	réinitialise l'unité de contrôle et passe en communication parallèle bi-directionnelle
RESET_S	17	réinitialise l'unité de contrôle et passe en communication série
HALT	18	arrête temporairement une acquisition
CONT	19	continue une acquisition temporairement arrêtée
S_SUM	20	active fonction de sommation
R_WHEEL	21	réinitialise la roue codeuse
E_WHEEL	22	active la roue codeuse
D_WHEEL	23	désactive la roue codeuse
S_SIMUL	25	mode simulation
S_TADJ	27	pente du balayage
B_FUNC	28	fonction du bouton poussoir
R_FREQ	29	taux de répétition
GMT	101	réception de la trame acquise

12.2 Programme pour un fonctionnement autonome (stockage sur carte SD et veille)

```
1 // compiled using arm-none-eabi-gcc 4.4.2 (compiled using summon-arm-toolchain)
  //   and libopenstm32 (version downloaded from git in april 2011)
3
  #include <libopencm3/stm32/f1/rcc.h>
5 #include <libopencm3/stm32/f1/gpio.h>
  #include <libopencm3/stm32/f1/rtc.h>
7 #include <libopencm3/stm32/usart.h>
  #include <libopencm3/stm32/pwr.h>
9 #include <libopencm3/stm32/nvic.h>
  #include <stdlib.h>
11 #include <comm.h>
  #include "usart.h"
13 #include <scb.h>
  #include "bkp.h"
15 ////////////////////////////////////////////////////////////////////////////////

17 #define temps_mesure 10 // en seconde (double pour la premiere mesure + 1 seconde pour relever trace)

19 ////////////////////////////////////////////////////////////////////////////////

21 #define sd_efsl
  #define jmf_putchar comm_put  // fourni dans comm.c
23
  // EFSL
25 #ifdef sd_efsl
  #include <efs.h>
27 #include <ls.h>
  #include <stdio.h>
29 #include <string.h>

31 volatile unsigned char SD_Present;
```

```
33#define SDFLUSH 50     // toutes les SDFLUSH trames, on ferme & rouvre la SD

35 EmbeddedFileSystem efs;
  EmbeddedFile file;
37
  #endif
39
  char ouvre_sd (void);
41
  #define MAXITER 0x7ffff
43#define DELA   10

45// CU input signals
  #define STROBE  0x01     // B0
47#define INIT  0x04
  #define SLCTIN  0x08
49
  // CU output signals
51#define BUSY  0x02     // instead of 0x20

53#define TAILMAX 2700

55// insterrupt
  #define send_char(USART, c)  do { \
57   USART_DR(USART) = (c&USART_DR_MASK); \
    while ((USART_SR(USART) & USART_SR_TXE) == 0); \
59   } while(0)

61 char bigtab[TAILMAX];

63 char b;
  volatile int nv = 0;
65 volatile char x[8], l[4];
  char cc;
67
  volatile char my_val = 0;
69
  /* Set STM32 to 72 MHz. */
71 void clock_setup (void)
  {
73   rcc_clock_setup_in_hse_8mhz_out_72mhz ();

75   /* Enable GPIOC clock. */
    rcc_peripheral_enable_clock (&RCC_APB2ENR,
77               RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN |
               RCC_APB2ENR_IOPCEN);
79
    /* Enable clocks for GPIO port B (for GPIO_USART3_TX) and USART3. */
81   rcc_peripheral_enable_clock (&RCC_APB2ENR, RCC_APB2ENR_USART1EN);

83   // enable clock
    rcc_peripheral_enable_clock (&RCC_APB1ENR, RCC_APB1ENR_BKPEN | RCC_APB1ENR_PWREN);  →
       ↪//BKP (backup domain)
85
  }
87
  void usart_setup (void)
89{
    /* Setup GPIO pin GPIO_USART1_TX. */
91   gpio_set_mode (GPIOA, GPIO_MODE_OUTPUT_10_MHZ,
       GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
93
    /* Setup UART parameters. */
95   usart_set_baudrate (USART1, 57600);
    usart_set_databits (USART1, 8);
97   usart_set_stopbits (USART1, USART_STOPBITS_1);
    usart_set_mode (USART1, USART_MODE_TX_RX);
99   usart_set_parity (USART1, USART_PARITY_NONE);
    usart_set_flow_control (USART1, USART_FLOWCONTROL_NONE);
```

```
101
      /* Finally enable the USART. */
103   usart_enable (USART1);
    }
105
    void gpio_setup (void)
107 {
      /* Set GPIO12 (in GPIO port C) to 'output push-pull'. */
109   gpio_set_mode (GPIOB, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
         GPIO8 | GPIO9 | GPIO10 | GPIO11 | GPIO12 | GPIO13 | GPIO14
111      | GPIO15);
      gpio_set_mode (GPIOC, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
113      INIT | STROBE | SLCTIN);
      gpio_set_mode (GPIOC, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, BUSY);
115 }

117 void nvic_setup (void)
    {
119   /* Without this the RTC interrupt routine will never be called. */
      nvic_enable_irq (NVIC_RTC_IRQ);
121   nvic_set_priority (NVIC_RTC_IRQ, 1);
    }
123
    void rtc_isr (void)
125 {
      if ((RTC_CRL & RTC_CRL_SECF) != 0x00)
127     {
          /* The interrupt flag isn't cleared by hardware, we have to do it. */
129       rtc_clear_flag (RTC_SEC);
        }
131   if ((RTC_CRL & RTC_CRL_ALRF) != 0x00)
        {
133       rtc_clear_flag (RTC_ALR);
          rtc_set_alarm_time (rtc_get_counter_val () + temps_mesure);
135       my_val = 1;

137     }
    }
139
    void jmf_printc (char c)
141 {
      cc = (c & 0xf0) >> 4;
143   if (cc < 10)
        usart_send_blocking (USART1, cc + '0');
145   else
        usart_send_blocking (USART1, cc - 10 + 'A');
147   cc = (c & 0x0f);
      if (cc < 10)
149     usart_send_blocking (USART1, cc + '0');
      else
151     usart_send_blocking (USART1, cc - 10 + 'A');
    }
153
    void jmf_prints (short c)
155 {
      cc = (c & 0xf000) >> 12;
157   if (cc < 10)
        usart_send_blocking (USART1, cc + '0');
159   else
        usart_send_blocking (USART1, cc - 10 + 'A');
161   cc = (c & 0x0f00) >> 8;
      if (cc < 10)
163     usart_send_blocking (USART1, cc + '0');
      else
165     usart_send_blocking (USART1, cc - 10 + 'A');
      cc = (c & 0x00f0) >> 4;
167   if (cc < 10)
        usart_send_blocking (USART1, cc + '0');
169   else
        usart_send_blocking (USART1, cc - 10 + 'A');
```

```
171   cc = (c & 0x0f);
      if (cc < 10)
173     usart_send_blocking (USART1, cc + '0');
      else
175     usart_send_blocking (USART1, cc - 10 + 'A');
    }
177
    void jmf_printf (char *c)
179 {
      int k = 0;
181   do
        {
183       usart_send_blocking (USART1, c[k]);
          k++;
185     }
      while (c[k] != 0);
187 }

189 void usleep (int k)
    {
191   volatile int i;
      for (i = 0; i < k * 2; i++) /* Wait a bit. */
193     __asm__ ("nop");
    }
195
    void busy_low (void)
197 {
      int i;
199   i = 0;
      do
201     {
          i++;
203       b = gpio_port_read (GPIOC);
        }
205   while (((b & BUSY) == 0) && (i < MAXITER));
      if (i == MAXITER)
207     jmf_printf ("  timeout read: BUSY still low\r\n");
    }
209
    void busy_high (void)
211 {
      int i;
213   i = 0;
      do
215     {
          i++;
217       b = gpio_port_read (GPIOC);
          usleep (DELA);
219     }
      while (((b & BUSY) != 0) && (i < MAXITER));
221   if (i == MAXITER)
        jmf_printf ("  timeout read: BUSY still high\r\n");
223 }

225 int read_command (unsigned char *cmd)
    {
227   int j = 0, k = 0, hi = 0;
      // set as input for reading data bus
229   gpio_set_mode (GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO8 | GPIO9 | GPIO10 |
          GPIO11 | GPIO12 | GPIO13 | GPIO14 | GPIO15);
      gpio_set (GPIOC, SLCTIN); //1) mise a un de slctin
231
      busy_low ();        //
233
      b = (gpio_port_read (GPIOB) >> 8);
235   hi = (((int) b) & 0xff);
      //jmf_printf(" hi= ");
237   //jmf_printc((char)hi);

239   gpio_clear (GPIOC, SLCTIN); // strobe pulse
```

```
    usleep (DELA);
241  gpio_set (GPIOC, STROBE + INIT + SLCTIN); //
    busy_high ();
243
    b = (gpio_port_read (GPIOB) >> 8);
245  k = (hi << 8) + (((int) b) & 0xff);
    gpio_clear (GPIOC, SLCTIN);
247  usleep (DELA);     // strobe pulse
    gpio_set (GPIOC, SLCTIN);
249  usleep (DELA);
    gpio_clear (GPIOC, INIT);
251
    if (cmd[2] != 101)
253    if (k > 5)
        {
255  jmf_printf (" k might be too big ");
    jmf_prints ((short) k);
257  k = 5;
        }
259
    j = 0;
261  do
      {
263        busy_low ();

265        b = (gpio_port_read (GPIOB) >> 8);
        if (cmd[2] == 101)
267  bigtab[j] = b;

269        gpio_clear (GPIOC, SLCTIN); // strobe pulse
        usleep (DELA);
271        gpio_set (GPIOC, SLCTIN + INIT);
        j++;
273
        busy_high ();
275
        b = (gpio_port_read (GPIOB) >> 8);
277        if ((cmd[2] == 101))
    bigtab[j] = b;
279
        gpio_clear (GPIOC, SLCTIN); // strobe pulse
281        usleep (DELA);
        gpio_set (GPIOC, SLCTIN);
283        gpio_clear (GPIOC, INIT);
        j++;
285
      }
287  while (j < k);
    gpio_set_mode (GPIOB, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
289      GPIO8 | GPIO9 | GPIO10 | GPIO11 | GPIO12 | GPIO13 | GPIO14
      | GPIO15);
291  gpio_clear (GPIOC, SLCTIN);

293  if (b == 254)
      jmf_printf (" OK\r\n");
295  else
      jmf_printf (" rien\r\n");
297
    jmf_printf (" commande ");
299  jmf_printc (cmd[2]);
    jmf_printf ("\r\n");
301  if (cmd[2] == 101)
      return (k);
303  else
      return (0);
305}

307void write_command (unsigned char *cmd)
{
309  int j = 0;
```

```
    jmf_printf ("Write: ");
311   do
      {
313     jmf_printc ((char) cmd[j]);
        gpio_port_write (GPIOB, (cmd[j] << 8)); // 1)envoie des commandes
315     usleep (DELA);
        gpio_clear (GPIOC, STROBE); // 2)strobe pulse
317     busy_low ();    //3) busy high, orignie low
        gpio_set (GPIOC, INIT + STROBE);  // strobe pulse  4) init    ///+strobe
319     busy_high ();   //5) busy low (origine high)
        gpio_clear (GPIOC, INIT); //6) init a l'etat bas
321     j++;
      }
323   while (j < (cmd[0] * 256 + cmd[1] + 2));  // boucle en fonction du nombre dans cmd 2
  }
325
  inline void put_char (uint32_t USART, unsigned char data)
327{
    send_char (USART, data);
329}

331void PWR_EnterSTANDBYMode (void)
  {
333   jmf_printf ("\r\n");
    jmf_printf ("bienvenue dans le mode veille\r\n");
335
    /* Set SLEEPDEEP bit of Cortex System Control Register */
337   SCB_SCR |= SCB_SCR_SLEEPDEEP;
    /* Select STANDBY mode */
339   PWR_CR |= PWR_CR_PDDS;
    /* Clear Wake-up flag */
341   PWR_CR |= PWR_CR_CWUF;
    while ((PWR_CSR & 1) == 0x00);
343   __asm volatile ("WFI");
  }
345
  void BKP (void)
347{

349   //Power Control register enable
    PWR_CR |= PWR_CR_DBP;
351
    // variable nv dans backup data register
353   nv = BKP_DR1;
    nv++;
355
    cc = (nv & 0xf000) >> 12;
357   if (cc < 10) l[0] = (cc + '0');
    else l[0] = (cc - 10 + 'A');
359
    cc = (nv & 0x0f00) >> 8;
361   if (cc < 10) l[1] = (cc + '0');
    else l[1] = (cc - 10 + 'A');
363
    cc = (nv & 0x00f0) >> 4;
365   if (cc < 10) l[2] = (cc + '0');
    else l[2] = (cc - 10 + 'A');
367
    cc = (nv & 0x0f);
369   if (cc < 10) l[3] = (cc + '0');
    else l[3] = (cc - 10 + 'A');
371   BKP_DR1 = nv;

373   jmf_printf ("variable nv : ");
    jmf_prints ((short) nv);
375   jmf_printf ("\r\n");

377}

379void RTC_GetTime (volatile u32 * THH, volatile u32 * TMM, volatile u32 * TSS) //MOI
```

```
    {
381   volatile u32 tmp;
      tmp = rtc_get_counter_val ();
383   *THH = (tmp / 3600) % 24;
      *TMM = (tmp / 60) % 60;
385   *TSS = tmp % 60;
    }
387
    void temps (void)
389 {
      volatile u32 heure, minute, seconde, r, tmp;
391   C tmp = (heure / 10);
      r = (heure - tmp * 10);
393   x[0] = tmp + '0';
      x[1] = r + '0';
395   x[2] = ':';
      usart_send_blocking (USART1, x[0]);
397   usart_send_blocking (USART1, x[1]);
      usart_send_blocking (USART1, x[2]);
399
      tmp = minute / 10;
401   r = (minute - tmp * 10);
      x[3] = tmp + '0';
403   x[4] = r + '0';
      x[5] = ':';
405   usart_send_blocking (USART1, x[3]);
      usart_send_blocking (USART1, x[4]);
407   usart_send_blocking (USART1, x[5]);

409   tmp = seconde / 10;
      r = (seconde - tmp * 10);
411   x[6] = tmp + '0';
      x[7] = r + '0';
413
      usart_send_blocking (USART1, x[6]);
415   usart_send_blocking (USART1, x[7]);
      jmf_printf ("\r\n");
417 }

419 int main (void)
    {
421   int k, w;
      unsigned char cmd[10];
423   volatile u32 j = 0, c = 0;

425   clock_setup ();
      gpio_setup ();
427   usart_setup ();

429   // Init ctrl :    ;
      gpio_port_write (GPIOC, STROBE);
431
      jmf_printf ("\r\n");
433   jmf_printf ("\r\nHit any key\r\n");
      //k = usart_recv_blocking(USART1);
435   jmf_printf ("OK, starting\r\n");
      jmf_printf ("\r\n");
437
      my_val = 0;
439   /*
       * If the RTC is pre-configured just allow access, don't reconfigure.
441    * Otherwise enable it with the LSE as clock source and 0x7fff as
       * prescale value.
443    */
      rtc_auto_awake (LSE, 0x7fff);
445
      /* Setup the RTC interrupt. */
447   nvic_setup ();

449   /* Enable the RTC interrupt to occur off the SEC flag. */
```

```
    rtc_interrupt_enable (RTC_ALR);
451 rtc_enable_alarm ();
    rtc_set_alarm_time (rtc_get_counter_val () + temps_mesure);
453
    while (1)
455   {
        BKP ();       // increment de nv
457     temps ();
        while (my_val == 0);
459     jmf_printf ("\r\n");
        usart1_put_string ("interruption en cours \r\n");
461
        jmf_printf ("\r\n");
463     SD_Present = ouvre_sd ();

465     for (k = 0; k < 100; k++)
    {
467   gpio_set (GPIOC, INIT);
      usleep (5);    // 1.57 ms
469   gpio_clear (GPIOC, INIT);
      usleep (5);
471 }

473     usleep (10);
        b = gpio_port_read (GPIOC);
475     if ((b & BUSY) != 0)
    jmf_printf ("Busy != 0 at init\r\n");
477
        cmd[0] = 0; cmd[1] = 1; cmd[2] = 16;     // RESET_P
479     write_command (cmd);
        read_command (cmd);
481     cmd[0] = 0; cmd[1] = 1; cmd[2] = 16;     // RESET_P
        write_command (cmd);
483     read_command (cmd);
        cmd[0] = 0; cmd[1] = 2; cmd[2] = 2; cmd[3] = 1; // S_STACKS //en fonction du type→
            ↪ de relever (mvt ou non)
485     write_command (cmd);
        read_command (cmd);
487     cmd[0] = 0; cmd[1] = 3; cmd[2] = 1; cmd[3] = 20; cmd[4] = 5;// S_SAMP definir le →
            ↪nombre d'echantillon
        write_command (cmd);
489     read_command (cmd);
        cmd[0] = 0; cmd[1] = 3; cmd[2] = 4; cmd[3] = 61; cmd[4] = 0;// S_FREQ choix de la→
            ↪ frequence.
491     write_command (cmd);
        read_command (cmd);
493     cmd[0] = 0; cmd[1] = 3; cmd[2] = 20; cmd[3] = 1; cmd[4] = 0;// S_SUM bip chaque →
            ↪acquisition ou pour 80% de memoire tampon
        write_command (cmd);
495     read_command (cmd);
        cmd[0] = 0; cmd[1] = 3; cmd[2] = 3; cmd[3] = 100; cmd[4] = 0;// S_SIGPOS
497     write_command (cmd);
        read_command (cmd);
499     cmd[0] = 0; cmd[1] = 1; cmd[2] = 12;     // START
        write_command (cmd);
501     read_command (cmd);

503     jmf_printf ("\r\n");
        jmf_printf ("Trace ?\r\n");
505     cmd[0] = 0; cmd[1] = 1; cmd[2] = 14;     // TRIG
        write_command (cmd);
507     read_command (cmd);
        cmd[0] = 0; cmd[1] = 1; cmd[2] = 101;    // GMT
509     write_command (cmd);
        w = read_command (cmd);
511
        jmf_printf ("Nbre de points : ");
513     jmf_prints ((short) w);
        jmf_printf ("\r\n");
515
```

```
        if (w > TAILMAX)
517  w = TAILMAX;
        for (k = 0; k < w; k++)
519  jmf_printc (bigtab[k]);

521     if (SD_Present == 1)

523  if (file_fopen (&file, &efs.myFs, "GPR/DATA.TXT", 'a') != 0)
       {
525      comm_puts ("# Couldn't open file DATA.TXT for appending\\n");
         if (file_fopen (&file, &efs.myFs, "GPR/DATA.TXT", 'w') != 0)
527        {
       comm_puts ("# Couldn't open file for writing\r\n");
529    SD_Present = 0;
           }
531    }
        if (SD_Present == 1)
533  {
       // Permet de mettre les 4 caracteres de la taille au debut de la trame
535    b = (w & 0xf000) >> 12;
       if (b < 10)
537      b = b + '0';
       else
539      b = b + 'A' - 10;
       file_write (&file, 1, &b);
541    b = (w & 0x0f00) >> 8;
       if (b < 10)
543      b = b + '0';
       else
545      b = b + 'A' - 10;
       file_write (&file, 1, &b);
547    b = (w & 0x00f0) >> 4;
       if (b < 10)
549      b = b + '0';
       else
551      b = b + 'A' - 10;
       file_write (&file, 1, &b);
553    b = (w & 0x000f);
       if (b < 10)
555      b = b + '0';
       else
557      b = b + 'A' - 10;
       file_write (&file, 1, &b);
559
       // pour ecrire la trame sur la carte sd
561
       for (k = 0; k < w; k++)
563      {
           b = (bigtab[k] & 0xf0) >> 4;
565        if (b < 10)
       b = b + '0';
567        else
       b = b + 'A' - 10;
569        file_write (&file, 1, &b);
           b = (bigtab[k] & 0x0f);
571        if (b < 10)
       b = b + '0';
573        else
       b = b + 'A' - 10;
575        file_write (&file, 1, &b);
         }
577    file_write (&file, 2, "\r\n");
       file_fclose (&file);
579    fs_umount (&efs.myFs);
     }
581     else
     {
583    comm_puts ("Erreur ecriture\r\n");
       SD_Present = 0;
585  }
```

```
587
        put_char (USART1, '\n');
589     put_char (USART1, '\r');
        my_val = 0;
591     PWR_EnterSTANDBYMode ();
      }
593 }

595 //          Ouverture de la SD
    char ouvre_sd ()
597 {
      char err;
599   err = efs_init (&efs, 0);
      if ((err) != 0)
601     {
          jmf_printf ("# SD Error\r\n");
603       return (0);
        }
605   else
        {
607       jmf_printf ("# SD ok\r\n");
          SD_Present = 1;
609
          //Creation du repertoire
611       if (mkdir (&efs.myFs, "GPR") == 0)  // si il n'existe pas
      jmf_printf ("# New data directory created.\r\n");
613       else
      jmf_printf ("# unable to create new dir (already existing ?)\r\n");
615
          // il FAUT utiliser 'a' pour append, sinon erreur quand le fichier existe deja
617       // et qu'on utilise 'w'
          if (file_fopen (&file, &efs.myFs, "GPR/DATA.TXT", 'a') != 0)
619   {
        jmf_printf ("# Couldn't open file DATA.TXT for appending\\n");
621     if (file_fopen (&file, &efs.myFs, "GPR/DATA.TXT", 'w') != 0)
          {
623         jmf_printf ("# Couldn't open file for writing\r\n");
            SD_Present = 0;
625       }
      }
627       else
      jmf_printf ("#File Open OK\r\n");
629       if (SD_Present == 1)
      {
631     file_write (&file, 8, x);
        file_write (&file, 2, "  ");
633     file_write (&file, 4, l);
        file_write (&file, 2, "\r\n");
635     jmf_printf ("#Data written\r\n");
        jmf_printf ("\r\n");
637   }
          else
639   jmf_printf ("#Failed to write data\r\n");
          file_fclose (&file);
641       fs_umount (&efs.myFs);  // ecrit reellement sur le disque
        }
643   return (SD_Present);
    }
```

12.3 Gestion de l'énergie

Les pages de la documentation du Cortex-M3 de ST (documentation référencée RM0008) concernant les modes veille et la gestion d'énergie sont copiées sur les Fig. 12.3-12.3.

12.4 Programme Interface graphique

```
 #include <libopencm3/stm32/f1/rcc.h>
2#include <libopencm3/stm32/f1/gpio.h>
 #include <libopencm3/stm32/f1/rtc.h>
```

Entering Standby mode

Refer to *Table 15* for more details on how to enter Standby mode.

In Standby mode, the following features can be selected by programming individual control bits:

- Independent watchdog (IWDG): the IWDG is started by writing to its Key register or by hardware option. Once started it cannot be stopped except by a reset. See *Section 19.3: IWDG functional description* in *Section 19: Independent watchdog (IWDG)*.
- real-time clock (RTC): this is configured by the RTCEN bit in the Backup domain control register (RCC_BDCR)
- Internal RC oscillator (LSI RC): this is configured by the LSION bit in the Control/status register (RCC_CSR).
- External 32.768 kHz oscillator (LSE OSC): this is configured by the LSEON bit in the Backup domain control register (RCC_BDCR)

Exiting Standby mode

The microcontroller exits the Standby mode when an external reset (NRST pin), an IWDG reset, a rising edge on the WKUP pin or the rising edge of an RTC alarm occurs (see *Figure 179: RTC simplified block diagram*). All registers are reset after wakeup from Standby except for *Power control/status register (PWR_CSR)*.

After waking up from Standby mode, program execution restarts in the same way as after a Reset (boot pins sampling, vector reset is fetched, etc.). The SBF status flag in the *Power control/status register (PWR_CSR)* indicates that the MCU was in Standby mode.

Refer to *Table 15* for more details on how to exit Standby mode.

Table 15. Standby mode

Standby mode	Description
Mode entry	WFI (Wait for Interrupt) or WFE (Wait for Event) while: – Set SLEEPDEEP in Cortex™-M3 System Control register – Set PDDS bit in Power Control register (PWR_CR) – Clear WUF bit in Power Control/Status register (PWR_CSR)
Mode exit	WKUP pin rising edge, RTC alarm event's rising edge, external Reset in NRST pin, IWDG Reset.
Wakeup latency	Reset phase

I/O states in Standby mode

In Standby mode, all I/O pins are high impedance except:

- Reset pad (still available)
- TAMPER pin if configured for tamper or calibration out
- WKUP pin, if enabled

Bit 8 **DBP**: Disable backup domain write protection.
In reset state, the RTC and backup registers are protected against parasitic write access.
This bit must be set to enable write access to these registers.
0: Access to RTC and Backup registers disabled
1: Access to RTC and Backup registers enabled
Note: If the HSE divided by 128 is used as the RTC clock, this bit must remain set to 1.

Bits 7:5 **PLS[2:0]:** PVD level selection.
These bits are written by software to select the voltage threshold detected by the Power Voltage Detector
000: 2.2V
001: 2.3V
010: 2.4V
011: 2.5V
100: 2.6V
101: 2.7V
110: 2.8V
111: 2.9V
Note: Refer to the electrical characteristics of the datasheet for more details.

Bit 4 **PVDE:** Power voltage detector enable.
This bit is set and cleared by software.
0: PVD disabled
1: PVD enabled

Bit 3 **CSBF**: Clear standby flag.
This bit is always read as 0.
0: No effect
1: Clear the SBF Standby Flag (write).

Bit 2 **CWUF:** Clear wakeup flag.
This bit is always read as 0.
0: No effect
1: Clear the WUF Wakeup Flag **after 2 System clock cycles**. (write)

Bit 1 **PDDS**: Power down deepsleep.
This bit is set and cleared by software. It works together with the LPDS bit.
0: Enter Stop mode when the CPU enters Deepsleep. The regulator status depends on the LPDS bit.
1: Enter Standby mode when the CPU enters Deepsleep.

Bit 0 **LPDS:** Low-power deepsleep.
This bit is set and cleared by software. It works together with the PDDS bit.
0: Voltage regulator on during Stop mode
1: Voltage regulator in low-power mode during Stop mode

5.4.2 Power control/status register (PWR_CSR)

Address offset: 0x04

Reset value: 0x0000 0000 (not reset by wakeup from Standby mode)

Additional APB cycles are needed to read this register versus a standard APB read.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved							EWUP	Reserved					PVDO	SBF	WUF
							rw						r	r	r

Bits 31:9 Reserved, must be kept at reset value.

Bit 8 **EWUP:** Enable WKUP pin

This bit is set and cleared by software.

0: WKUP pin is used for general purpose I/O. An event on the WKUP pin does not wakeup the device from Standby mode.

1: WKUP pin is used for wakeup from Standby mode and forced in input pull down configuration (rising edge on WKUP pin wakes-up the system from Standby mode).

Note: This bit is reset by a system Reset.

Bits 7:3 Reserved, must be kept at reset value.

Bit 2 **PVDO:** PVD output

This bit is set and cleared by hardware. It is valid only if PVD is enabled by the PVDE bit.

0: V_{DD}/V_{DDA} is higher than the PVD threshold selected with the PLS[2:0] bits.

1: V_{DD}/V_{DDA} is lower than the PVD threshold selected with the PLS[2:0] bits.

Note: The PVD is stopped by Standby mode. For this reason, this bit is equal to 0 after Standby or reset until the PVDE bit is set.

Bit 1 **SBF:** Standby flag

This bit is set by hardware and cleared only by a POR/PDR (power on reset/power down reset) or by setting the CSBF bit in the *Power control register (PWR_CR)*

0: Device has not been in Standby mode

1: Device has been in Standby mode

Bit 0 **WUF:** Wakeup flag

This bit is set by hardware and cleared only by a POR/PDR (power on reset/power down reset) or by setting the CWUF bit in the *Power control register (PWR_CR)*

0: No wakeup event occurred

1: A wakeup event was received from the WKUP pin or from the RTC alarm

Note: An additional wakeup event is detected if the WKUP pin is enabled (by setting the EWUP bit) when the WKUP pin level is already high.

Doc ID 13902 Rev 14 77/1096

```
4#include <libopencm3/stm32/usart.h>
 #include <libopencm3/stm32/pwr.h>
6#include <stdlib.h>
 #include "comm.h"
8#include "usart.h"
```

```
#include <scb.h>
10
#define MAXITER 0x7ffff
12#define DELA    10
#define STROBE  0x01     // B0
14#define INIT  0x04
#define SLCTIN  0x08
16
// CU output signals
18#define BUSY  0x02    // instead of 0x20

20#define TAILMAX 2700

22#define send_char(USART, c)  do { \
   USART_DR(USART) = (c&USART_DR_MASK); \
24   while ((USART_SR(USART) & USART_SR_TXE) == 0); \
   } while(0)
26

28//#define jmf_debug

30char bigtab[TAILMAX];

32int b;
unsigned char cmd[10];
34
/* Set STM32 to 72 MHz. */
36void clock_setup (void)
{
38   rcc_clock_setup_in_hse_8mhz_out_72mhz ();

40   /* Enable GPIOC clock. */
   rcc_peripheral_enable_clock (&RCC_APB2ENR,
42               RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN |
               RCC_APB2ENR_IOPCEN);
44
   /* Enable clocks for GPIO port B (for GPIO_USART3_TX) and USART3. */
46   rcc_peripheral_enable_clock (&RCC_APB2ENR, RCC_APB2ENR_USART1EN);
}
48
void usart_setup (void)
50{
   /* Setup GPIO pin GPIO_USART1_TX. */
52   gpio_set_mode (GPIOA, GPIO_MODE_OUTPUT_10_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
54
   /* Setup UART parameters. */
56   usart_set_baudrate (USART1, 57600);
   usart_set_databits (USART1, 8);
58   usart_set_stopbits (USART1, USART_STOPBITS_1);
   usart_set_mode (USART1, USART_MODE_TX_RX);
60   usart_set_parity (USART1, USART_PARITY_NONE);
   usart_set_flow_control (USART1, USART_FLOWCONTROL_NONE);
62
   /* Finally enable the USART. */
64   usart_enable (USART1);
}
66
void gpio_setup (void)
68{
   /* Set GPIO12 (in GPIO port C) to 'output push-pull'. */
70   gpio_set_mode (GPIOB, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
      GPIO8 | GPIO9 | GPIO10 | GPIO11 | GPIO12 | GPIO13 | GPIO14
72      | GPIO15);
   gpio_set_mode (GPIOC, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
74      INIT | STROBE | SLCTIN | GPIO7);
   gpio_set_mode (GPIOC, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, BUSY);
76}

78
```

```
void jmf_printc (char c)
{
  char cc;
  cc = (c & 0xf0) >> 4;
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
  cc = (c & 0x0f);
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
}

void jmf_prints (short c)
{
  char cc;
  cc = (c & 0xf000) >> 12;
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
  cc = (c & 0x0f00) >> 8;
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
  cc = (c & 0x00f0) >> 4;
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
  cc = (c & 0x0f);
  if (cc < 10)
    usart_send_blocking (USART1, cc + '0');
  else
    usart_send_blocking (USART1, cc - 10 + 'A');
}

void jmf_printf (char *c)
{
  int k = 0;
  do
    {
      usart_send_blocking (USART1, c[k]);
      k++;
    }
  while (c[k] != 0);
}

void usleep (int k)
{
  volatile int i;
  for (i = 0; i < k * 2; i++) /* Wait a bit. */
    __asm__ ("nop");
}


void busy_low (void)
{
  int i;
  i = 0;
  do
    {
      i++;
      b = gpio_port_read (GPIOC);
    }       // apres le point virgule usleep(DELA)
  while (((b & BUSY) == 0) && (i < MAXITER));
  /*if (i == MAXITER)
```

```
      jmf_printf("  timeout read: BUSY still low\r\n"); */
150}

152void busy_high (void)
{
154  int i;
  i = 0;
156  do
    {
158      i++;
      b = gpio_port_read (GPIOC);
160      usleep (DELA);
    }
162  while (((b & BUSY) != 0) && (i < MAXITER));
  /*if (i == MAXITER)
164      jmf_printf("  timeout read: BUSY still high\r\n"); */
}
166
int read_command (unsigned char *cmd)
168{
  int j = 0, k = 0, hi = 0;
170  char b;
  // input for reading data bus
172  gpio_set_mode (GPIOB, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO8 | GPIO9 | GPIO10 |→
      ↪ GPIO11 | GPIO12 | GPIO13 | GPIO14 | GPIO15);
  gpio_set (GPIOC, SLCTIN); //1) mise a un de slctin
174
  busy_low ();       //
176
  b = (gpio_port_read (GPIOB) >> 8);
178  hi = (((int) b) & 0xff);
#ifdef jmf_debug
180  jmf_printf (" hi= ");
  jmf_printc ((char) hi);
182#endif
  gpio_clear (GPIOC, SLCTIN); // strobe pulse
184  usleep (DELA);
  gpio_set (GPIOC, STROBE + INIT + SLCTIN); //
186  busy_high ();

188  b = (gpio_port_read (GPIOB) >> 8);
  k = (hi << 8) + (((int) b) & 0xff);
190#ifdef jmf_debug
  jmf_printf (" hi+lo= ");
192  jmf_prints ((short) k);
  jmf_printf ("\r\n");
194#endif
  gpio_clear (GPIOC, SLCTIN);
196  usleep (DELA);     // strobe pulse
  gpio_set (GPIOC, SLCTIN);
198  usleep (DELA);
  gpio_clear (GPIOC, INIT);
200
  if (cmd[2] != 101)
202    if (k > 5)
      {
204  //jmf_printf(" k might be too big ");
  //jmf_prints((short)k);
206  k = 5;
      }
208
  j = 0;
210  do
    {
212      busy_low ();

214      b = (gpio_port_read (GPIOB) >> 8);
      if (cmd[2] == 101)
216  bigtab[j] = b;
```

```
218       gpio_clear (GPIOC, SLCTIN); // strobe pulse
          usleep (DELA);
220       gpio_set (GPIOC, SLCTIN + INIT);
          j++;
222
          busy_high ();
224
          b = (gpio_port_read (GPIOB) >> 8);
226       if ((cmd[2] == 101))
      bigtab[j] = b;
228
          gpio_clear (GPIOC, SLCTIN); // strobe pulse
230       usleep (DELA);
          gpio_set (GPIOC, SLCTIN);
232       gpio_clear (GPIOC, INIT);
          j++;
234
        }
236   while (j < k);
      gpio_set_mode (GPIOB, GPIO_MODE_OUTPUT_10_MHZ, GPIO_CNF_OUTPUT_PUSHPULL,
238       GPIO8 | GPIO9 | GPIO10 | GPIO11 | GPIO12 | GPIO13 | GPIO14
          | GPIO15);
240   gpio_clear (GPIOC, SLCTIN);
    #ifdef jmf_debug
242   if (b == 254)
        jmf_printf ("OK\r\n");
244   else
        jmf_printf (" rien\r\n");
246   jmf_printf (" commande ");
      jmf_printc (cmd[2]);
248   jmf_printf ("\r\n");
    #endif
250   if (cmd[2] == 101)
        return (k);
252   else
        return (0);
254 }

256 void
    write_command (unsigned char *cmd)
258 {
      int j = 0;
260 #ifdef jmf_debug
      jmf_printf ("Write: ");
262 #endif
      do
264     {
    #ifdef jmf_debug
266       jmf_printc ((char) cmd[j]);
    #endif
268       gpio_port_write (GPIOB, (cmd[j] << 8)); // 1)envoie des commandes
          usleep (DELA);
270
          gpio_clear (GPIOC, STROBE); // 2)strobe pulse
272       busy_low ();                // 3) busy high, orignie low
          gpio_set (GPIOC, INIT + STROBE);  // strobe pulse  4) init    ///+strobe
274       busy_high ();   //5) busy low (origine high)
          gpio_clear (GPIOC, INIT); //6) init a l'etat bas
276       j++;
        }
278   while (j < (cmd[0] * 256 + cmd[1] + 2));  // boucle en fonction du nombre dans cmd 2
    }
280
    void
282 put_string (uint32_t USART, char *string)
    {
284   int i;
      for (i = 0; string[i] != 0; i++)
286     send_char (USART, string[i]);
    }
```

```
288
    inline void
290 put_char (uint32_t USART, unsigned char data)
    {
292   send_char (USART, data);
    }
294


296 void
    init (void)
298 {
      cmd[0] = 0; cmd[1] = 1; cmd[2] = 16;        // RESET_P
300   write_command (cmd);
      read_command (cmd);
302   cmd[0] = 0; cmd[1] = 1; cmd[2] = 16;        // RESET_P
      write_command (cmd);
304   read_command (cmd);
      cmd[0] = 0; cmd[1] = 2; cmd[2] = 2; cmd[3] = 1; // S_STACKS en fonction du type de →
          ↪relever (mvt ou non)
306   write_command (cmd);
      read_command (cmd);
308   cmd[0] = 0; cmd[1] = 3; cmd[2] = 20; cmd[3] = 1; cmd[4] = 0;  // S_SUM active bip a →
          ↪chaque acq. ou pour 80% de memoire tampon
      write_command (cmd);
310   read_command (cmd);
    }
312
    void setting (unsigned char bl_sample, unsigned char bh_sample,
314     unsigned char bl_freq, unsigned char bh_freq,
        unsigned char bl_sigpos, unsigned char bh_sigpos)
316 {
    #ifdef jmf_debug
318   jmf_printf ("bl_sample est de : "); jmf_printc ((char) bl_sample); jmf_printf ("\r\n→
          ↪");
      jmf_printf ("bh_sample est de : "); jmf_printc ((char) bh_sample); jmf_printf ("\r\n→
          ↪");
320   jmf_printf ("bl_freq est de : ");   jmf_printc ((char) bl_freq);   jmf_printf ("\r\n→
          ↪");
      jmf_printf ("bh_freq est de : ");   jmf_printc ((char) bh_freq);   jmf_printf ("\r\n→
          ↪");
322   jmf_printf ("bl_sigpos est de : "); jmf_printc ((char) bl_sigpos); jmf_printf ("\r\n→
          ↪");
      jmf_printf ("bh_sigpos est de : "); jmf_printc ((char) bh_sigpos); jmf_printf ("\r\n→
          ↪");
324 #endif

326   cmd[0] = 0; cmd[1] = 3; cmd[2] = 1; cmd[3] = bl_sample; cmd[4] = bh_sample; // S_SAMP→
          ↪ nombre d'echantillon
      write_command (cmd);
328   read_command (cmd);

330   cmd[0] = 0; cmd[1] = 3; cmd[2] = 4; cmd[3] = bl_freq; cmd[4] = bh_freq;     // S_FREQ→
          ↪ frequence.
      write_command (cmd);
332   read_command (cmd);

334   cmd[0] = 0; cmd[1] = 3; cmd[2] = 3; cmd[3] = bl_sigpos; cmd[4] = bh_sigpos; // →
          ↪S_SIGPOS
      write_command (cmd);
336   read_command (cmd);
    }
338
    int main (void)
340 {
      int k, w;
342   char b;
      unsigned char bl_sample, bh_sample, bl_freq, bh_freq, bl_sigpos, bh_sigpos;
344
      clock_setup ();
346   gpio_setup ();
```

```
    usart_setup ();
348
    // Init ctrl :    ;
350 gpio_port_write (GPIOC, STROBE);
#ifdef jmf_debug
352 jmf_printf ("\r\nHit any key\r\n");
    k = usart_recv_blocking (USART1);
354 jmf_printf ("\r\nOK, starting\r\n");
#endif
356 for (k = 0; k < 100; k++)
      {
358     gpio_set (GPIOC, INIT);
        usleep (5);   // 1.57 ms
360     gpio_clear (GPIOC, INIT);
        usleep (5);
362   }

364 usleep (10);
    b = gpio_port_read (GPIOC);
366 if ((b & BUSY) != 0)
      jmf_printf ("Busy != 0 at init\r\n");
368
    init ();
370 jmf_printf ("!");

372 while (1)
      {
374     b = usart_recv_blocking (USART1);

376     if (b == '!')
    {
378   cmd[0] = 0;
      cmd[1] = 1;
380   cmd[2] = 13;    // STOP
      write_command (cmd);
382   read_command (cmd);
      bl_sample = usart_recv_blocking (USART1);
384   bh_sample = usart_recv_blocking (USART1);
      bl_freq = usart_recv_blocking (USART1);
386   bh_freq = usart_recv_blocking (USART1);
      bl_sigpos = usart_recv_blocking (USART1);
388   bh_sigpos = usart_recv_blocking (USART1);

390   setting (bl_sample, bh_sample, bl_freq, bh_freq, bl_sigpos,
         bh_sigpos);
392
      cmd[0] = 0; cmd[1] = 1; cmd[2] = 12;    // START
394   write_command (cmd);
      read_command (cmd);
396
      jmf_printf ("?");
398 }

400     if (b == '?')
    {
402#ifdef jmf_debug
      jmf_printf ("acquisition\r\n");
404#endif
      cmd[0] = 0; cmd[1] = 1; cmd[2] = 14;    // TRIG
406   write_command (cmd);
      read_command (cmd);
408   usleep (1);
      cmd[0] = 0; cmd[1] = 1; cmd[2] = 101;   // GMT
410   write_command (cmd);
      w = read_command (cmd);
412#ifdef jmf_debug
      jmf_printf ("Nbre de points : ");
414#endif
      jmf_prints ((short) w);
416
```

```
      if (w > TAILMAX)
418     w = TAILMAX;
      for (k = 0; k < w; k++)
420     {
          jmf_printc (bigtab[k]);
422     }

424   jmf_printf ("#");
    }
426   }
}
```

Références

[1] H.M. Jol, *Ground penetrating radar : theory and application*, Elsevier (2009)

[2] D.J. Daniels, *Ground penetrating radar*, IET Radar, Sonar, Navigation and Avionics Series 15, Institution of Engineerings and Technology (2004)

[3] A. Saintenoy, J.-M. Friedt, F. Tolle, É. Bernard, D. Laffly, C. Marlin, & M. Griselin, *High density coverage investigation of the Austre Lovénbreen (Svalbard) using Ground Penetrating Radar*, 6th International Workshop on Advanced Ground Penetrating Radar (IWAGPR), 2011 (22-24 June 2011, Aachen, Germany)

[4] J.W. Stockwell & J.K. Cohen, *The New SU Users Manual, Version 4.0*, Janvier 2008 disponible à `http://www.seismicunix.com/wiki/images/9/9f/SU_manual_600dpi_a4.pdf`. Les codes sources sont disponibles à `http://www.cwp.mines.edu/cwpcodes/`, avec un wiki d'exemples à `http://www.seismicunix.com/w/Main_Page`