Simulateur PDI: Document de déploiement (Sprint 1)

Rapport présenté au
Professeur Gilbert Arbez
Dans le cadre du cours de
Conception de systèmes en temps réel (SEG4545/CEG4566)

Équipe 6
Par Shanel Gauthier n° 7900514,
Mikaël Medeiros-do Couto n° 8408984,
Luan Phung n° 7846227,
Christian Savoie n° 8373100,
Abdelkarim Sellami n° 8070589 et
Amine Yousfi Charif n° 8013669

UNIVERSITÉ D'OTTAWA
Le 17 février 2010

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

Introduction

Dans le cadre du cours de conception de systèmes en temps réel, il est question de développer un logiciel qui « s'exécutera dans le simulateur PDI (panneau de distribution intelligent) et qui simule un nano réseau électrique conçu pour alimenter des appareils d'une maison avec l'énergie solaire » [1]. Les informations de la conception se trouvent dans le document de déploiement de base à l'exception de trois modules [1]. Ces trois modules sont « Task », « ADC » et « GPIO » et ont été développés dans le cadre du sprint 1. L'objectif du sprint 1 est « de faire le développement des modules logiciels de bas niveau qui interagissent avec le matériel pour faire la collection des données de signal, de la fonctionnalité d'analyse des données collectées, des modules pour contrôler les relais et un module de tâches pour tester des modules développés » [1]. Ceci dit, ce rapport est, en fait, la continuité du document de déploiement de bases en y présentant la conception des trois modules développés.

3.2 Conception du module Task

Le module « Task » gère quatre différentes tâches en utilisant un pseudo-noyau de code cyclique pour faire l'ordonnancement des tâches. Les tâches sont appelées en boucle sans l'utilisation d'interruption. En effet, une instruction dans une tâche teste un drapeau qui indique si un évènement s'est produit ou non. Il sera question de traiter de la conception des quatre tâches en décrivant la logique utilisée et les structures de données importantes.

3.2.1 Tâche : Gestion des relais

La première tâche appelée dans le code cyclique est la tâche « manageRelays_task ». Cette tâche s'occupe de gérer l'ouverture et la fermeture des relais. Il est important de noter que cette tâche est indépendante des autres. La Figure 1 présente le diagramme de machine d'états de la gestion de l'ouverture et de la fermeture des relais. Dans ce diagramme, il est possible de voir qu'au départ, tous les relais du côté source et du côté charge sont ouverts. On ferme un relai du côté source puis chaque relai du côté charge est fermé un à la fois. Une fois les trois relais charges fermés, on les ouvre un à la fois puis on ouvre le relai de la source. On exécute ces étapes pour chaque relai du côté source. On répète ce cycle d'ouverture et de fermeture. Il est important de noter que seulement un relai source est ouvert à la fois et que la fermeture de relais est séparée par un temps d'au moins 10 secondes.

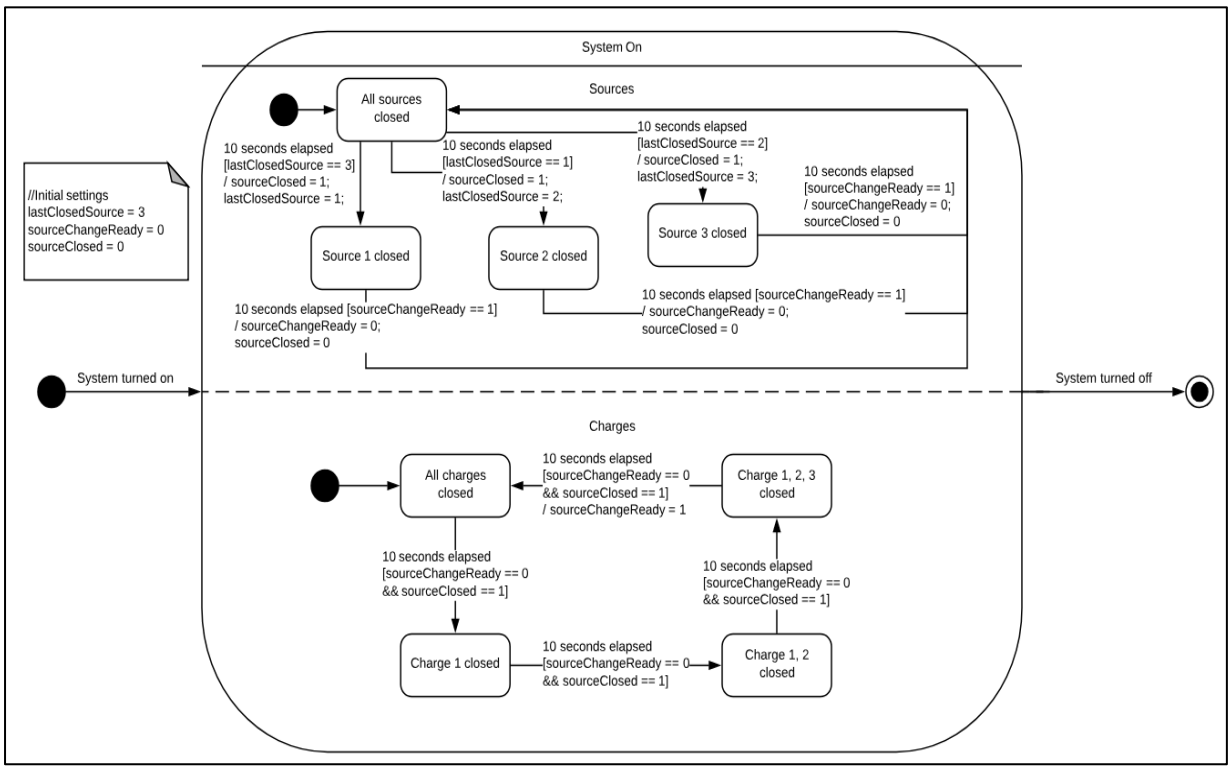

Figure 1- Machine d'états de la fermeture et l'ouverture des relais

L'implémentation de la machine d'états en langage C se trouve dans l'Annexe 1. Il est possible de voir que lorsqu'une transition n'est pas possible, la fonction « badState » est appelée. Ceci ne devrait jamais arriver. En utilisant la variable « timerHandle », il est possible de déterminer combien de temps c'est écoulé depuis la dernière fermeture ou ouverture de relais. Si 10 secondes se sont écoulées depuis la dernière transition, on modifie l'état de la machine à états et on change les relais concernés en même temps. Dans le cas où moins de 10 secondes se sont écoulées, on quitte la tâche pour permettre au pseudo-noyau de code cyclique d'exécuter une autre tâche. Afin de déterminer que 10 secondes se sont écoulées, un « DmTimer » qui est une horloge est utilisée et celui-ci est initialisé dans le constructeur par le transfert d'une adresse. En effet, étant déjà initialisé dans le module « main », le « DMTimer » est transféré par une adresse. En utilisant ce minuteur, il faut initialiser le compte actuel afin de pouvoir savoir quand 10 secondes se sont écoulées afin de faire la gestion des relais. Comme ce minuteur est initialisé avec un « prescalar » de 3 dans le module « main », cela veut dire que chaque tic d'horloge est équivalent à 0.66667 microsecondes. En faisant une équivalence, cela veut dire que pour 10 secondes, il faut environ 750 000 tics

s'écoulent. Donc, en utilisant une soustraction et le temps actuel noté par le compteur, il suffit de faire une soustraction et si jamais la différence est supérieure ou égale à 750 000, on peut procéder à la prochaine étape tout en mettant à jour le compteur actuel.

3.2.2 Tâche : Data Acquisition

La tâche « acquisition_task » est responsable de démarrer l'acquisition des données pour les 7 ports (3 ports du côté source, 3 ports du côté charge et le grid). L'Annexe 2 contient la structure des ports « PORT_DATA ». Chaque port possède trois différents drapeaux comme montrés ci-dessous:

```
bool acquireDone; /*!< set to true when data acquisition is complete */
bool startAnalysis; /*!< set to true indicates analysis can start */
bool analysisDone;    /*!< set to true when data analysis is complete */
```

Ces drapeaux sont utilisés pour déterminer quand démarrer l'acquisition et l'analyse des 7 ports. La Figure 2 contient le diagramme de machine d'états décrivant l'acquisition des données pour les 7 ports. Au départ, selon la machine d'états, on démarre l'acquisition pour les ports 1 (du côté source et du côté charge). Lorsque le drapeau « isAcquireDone » est à « true » pour les ports 1, on change la valeur du drapeau « startAnalysis » à « true » pour les ports 1, on démarre l'acquisition des données des ports 2 et on quitte l'état « Collecte Ports 1 » pour aller dans l'état « Collecte Ports 2 ». Il est important de noter que si les drapeaux isAcquireDone ne sont pas à « true » alors on quitte la tâche pour permettre au pseudo-noyau de code cyclique d'exécuter une autre tâche. Lorsque le drapeau « isAcquireDone » est à « true » pour les ports 2, on change la valeur du drapeau « startAnalysis » à « true » pour les ports 2, on démarre l'acquisition des données des ports 3 et du grid et on quitte l'état « Collecte Ports 3 » pour aller dans l'état « Collecte Ports 3 ». Il est important de noter que si les drapeaux isAcquireDone ne sont pas à « true » alors on quitte la tâche pour permettre au pseudo-noyau de code cyclique d'exécuter une autre tâche. Lorsque le drapeau « isAcquireDone » est à « true » pour les ports 3 et pour le grid, on change la valeur du drapeau « startAnalysis » à « true » pour les ports 3 et pour le grid et on quitte l'état « Collecte Ports 3 » pour aller dans l'état « En attente ». Il est important de noter que si les drapeaux isAcquireDone ne sont pas à « true » alors on quitte la tâche pour permettre au pseudo-noyau

de code cyclique d'exécuter une autre tâche. On reste dans l'état « En attente » jusqu'à ce que tous les drapeaux « isAcquireDone » pour les 7 ports deviennent de valeur « false ». Lorsqu'on est dans l'état « En attente », si tous les drapeaux « isAcquireDone » ne sont pas à « false » alors on quitte la tâche pour permettre au pseudo-noyau de code cyclique d'exécuter une autre tâche.

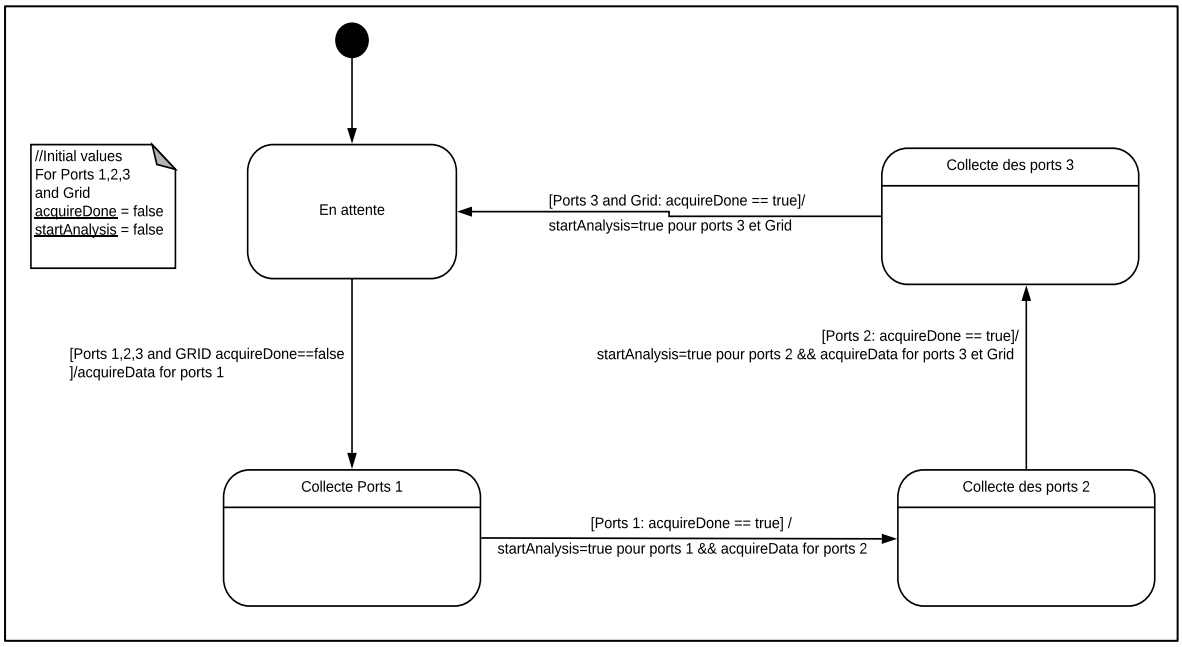

Figure 2- Machine d'états de l'acquisition des données

L'utilisation d'une déclaration « switch » est utilisée pour choisir l'état actuel dans le code. De plus, l'acquisition des données d'un port du côté source, d'un port du côté charge et du grid se font en parallèle. L'ensemble des ADC situés dans les ports sources est connecté à l'interface SPI0 du microcontrôleur, tandis que l'ensemble des ADC situés dans les ports de charge est connecté à l'interface SPI1 du microcontrôleur. Ceci permet donc de démarrer l'acquisition du côté source et charge simultanément. De plus, comme il y a seulement un grid, l'acquisition des données du grid se fait en dernier, en même temps que les troisièmes ports source et charge. Les fonctions nécessaires pour l'acquisition de données sont retrouvées dans le module « Acquisition.c ».

3.2.3 Tâche : Data Analysis

Cette tâche est responsable de démarrer l'analyse des ports et du grid. La tâche boucle à travers chaque port dans les trois groupes de ports (Source, Load, Grid) et démarre l'analyse dès qu'elle trouve un port ayant son drapeau « startAnalysis » à « true » et son drapeau « analysisDone » à « false ». Une fois l'analyse d'un port démarrée, la fonction retourne pour permettre au pseudo-noyau de code cyclique d'exécuter une autre tâche.

3.2.4 Tâche : Display Task

Cette tâche affiche les données scalaires (vrms, irms, realPower, reactivePower, powerFactor) sur la console UART lorsque les données sont analysées sur tous les ports. Ceci est détecté en bouclant à travers chaque port dans les trois groupes de ports (Source, Load, Grid) et en vérifiant que tous les ports actifs ont leur drapeau « analysisDone » à « true ». Une boucle passe à travers tous les ports une deuxième fois pour afficher les données à l'aide de la fonction uart_print(), puis pour mettre les drapeaux « acquireDone » et « analysisDone » à « false » pour chaque port.

L'affichage est fait au travers du module « uart.c ». Lors de l'initiation du module « task.c », on imprime le caractère spécial « VT100_CLR_SCR » pour vider l'écran au début de l'affichage. Alors, dans chaque cycle d'affichage, les étapes sont les suivantes :

- Imprimer le caractère spécial « VT100_HVHOME » pour mettre le curseur au début du terminal.
- Construire le tampon à être imprimé avec le module « uart.c » avec les étapes suivantes:
    - Ajouter l'entête de la table au buffer
    - Ajouter les valeurs des sources au buffer
    - Ajouter les valeurs des charges au buffer
    - Ajouter les valeurs du grid au buffer
- Le tampon est passé à la fonction « uart_print() »

3.7 Conception du module ADC

Le module ADC fournit la fonctionnalité permettant de collecter des données (signaux numérisés) du grid, c'est-à-dire deux signaux de tension (hot et neutre). Les interruptions (thread Hwi) sont utilisées pour remplir les tampons. Il sera question de traiter de la vue d'ensemble du matériel et de la vue d'ensemble du logiciel.

3.7.1 Vue d'ensemble du matériel

Le simulateur PDI contient deux types d'ADC (« Analog to Digital Converter »). Il y a ceux propres aux ports et un seul pour le grid dont le but est de collecter les données de celui-ci sous forme d'échantillons. L'ADC propre au grid a pour but de lancer une opération de collecte d'au moins 16.7 millisecondes qui est un cycle de 60 Hz. À la fin de chaque collecte, l'ADC aura stocké 1024 échantillons au total dont 512 échantillons de Vn (Neutral voltage) et 512 autres échantillons de Vh (Hot voltage). C'est à l'aide d'interruptions (Hwi thread) que les échantillons sont collectés sous forme analogue et ensuite stockés dans un tampon. Une fois la collection terminée le module ADC a pour but de mettre une variable à « true » afin de signer l'arrêt ou la fin de la collection d'échantillons. L'ADC utilise des mots de 16 bits pour transférer un échantillon de 12 bits. La Figure 3 présente la configuration de la carte mère. Le microcontrôleur « TI Satara AM335XQuatres » qui est installé dans la carte BeagleBone contient l'interface ADC.

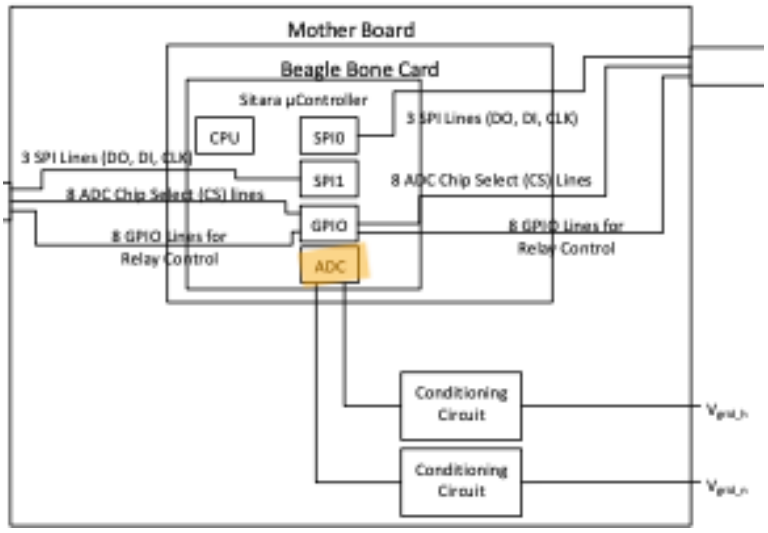

Figure 3 - Configuration de la carte mère où le ADC est surligné

La conception du module ADC repose sur la structure de données « ADC_REGS » qui est présentée dans l'Annexe 3. Cette structure contient l'ensemble des registres de l'interface ADC.

3.7.2 Vue d'ensemble du logiciel

Le module ADC se trouve dans la couche matérielle. La couche matérielle fournit, en fait comme son nom l'indique, les fonctionnalités pour manipuler le matériel. Dans la Figure 4, il est possible de voir que le module ADC utilise directement le module DMTimer pour que la collection de données se fasse pendant au moins 16.7ms. Le module « Data Acquisition » utilise directement les fonctions de l'ADC. Ce dernier se trouve dans la couche « Fonctionnalité » qui fournit, comme son nom l'indique, les fonctions de traitement du système. Il est aussi possible de voir, dans la Figure 4, que la tâche « Acquisition » qui se trouve dans la couche tâche utilise directement les fonctions du module « Data Acquisition ».

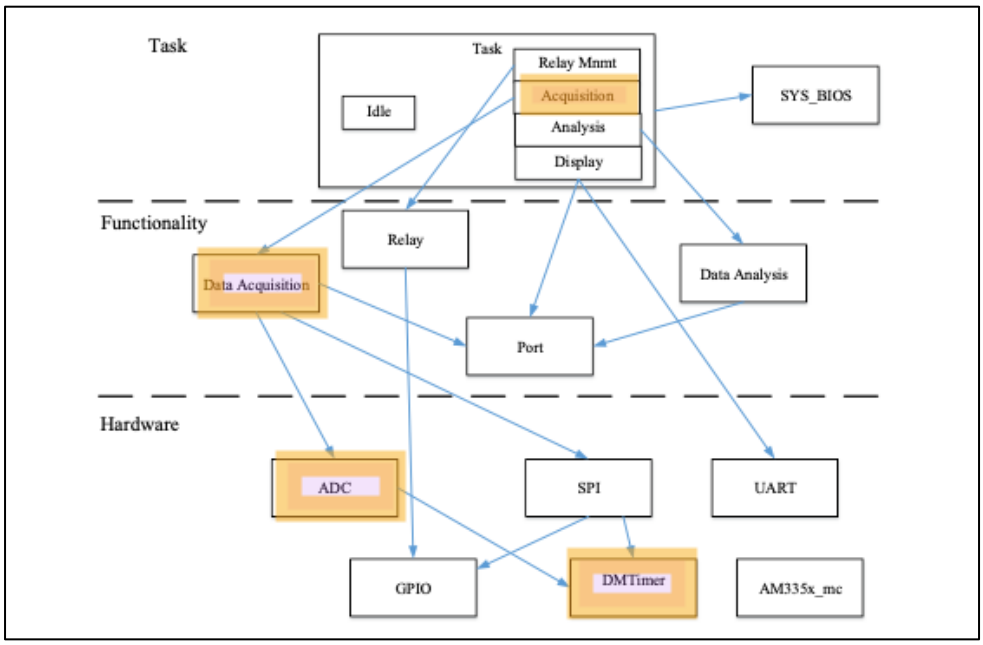

Figure 4 - Relation entre les modules nécessitant l'implémentation du module ADC

Comme l'ADC du GRID doit collecter des échantillons, certains paramètres doivent être définis localement dans le module ADC. On retrouve ci-dessous la structure locale utilisée pour stocker les données associées avec l'ADC :

```
//structure to store data associated with a ADC
typedef struct
{
   // Variables for use with ISR
   bool done; //Set to true, when finished collecting samples.
   bool *donePtr;  // for advising calling function - can be used for polling
   uint32_t numSamples; //number of samples to collect
   uint16_t *buffers; //pointer to buffers. start of each buffer is at address (buffers+sizeBuffer*BUF1),
(buffers+sizeBuffer*BUF2)etc.
   bool receivedFirst; // number of samples received for each group of signals.
   uint32_t *start_time;  // time data collection starts
   int numReceived;  // number of samples received for each group of signals.
   uint32_t *end_time;    // time data collection ends.
   int timerHandle; // for time stamps
   bool initialized;
} ADC;
```

L'utilisation d'une structure permet d'assurer une meilleure gestion des données.

3.7.2.1 Initialisation ADC

Avant de démarrer l'acquisition des données, l'initialisation de l'ADC doit être réalisée. Pour ce faire, il est nécessaire d'activer le module lui-même à l'aide de la méthode « activateModule (MC_ADC) » où « MC_ADC » pointe vers l'identificateur du matériel. Ensuite, l'initialisation de certaines variables de la structure locale ADC doit être réalisée dont « done » qui est utilisé comme un drapeau. Lorsque ce drapeau a la valeur « true » alors le ADC n'est pas occupé et lorsque la valeur est à « false », cela signifie qu'on ne peut commencer le démarrage de l'acquisition de données parce que le ADC est occupé.

Par la suite, il est primordial de configurer l'ADC. La configuration doit satisfaire plusieurs exigences qui sont :

1) Numériser les signaux sur AN0 (Vh) et AN1 (Vn) aux étapes 1 et 2 respectivement.

2) Utiliser l'horloge 24 MHz sans mise à l'échelle (1/24 microseconde = 41 2/3 nanosecondes)

3) La séquence de conversion comprend l'étape 1 avec un délai d'ouverture de 333 horloge suivie de l'étape 2 sans délai d'ouverture. Les deux étapes utilisent un délai d'échantillonnage simple.

4) Les deux étapes stockent les données de conversion dans FIFO0, de telle sorte que les données lues alternent entre les signaux, c'est-à-dire: Vh Vn Vh Vn et ainsi de suite.

5) Les interruptions sont générées sur le seuil FIFO0, configuré à un niveau de seuil de 32.

Pour satisfaire ces exigences, il est primordial d'écrire dans le registre « fifo0threshold » la valeur de 32 pour signaler que la valeur seuil est de bel et bien de 32 :

```
adcRegs->fifo0threshold = 32;
```

Par la suite, il faut mettre à 0 le deuxième bit du registre « irqenable_set » pour activer l'interruption lorsque la valeur seuil dans FIFO0 est de 32 :

```
adcRegs->irqenable_set |= FIFO0_THRESHOLD_BIT;
```

En d'autres mots, le ADC est configuré pour lancer une interruption lorsque le tampon a atteint sa valeur seuil.

3.7.2.2 Démarrage de l'acquisition des données du ADC

Lorsque le module « Data Acquisition » tente de démarrer l'acquisition du ADC, le drapeau de la structure ADC local est testé. L'acquisition peut commencer seulement si la valeur de ce drapeau est à « true » puisque l'ADC n'est pas occupé. Par la suite, il faut mettre la valeur du drapeau à « false » parce que l'acquisition va commencer sous peu. Il est important de réinitialiser le registre « irqstatus » en effacant tout évènement s'il y en a en attente :

```
adcRegs->irqstatus = FIFO0_THRESHOLD_BIT;
```

Il est alors question de démarrer la collecte des échantillons en mettant à 1 le bit 0 du registre « cntrl ». Ceci a pour effet de démarrer l'acquisition des échantillons.

```
adcRegs->cntrl |= ENABLEADC;
```

3.7.2.3 Gestion des interruptions

Une fois reçue, l'interruption est gérée par la fonction « adc_isr ». Lorsque l'interruption est lancée, cela signifie que la valeur seuil de 32 échantillons a été atteint dans FIFO0 et que le registre « irqstatus » a la valeur « 1 » pour son deuxième bit. Une fois

l'interruption reçue, il est important d'effacer l'évènement. Par la suite, il faut commencer à vider les données dans le tampon « fifo0data » jusqu'à ce qu'il n'est aucun. Lorsque le tampon est vidé et que le nombre d'échantillons reçu est inférieur à 1024, on quitte la fonction puisque la gestion de l'interruption est terminée. Pour savoir si le tampon est vide, il suffit de lire la valeur dans le registre « fifo0count » qui donne le nombre d'échantillons dans FIFO0. La collecte de données arrête lorsque le nombre d'échantillons reçu est de 1024. On change alors la valeur de plusieurs drapeaux. En effet, le drapeau « done » de la structure locale ADC est remis à « true » puisque l'ADC a terminé la collecte. Le drapeau *adc.donePtr est aussi mis à « true » ce qui a pour effet de modifier le drapeau du port en question de la structure PORT_DATA. Cela signifie que l'acquisition est terminée. Le module Task attend, en effet, ce changement de drapeau dans sa machine d'états de la tâche acquisition.

Il est aussi important de noter que la fréquence du grid est calculée en se basant sur le temps de départ et le temps de fin de l'acquisition de données. Ceci dit, lorsque le premier échantillon est lu du tampon, on prend ce temps comme le « start_time ». Lorsqu'un nombre suffisant d'échantillons ont été lus à partir de la FIFO0, il peut y avoir des échantillons supplémentaires dans la FIFO. Le nombre restant doit être pris en compte lors de la détermination de l'heure de fin du dernier échantillon lu. Ceci dit, on calcule le « end time » avec cette formule :

```
*adc.end_time = getCounter(adc.timerHandle);
*(adc.end_time) -= ((*(adc.end_time) - *(adc.start_time))
                    / ((adc.numSamples / fifoNumWords) + 1));
```

où « fifoNumWords » est le nombre d'échantillon restant dans le tampon après la fin de l'acquisition de données et « adc.numSamples » le nombre d'échantillons à collecter (1024).

3.9 Conception du module GPIO

Le module « General Purpose Input/Output » (GPIO) est une interface parallèle simple utilisée pour contrôler les relais du simulateur IDP et adresser l'ADC (« analog to digital converter ») sur les cartes relais pour l'échange de données des signaux de tension et de courant numérisés. En d'autres mots, le module fournit la fonctionnalité permettant de définir et de supprimer les bits qui contrôlent la tension qui apparait sur les broches GPIO correspondantes. La vue d'ensemble du matériel et la conception logicielle sera traitée.

3.9.1 Vue d'ensemble du matériel

Il sera d'abord question de traiter de la vue d'ensemble du matériel. La Figure 5 présente la configuration de la carte mère. Le microcontrôleur « TI Satara AM335XQuatres » qui est installé dans la carte BeagleBone contient quatre interfaces GPIO où chacun peut contrôler jusqu'à 32 broches.

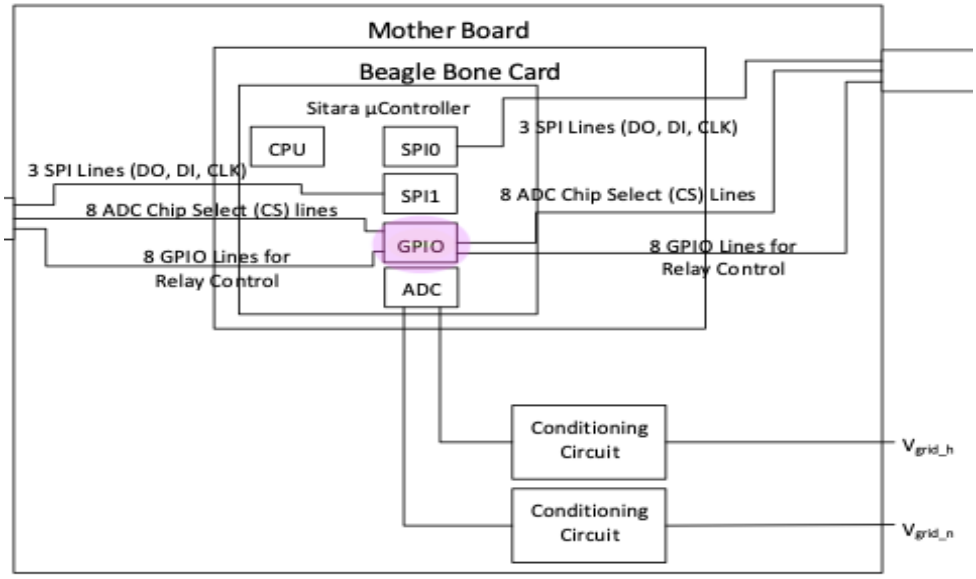

Figure 5 - Configuration de la carte mère où le GPIO est surligné

Dans le Tableau 1, on y retrouve l'adressage de base pour les différents registres du module GPIO. En effet, les registres du GPIO0 se trouvent entre l'adresse 44E0_7000 et 44E0_7FFF, les registres du GPIO1 se trouvent entre l'adresse 4804_C000 et 4804_CFFF, les registres du GPIO2 se trouvent entre l'adresse 481A_C000 et 481A_CFFF et les registres du

GPIO3 se trouvent entre l'adresse 481A_C000 et 481A_CFFF. Il est important de noter que seul les interfaces GPIO0, GPIO1 et GPIO2 sont utilisés dans le PDI.

Tableau 1: Adresse de base des interfaces GPIO

GPIO Identifiant	**Adresses**
GPIO0	44E0_7000 à 44E0_7FFF
GPIO1	4804_C000 à 4804_CFFF
GPIO2	481A_C000 à 481A_CFFF
GPIO3	481A_E000 et 481A_EFFF.

La Figure 6 contient une capture d'écran du manuel d'« AM335x and AMIC110 Sitara™ Processors Technical Reference » [2] qui présente le répertoire des registres mappés en mémoire pour l'interface GPIO. Parmi cette liste, quatre sont pertinents à la conception du module qui est « GPIO_OE », « GPIO_DATAOUT », « GPIO_CLEARDATAOUT » et « GPIO_SETDATAOUT ».

Offset	Acronym	Register Name	Section
0h	GPIO_REVISION		Section 25.4.1.1
10h	GPIO_SYSCONFIG		Section 25.4.1.2
20h	GPIO_EOI		Section 25.4.1.3
24h	GPIO_IRQSTATUS_RAW_0		Section 25.4.1.4
28h	GPIO_IRQSTATUS_RAW_1		Section 25.4.1.5
2Ch	GPIO_IRQSTATUS_0		Section 25.4.1.6
30h	GPIO_IRQSTATUS_1		Section 25.4.1.7
34h	GPIO_IRQSTATUS_SET_0		Section 25.4.1.8
38h	GPIO_IRQSTATUS_SET_1		Section 25.4.1.9
3Ch	GPIO_IRQSTATUS_CLR_0		Section 25.4.1.10
40h	GPIO_IRQSTATUS_CLR_1		Section 25.4.1.11
44h	GPIO_IRQWAKEN_0		Section 25.4.1.12
48h	GPIO_IRQWAKEN_1		Section 25.4.1.13
114h	GPIO_SYSSTATUS		Section 25.4.1.14
130h	GPIO_CTRL		Section 25.4.1.15
134h	GPIO_OE		Section 25.4.1.16
138h	GPIO_DATAIN		Section 25.4.1.17
13Ch	GPIO_DATAOUT		Section 25.4.1.18
140h	GPIO_LEVELDETECT0		Section 25.4.1.19
144h	GPIO_LEVELDETECT1		Section 25.4.1.20
148h	GPIO_RISINGDETECT		Section 25.4.1.21
14Ch	GPIO_FALLINGDETECT		Section 25.4.1.22
150h	GPIO_DEBOUNCENABLE		Section 25.4.1.23
154h	GPIO_DEBOUNCINGTIME		Section 25.4.1.24
190h	GPIO_CLEARDATAOUT		Section 25.4.1.25
194h	GPIO_SETDATAOUT		Section 25.4.1.26

Figure 6 – Registres du GPIO [2]

Le registre GPIO_OE est utilisé pour activer les capacités de sortie des broches. Chaque bit du registre correspond à une broche respective. Lors de la réinitialisation, toutes les broches liées à GPIO sont configurées avec les capacités d'entrée ce qui signifie que tous les bits ont la valeur de « 1 ». Pour activer une broche avec les capacités de sortie, il suffit de donner au bit correspondant la valeur de « 0 ». La seule fonction de ce registre est de porter la configuration des broches. Ceci dit, il est possible de voir dans la Figure 7 que les broches GPIO1_12, GPIO1_13, GPIO1_14 doivent être configurés avec les capacités de sortie dans le but d'ouvrir et de fermer les relais des trois ports du côté de la source. De plus, les broches GPIO0_8, GPIO0_9, GPIO0_10 doivent être configurés avec les capacités de sortie dans le but d'adresser les ADC des ports du côté de la source. Ceci signifie que le douzième, le treizième et le quatorzième bit du registre GPIO_OE de l'interface GPIO1 et que le huitième, le neuvième et le dixième bit du registre GPIO_OE de l'interface GPIO0 doivent être mis à « 0 ».

Ribbon Cable Connections: Source Side			
AM335x Connection	**BeagleBone Header Connection**	**Ribbon Cable Line**	**Description**
GPIO1:12, T12	P8-12	3	Relay P1
GPIO1:13, R12	P8-11	4	Relay P2
GPIO1:14, V13	P8-16	5	Relay P3
GPIO1:15, U13	P8-15	6	Relay P4
GPIO1:17, V14	P9-23	7	Relay P5
GPIO0:22, U10	P8-19	8	Relay P6
GPIO0:23, T10	P8-13	9	Relay P7
GPIO0:27, U12	P8-17	10	Relay P8
		11, 12	+5 V
		13, 19, 25, 16, 22, 28, 30	Digital Ground (DGND)
Decoded using 4 GPIO Lines: GPIO0:8, V2 GPIO0:9, V3 GPIO0:10, V4 SPI0_CS0, A16 GPIO0:8 is LSB	P8-35 P8-33 P8-31 P9-17	14	SPI 0 Chip Select (CS) 8
		15	SPI 0 Chip Select (CS) 7
		17	SPI 0 Chip Select (CS) 6
		18	SPI 0 Chip Select (CS) 5
		20	SPI 0 Chip Select (CS) 4
		21	SPI 0 Chip Select (CS) 3
		23	SPI 0 Chip Select (CS) 2
		24	SPI 0 Chip Select (CS) 1
UART2_TXD, B17	P9-21	26	SPI 0 Data 0 (D0) (Configure as Mode 0)
I2C1_SDA, B16	P9-18	27	SPI 0 Data 1 (D1) (Configure as Mode 0)
UART2_RXD, A17	P9-22	29	SPI 0 Clock (CLK) (Configure as Mode 0)

Figure 7- Connexion au câble « ribbon » du côté de la source

Il est possible de voir dans la Figure 8 que les broches GPIO2_1, GPIO0_26, GPIO1_16 doivent être configurés avec les capacités de sortie dans le but d'ouvrir et de fermer les relais des trois ports du côté de la charge. De plus, les broches GPIO2_23, GPIO2_24, GPIO2_25 doivent être configurés avec les capacités de sortie dans le but

d'adresser les ADC des ports du côté de la charge. Ceci signifie que le premier bit de GPIO2, le vingt-sixième bit de GPIO0 et le seizième bit de GPIO1 pour le registre GPIO_OE doivent être mis à « 0 ». Ceci signifie aussi que le vingt-troisième, le vingt-quatrième et le vingt-cinquième bit de GPIO2 du registre GPIO_OE doivent être mis à « 0 ».

Ribbon Cable Connections: Load Side			
AM335x Connection For IDP	**BeagleBone Header Connection**	**Ribbon Cable Connection**	**Description**
GPIO2:1, V12	P8-18	3	Relay P1
GPIO0:26, T11	P8-14	4	Relay P2
GPIO1:16, R13	P9-15	5	Relay P3
GPIO1:18, U14	P9-14	6	Relay P4
GPIO1:19, T14	P9-16	7	Relay P5
GPIO2:6, R1	P8-45	8	Relay P6
GPIO2:7, R2	P8-46	9	Relay P7
GPIO2:12, T3	P8-39	10	Relay P8
		11, 12	+5 V
		13, 19, 25, 16, 22, 28, 30	Digital Ground (DGND)
Decoded using 4 GPIO Lines: GPIO2:23, R5 GPIO2:24, V5 GPIO2:25, R6 SPI1_CS, C12 GPIO2:23 is LSB	P8-29 P8-28 P8-30 P9-28	14	SPI 1 Chip Select (CS) 8
		15	SPI 1 Chip Select (CS) 7
		17	SPI 1 Chip Select (CS) 6
		18	SPI 1 Chip Select (CS) 5
		20	SPI 1 Chip Select (CS) 4
		21	SPI 1 Chip Select (CS) 3
		23	SPI 1 Chip Select (CS) 2
		24	SPI 1 Chip Select (CS) 1
SPI1_D0, B13	P9-29	26	SPI 1 Data 0 (D0)
SPI1_D1, D12	P9-30	27	SPI 1 Data 1 (D1)
SPI1_CLK, A13	P9-31	29	SPI 1 Clock (CLK)

Figure 8 - Connexion au câble « ribbon » du côté de la charge

Le registre GPIO_DATAOUT est utilisé pour définir la valeur des broches de sortie GPIO. Le manuel d'« AM335x and AMIC110 Sitara™ Processors Technical Reference » *[2]* décrit comment modifier le contenu du registre. En effet, les données sont écrites dans le registre GPIO_DATAOUT de manière synchrone avec l'horloge de l'interface. Il est possible d'accéder à ce registre à l'aide d'opérations directes de lecture et d'écriture ou à l'aide des registres GPIO_DATAOUT et GPIO_CLEARDATAOUT. En effet, l'écriture d'un « 1 » sur un bit du registre GPIO_CLEARDATAOUT efface à « 0 » le bit correspondant dans le registre GPIO_DATAOUT et écrire un « 0 » n'a aucun effet. De plus, l'écriture d'un « 1 » dans un bit du registre GPIO_SETDATAOUT initialise à « 1 » le bit correspondant dans le registre GPIO_DATAOUT et écrire un « 0 » n'a aucun effet. Il est important de noter que la lecture du registre GPIO_SETDATAOUT ou du registre GPIO_CLEARDATAOUT renvoie la valeur du registre de sortie de données (GPIO_DATAOUT).

3.9.2 Vue d'ensemble du logiciel

Le module GPIO se trouve dans la couche matérielle. La couche matérielle fournit, en fait comme son nom l'indique, les fonctionnalités pour manipuler le matériel. Dans la Figure 9, il est possible de voir que seul le module « Relay » utilise directement les fonctions de GPIO. Ce dernier se trouve dans la couche « Fonctionnalité » qui fournit, comme son nom l'indique, les fonctions de traitement du système. Il est aussi possible de voir, dans la Figure 9, que la tâche « Relay Mnmt » qui se trouve dans la couche tâche utilise directement les fonctions du module Relay.

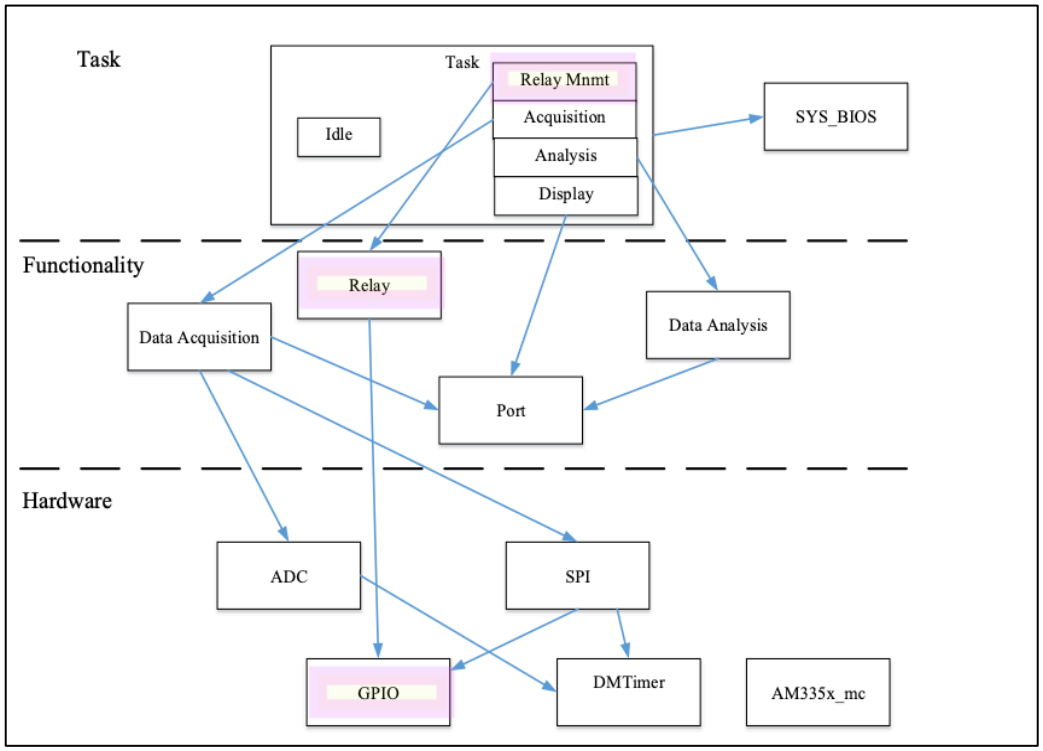

Figure 9 - Relation entre les modules nécessitant l'implémentation du module GPIO

La conception du module GPIO repose sur la structure de données « GPIO_REGS » est présenté dans l'Annexe 4. Cette structure contient l'ensemble des registres des interfaces GPIO. Parmi cette liste, seuls les registres « dataout », « cleardataout », « setdataout » et « oe » sont pertinents à la conception du module. Comme il y a seulement trois des quatre interfaces GPIO qui sont utilisées, le module GPIO utilise un tableau de taille 3 nommé « gpio » contenant des pointeurs vers les structures *GPIO_REGS.*

```
GPIO_REGS *gpio[3];
        gpio[0] =(GPIO_REGS *) GPIO0_REG;
        gpio[1] = (GPIO_REGS *) GPIO1_REG;
        gpio[2] = (GPIO_REGS *) GPIO2_REG;
```

Par la suite, il est primordial de configurer les broches des registres GPIO_OE pour activer les capacités de sortie et adresser l'ADC sur les ports. Les broches GPIO1_12, GPIO1_13, GPIO1_14 doivent être configurés avec les capacités de sortie dans le but d'ouvrir et de fermer les relais des trois ports du côté de la source. De plus, les broches GPIO0_8, GPIO0_9, GPIO0_10 doivent être configurés avec les capacités de sortie dans le but d'adresser les ADC des ports du côté de la source. Ceci signifie que le douzième, le treizième et le quatorzième bit du registre GPIO_OE de l'interface GPIO1 et que le huitième, le neuvième et le dixième bit du registre GPIO_OE de l'interface GPIO0 doivent être mis à « 0 ». Les broches GPIO2_1, GPIO0_26, GPIO1_16 doivent être configurés avec les capacités de sortie dans le but d'ouvrir et de fermer les relais des trois ports du côté de la charge. De plus, les broches GPIO2_23, GPIO2_24, GPIO2_25 doivent être configurés avec les capacités de sortie dans le but d'adresser les ADC des ports du côté de la charge. Ceci signifie que le premier bit de GPIO2, le vingt-sixième bit de GPIO0 et le seizième bit de GPIO1 pour le registre GPIO_OE doivent être mis à « 0 ». Ceci signifie aussi que le vingt-troisième, le vingt-quatrième et le vingt-cinquième bit de GPIO2 du registre GPIO_OE doivent être mis à « 0 ». On peut voir ci-dessous, un aperçu du code activant les capacités des broches désirées.

```
#define GPIO0_OUT_PINS (BIT8 | BIT9 | BIT10 | BIT26)
#define GPIO1_OUT_PINS (BIT12 | BIT13 | BIT14 | BIT16)
#define GPIO2_OUT_PINS (BIT1 | BIT23 | BIT24 | BIT25)

    gpio[0]->oe &=  ~GPIO0_OUT_PINS;
    gpio[1]->oe &=  ~GPIO1_OUT_PINS;
    gpio[2]->oe &=  ~GPIO2_OUT_PINS;
```

RÉFÉRENCES

[1] ARBEZ, Gilbert. Simulateur PDI: Collecte de données Version 1.0.0; Document de déploiement. Ottawa: Université d'Ottawa, 2018.

[2] TEXAS INSTRUMENTS. AM335x and AMIC110 Sitara Processors: Technical Reference Manual. Ottawa: 2011.

ANNEXE 1

Implémentation machine d'états en C:

Handler transitions [RELAY_STATE_COUNT][LAST_CLOSED_COUNT][RELAY_STATE_COUNT][SOURCE_STATE_COUNT] =

```
{ { // Source off
{ //Last closed source 1
{ badState, closeSource2, badState },  //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
},
{ //Last closed source 2
{ badState, closeSource3, badState }, //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
},
{ //Last closed source 3
{ badState, closeSource1, badState }, //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
} },
 { // Source 1 closed
 { //Last closed source 1
 { closeCharge1, badState, openSource1 }, //Charge off
 { closeCharge2, badState, badState },   //Charge 1
 { closeCharge3, badState, badState },   //Charge 2
 { openAllCharges, badState, badState }  //Charge 3
 },
 { //Last closed source 2
 { badState, badState, badState },    //Charge off
 { badState, badState, badState },    //Charge 1
 { badState, badState, badState },    //Charge 2
 { badState, badState, badState }     //Charge 3
 },
 { //Last closed source 3
 { badState, badState, badState },    //Charge off
 { badState, badState, badState },    //Charge 1
 { badState, badState, badState },    //Charge 2
```

```
{ badState, badState, badState }     //Charge 3
} },
{ // Source 2 closed
{ //Last closed source 1
{ badState, badState, badState },    //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
},
{ //Last closed source 2
{ closeCharge1, badState, openSource2 }, //Charge off
{ closeCharge2, badState, badState },   //Charge 1
{ closeCharge3, badState, badState },   //Charge 2
{ openAllCharges, badState, badState }  //Charge 3
},
{ //Last closed source 3
{ badState, badState, badState },    //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
} },
{ // Source 3 closed
{ //Last closed source 1
{ badState, badState, badState },    //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
},
{ //Last closed source 3
{ badState, badState, badState },    //Charge off
{ badState, badState, badState },    //Charge 1
{ badState, badState, badState },    //Charge 2
{ badState, badState, badState }     //Charge 3
},
{ //Last closed source 2
{ closeCharge1, badState, openSource3 }, //Charge off
{ closeCharge2, badState, badState },   //Charge 1
{ closeCharge3, badState, badState },   //Charge 2
{ openAllCharges, badState, badState }  //Charge 3
} }, };
```

ANNEXE 2

Structure des ports :

```
typedef struct
{
   // Associate buffer to a SRC
   PORT_GROUP group;  /*!< SOURCE, LOAD or GRID */
   PORT_ID id; /*! Port identifier (number); */
   // Status information
   RELAY_STATUS relayStatus;  /*!< OPEN - relay closed, CLOSE - relay open */
   bool acquireDone; /*!< set to true when data acquisition is complete */
   bool startAnalysis; /*!< set to true indicates analysis can start */
   bool analysisDone;     /*!< set to true when data analysis is complete */
   // Use to time stamp data collection
   uint32_t start_time;   /*!< start time of the data collection */
   uint32_t end_time;     /*!< end time of the data collection */
   // For filling raw data values
   FILL_TYPE transferType;  /*!< set to VOLTAGE, CURRENT, or BOTH to indicate what data is being
collected. */
   uint16_t databufs[DATA_SIZE]; /*!< for Vh, Vn, Ih, In raw ADC data */
   // For data analysis
   double voltageBuf[DATA_SIZE_SIGNAL2]; /*!< For V (volts) */
   double currentBuf[DATA_SIZE_SIGNAL2]; /*!< For I (amps) */
   double vrms;  /*!< rms voltage (volts) */
   double irms;  /*!< rms current (amps) */
   double realPower; /*!< Watts */
   double reactivePower; /*!< Volt-amps */
   double frequency;  /*!< Frequency in Hertz */
   double powerFactor; /*!< power factor */
   double tzv1, tzv2, tzi; /*!< for debugging */
} PORT_DATA;  /*!< Port Data structure */
```

ANNEXE 3

Capture d'écran de la structure « ADC_REGS *» du module « adcDefinition.h »*

```
typedef volatile struct
{
    volatile uint32_t revision;       // Offset:0x0    ADC_REVISION - system configuration
    volatile uint32_t rsvd1[3];       // Offset 0x4 to 0xFF  - reserved bytes
    volatile uint32_t sysconfig;      // Offset 0x10  ADC_SYSCONFIG - system configuration
    volatile uint32_t rsvd2[4];       // Offset 0x14 to 0x23  - reserved bytes
    volatile uint32_t irqstatus_raw;  // Offset 0x24  ADC_IRQSTATUS_RAW - IRQ status unmasked
    volatile uint32_t irqstatus;      // Offset 0x28  ADC_IRQSTATUS - IRQ status register
    volatile uint32_t irqenable_set;  // Offset 0x2C  ADC_IRQENABLE_SET -  IRQ enable set register
    volatile uint32_t irqenable_clr;  // Offset 0x30  ADC_IRQENABLE_SET -  IRQ enable clear register
    volatile uint32_t irqwakeup;      // Offset 0x34  ADC_IRQWAKEUP - IRQ wakeup register
    volatile uint32_t dmaenable_set;  // Offset 0x38  ADC_DMAENABLE_SET -DMA enable set register
    volatile uint32_t dmaenable_clr;  // Offset 0x3C  ADC_DMAENABLE_CLR -DMA enable clear register
    volatile uint32_t cntrl;          // Offset 0x40  ADC_CTRL -  ADC control register
    volatile uint32_t stat;           // Offset 0x44  ADC_STAT -  ADC status register
    volatile uint32_t range;          // Offset 0x48  ADC_RANGE -  ADC range register
    volatile uint32_t clkdiv;         // Offset 0x4C  ADC_CLKDIV -  ADC clock divider register
    volatile uint32_t misc;           // Offset 0x50  ADC_MISC -  ADC miscellaneous register
    volatile uint32_t stepenable;     // Offset 0x54  ADC_STEPENABLE -  ADC step enable register
    volatile uint32_t idleconfig;     // Offset 0x58  ADC_IDLECONFIG -  ADC idle config register
    volatile uint32_t ts_charge_stepconfig; // Offset 0x5C  ADC_TS_CHARGE_STEPCONFIG -  ADC TS charge
step config register
    volatile uint32_t ts_charge_delay; // Offset 0x60  ADC_TS_CHARGE_DELAY -  ADC charge delay register
    volatile uint32_t step[NUM_STEPS][2]; // Offset 0x64 to 0xE3 ADC_CONFIG and ADC_STEP,
configuration and step registers 1 to 16 (indexed 0 to 16)
    volatile uint32_t fifo0count;     // Offset 0xE4  ADC_FIFO0OUNT -  ADC FIFO 0 count register
    volatile uint32_t fifo0threshold; // Offset 0xE8  ADC_FIFO0THRESHOLD -  ADC FIFO 0 Threshold
register
    volatile uint32_t dma0req;        // Offset 0xEC  ADC_DMA0REQ -  ADC DMA 0 request register
    volatile uint32_t fifo1count;     // Offset 0xF0  ADC_FIFO1OUNT -  ADC FIFO 1 count register
    volatile uint32_t fifo1threshold; // Offset 0xF4  ADC_FIFO1THRESHOLD -  ADC FIFO 1 Threshold
register
    volatile uint32_t dma1req;        // Offset 0xF8  ADC_DMA1REQ -  ADC DMA 1 request register
    volatile uint32_t rsvd3;          // Offset 0xFC - reserved bytes
    volatile uint32_t fifo0data;      // Offset 0x100  ADC_FIFO0DATA -  ADC fif0 data register
    volatile uint32_t rsvd4[63];      // Offset 0x104 to 1FF - reserved bytes
    volatile uint32_t fifo1data;      // Offset 0x200  ADC_FIFO0DATA -  ADC fif1 data register

} ADC_REGS;
```

ANNEXE 4

Capture d'écran de la structure « GPIO_REGS » du module « gpioDefinition.h »

```
// Register Structure
typedef struct
{
    volatile uint32_t revision;        // Offset 0x00 - Read only register giving revision number
    volatile uint32_t rsvd1[3];        // Not used
    volatile uint32_t sysconfig;       // Offset 0x10 - Controls various parameters of L4 interconnect.
    volatile uint32_t rsvd2[3];        // Not used
    volatile uint32_t eoi;             // Offset 0x20 - Supports DMA events.
    volatile uint32_t irqstatus_raw_0; // Offset 0x24 - Provides core status information for interrupts.
    volatile uint32_t irqstatus_raw_1; // Offset 0x28 - Provides core status information for interrupts.
    volatile uint32_t irqstatus_0;     // Offset 0x2c - Provides core status information for enabled interrupts.
    volatile uint32_t irqstatus_1;     // Offset 0x30 - Provides core status information for enabled interrupts.
    volatile uint32_t irqstatus_set_0; // Offset 0x34 - Enables specific interrupt event to trigger.
    volatile uint32_t irqstatus_set_1; // Offset 0x38 - Enables specific interrupt event to trigger.
    volatile uint32_t irqstatus_clr_0; // Offset 0x3c - Clears specific interrupt event to trigger.
    volatile uint32_t irqstatus_clr_1; // Offset 0x40 - Clears specific interrupt event to trigger.
    volatile uint32_t irqwaken_0;      // Offset 0x44 - Enables wakeup events on an interrupt.
    volatile uint32_t irqwaken_1;      // Offset 0x48 - Enables wakeup events on an interrupt.
    volatile uint32_t rsvd3[50];       // Not used
    volatile uint32_t sysstatus;       // Offset 0x114 - Provides reset status.
    volatile uint32_t rsvd4[6];        // Not used
    volatile uint32_t ctrl;  // Offset 0x130 - Controls clock gating functionality, i.e. enables module.
    volatile uint32_t oe; // Offset 0x134 □ Output Enable pin (clear bit to 0) output capability.
    volatile uint32_t datain;          // Offset 0x138 - Registers data read from the GPIO pins.
    volatile uint32_t dataout;         // Offset 0x13c - Sets value of GPIO output pins.
    volatile uint32_t leveldetect0;    // Offset 0x140 - Enable/disable line low-level (0) interrupt.
    volatile uint32_t leveldetect1;    // Offset 0x144 - Enable/disable line high-level (0) interrupt.
    volatile uint32_t risingdetect;    // Offset 0x148 - Enable/disable line rising-edge interrupt.
    volatile uint32_t fallingdetect;   // Offset 0x14c - Enable/disable line falling-edge interrupt.
    volatile uint32_t debouncingenable;  // Offset 0x150 - Enable/disable debouncing.
    volatile uint32_t debouncingtime;    // Offset 0x154 - Controls debouncing time (for all ports).
    volatile uint32_t rsvd5[14];       // Not used
    volatile uint32_t cleardataout;    // Offset 0x190 - Clears to 0 bits in dataout
    volatile uint32_t setdataout;      // Offset 0x194 - Sets to 1 bits in dataout
} GPIO_REGS;
```