

Sprint 1 - Livrable #2

SEG4545/CEG4566/CSI4141 - Conception de système en temps réel



uOttawa

Hiver 2023

School of Engineering Design and Teaching innovation

University of Ottawa

Professor: Gilbert Joseph Paul Arbez

Groupe: 5

Étudiant 1: **Berté, Tata, 300100935**

Étudiant 2: **El-Saleh, Hady, 300028268**

Étudiant 3: **Cisse, Hamidou, 300088010**

Étudiant 4: **Flynn, Benjamin, 300106230**

Étudiant 5 : **Younes, Anys, 300145843**

Étudiant 6 : **Nabil wahbi, 300144558**

Date de soumission: **Samedi 04 Février, 2023**

Introduction


L'objectif de ce deuxième livrable est de présenter la conception révisée de plusieurs modules principaux pour notre système. Autrement dit, nous nous concentrons sur la conception du module de tâches, du module ADC et du module GPIO. La section 3.2 détaille la conception du module de tâches, y compris ses fonctions et son implémentation. La section 3.7 présente la conception du module ADC, y compris son schéma de circuit et ses mesures de performance. Enfin, la section 3.9 décrit la conception du module GPIO, y compris son interface et son utilisation dans le système global. Ensemble, ces modules constituent une partie cruciale de la conception globale du système et contribuent à ses performances et à sa fonctionnalité.

L'ébauche de Conception

Cette section vise à présenter les ébauches de base pour les modules de tâche, ADC et GPIO. Pour répondre aux exigences et aux spécifications, une ébauche de conception détaillée est fournie sur la conception et leur mise en œuvre.

3.2 Task Module Design : Ébauche de conception

Le module Task offre des tâches ou fonctions utilisées par le système pour effectuer différentes actions dont le déroulement est indépendant d'une interruption générée par le matériel ou entrée extérieur au module. Ces différentes tâches sont exécutées l'une après l'autre par un pseudo noyau de code cyclique. Le pseudo noyau est une fonction utilisateur écrite en C et rajoutée au noyau SYMBIOS. Cela permet alors au RTOS (Real Time Operating System) de se charger de l'ordonnancement des tâches et des appels aux différentes fonctions présentes dans son noyau. Le pseudo noyau est formé de 5 fonctions qui constituent nos différentes tâches:

- **manageRelays_task():** Cette fonction suit une Machine d'état prédéfini pour ouvrir et fermer les relays. 
- **acquisition_task():** Cette fonction se charge de remplir les valeurs scalaires des structures PORT_DATA.
- **analysis_task():** Cette fonction s'occupe de l'analyse des données reçues par acquisition à l'aide du module Analysis.
- **display_task():** Cette fonction se charge d'afficher sur le terminal une vue des données de voltages et courants des différents ports et du grid.
- **freeze_task():** : Cette fonction arrête le processus d'acquisition des données et fige le terminal.

3.2.1 manageRelay_task() function:

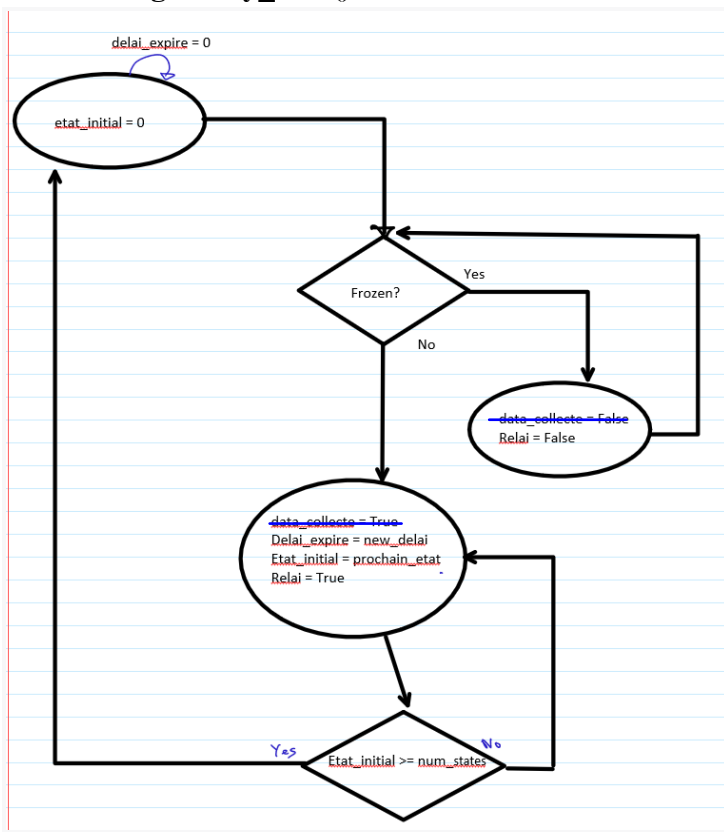


Figure 1: Cet figure montre une Machine d'état pour le module *relay*

Le FSM ci-dessus décrit la gestion de relais d'où d'abord, l'état initial est set a 0 avec un variable `delai_expire` pour indiquer l'expiration du temps de 5 secondes qui est set aussi a 0. Ensuite, ça passe à une condition d'où si l'état est figé, la collection de données est arrêtée ainsi que les relais sont suspendus. Sinon, la collecte des données est fait, le `delai_expire` est set au nouveau délai, l'état initial passe au prochain état et le fermeture/ouverture des relais est active. Enfin, ça passe au dernier condition ou si le temps de l'état initial est supérieur ou égal à celui de la minuterie, ça se passe au premier état et ça reset. Sinon, ça revient à l'état précédent et passe au prochain état dès que la condition est atteinte.

Initial State	Next State	Relais S1	Relais S2	Relais S3	Relais L1	Relais L2	Relais L3	Delay
Tout Relais sont ouvert	N/A	Open	Open	Open	Open	Open	Open	0 sec
Tout Relais sont ouvert	S1	Open	Closed	Closed	Closed	Closed	Closed	5 sec
S1	S2	Closed	Open	Closed	Closed	Closed	Closed	5 sec
S2	S3	Closed	Closed	Open	Closed	Closed	Closed	5 sec

Tableau 1 : Ce tableau montre le fonctionnement du FSM

Dans le tableau ci-dessus, nous avons l'état initial où tous les relais sont ouverts et un délai de 0 secondes. Si aucun caractère est appuyé, l'état reste le même. Cependant, si le relais est fermé, les relais S2 et S3 seront fermés ainsi que les relais L1, L2 et L3 du S1. Aussi, un délai de 5 secondes est ajouté entre l'ouverture et la fermeture de relais(Sx) et non celles des Lx. Pour l'état prochain, on pourra passer à S1 ou S2 et le même principe sera appliqué.

Nous implémentons le délai de 5 secondes en utilisant une variable `timer_track`. La fonction `getCounter()` nous renvoie la valeur actuelle du compteur général, nous sauvegardons cette valeur dans `timer_track` initialement puis à chaque cycle du noyau cyclique nous vérifions si la différence entre `getCounter()` et `timer_track` est supérieure à 7000 qui correspond au nombre de cycles nécessaire pour un délai de 5 secondes. Si la condition est vraie, nous passons à l'état suivant et changeons la valeur de `timer_track` pour la valeur actuelle de `getCounter()` et recommençons l'opération.

3.2.2 Data Acquisition Task Function

La fonction *Acquisition* du module *task* s'occupe dans le module tâche de la synchronisation, la collecte des données. La réalisation synchronisation se fait avec des variables booléennes dans les structures **PORT DATA** au niveau des tâches d'analyse et d'affichage. La collecte des données s'effectue par le matériel (hardware) de deux manières différentes soit par l'intermédiaire du module SPI/ADC ou par les ports sources, charges et grid.

Elle est effectuée par le module d'acquisition qui fournit une fonction pour commencer la collecte sur les ports source, charge et grid.

Le module initie la collecte de données pour le port souhaité du côté de la charge, de la source ou du grid.

Une structure **PORT DATA** est obtenue du module Port Data qui sera comblée par le module ADC/SPI. La collecte des données est dirigée par des interruptions et se termine lorsque toutes les données ont été collectées. Un indicateur *acquireDone* est mis à jour dans la structure **PORT DATA** lorsque la collecte est terminée.

Le FSM de ce module est décrit dans la figure 2 ci-dessus:

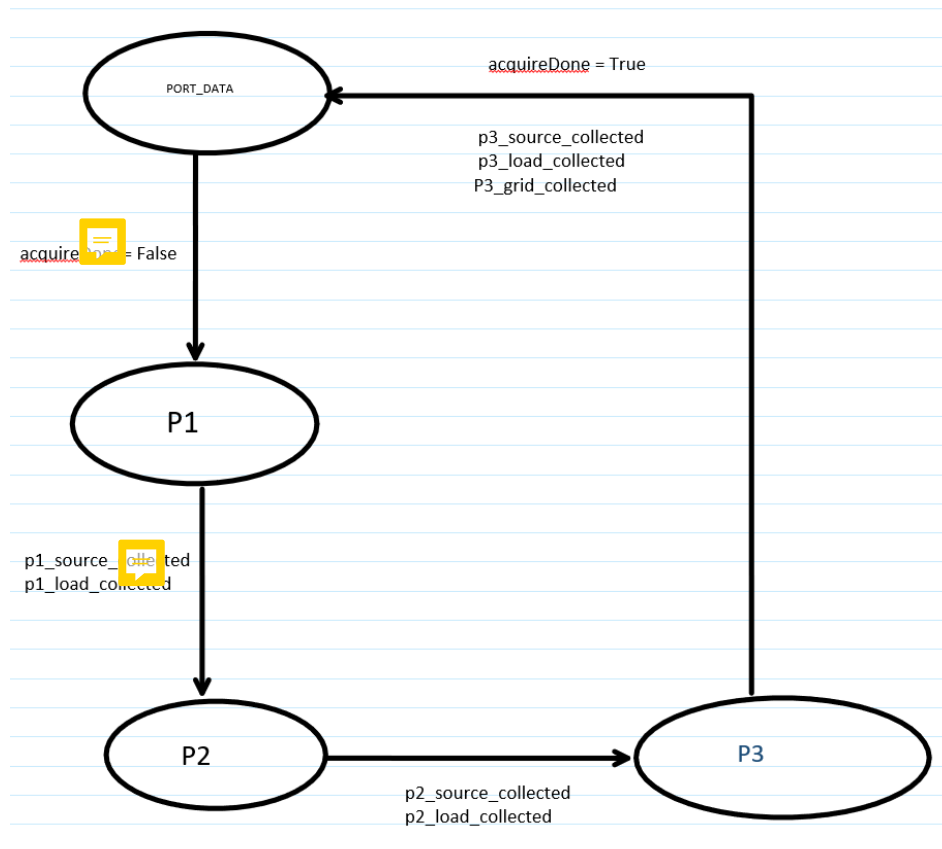


Figure 2: Cet figure montre une Machine d'état pour le module *data acquisition*

Le FSM du module data acquisition est décrit selon le suivant. D'abord, l'état initial (**PORT_DATA**) est en attente du début de la collecte où le processus de collecte des données n'a pas encore commencé. Lorsque la condition pour passer à l'état suivant (P1) est atteinte qui est définie par une valeur booléenne qui indique le début du processus de collecte, la collection du P1 débute. Dans cet état, les tâches de collecte de données pour les ports P1 de la source et de la charge seront effectuées. La condition pour passer à l'état suivant sera la valeur booléenne de source/load qui indique la fin de la collecte des données des ports. Ensuite, on passe au prochain état P2 et le même principe est fait jusqu'au état P3 d'où les tâches de collecte de données pour les ports P3 de la source, de la charge et du grid seront effectuées. Enfin, un indicateur **acquireDone** est mis à jour dans la structure **PORT DATA** lorsque la collecte est terminée. Ensuite, le prochain état sera l'état initial.

3.2.3 Data Analysis Task Function

Cette tâche s'exécute lorsqu'un port est prêt à être analysé. Chaque appel à la fonction *analysis_task()* analyse et modifie les données spécifiques à un port, nous utilisons alors une variable *port_nextAnalysis* ainsi qu'un tableau *analysisPort[]* pour analyser successivement tous les ports à chaque appel de la fonction *analysis_task()*. Le module *analysis* possède une fonction *analysisAll()* qui nous permettra de convertir des données brutes reçues à partir des ADC en valeur scalaire ou voltages et courants.


~~Ces valeurs scalaires devraient être accessibles par divers modules externes tel que *display*.~~



Lorsqu'un drapeau *startAnalysis* est mis à 1 nous vérifions que le drapeau *analysisDone* est à 0, nous analysons alors les données présentes avec dans le port et enregistrons les résultats dans la structure *PORT_DATA*.



3.2.4 Display Task Function

Le module Display Task surveille les structures *PORT_DATA* afin d'afficher les données des ports. Elle affiche les valeurs scalaires calculées dans la tâche d'analyse pour chacun des ports dans un terminal ASCII. Les données seront affichées dans un terminal en utilisant les fonctions de l'API qui dépendent des interruptions. Elles se chargeront de copier les données qui seront transmises à l'aide d'un tampon d'envoi (ring buffer) puis les données seront copiées dans des tampons de réception (ring-buffer). L'ISR (Interrupt Service Routine) s'occupera de remplir ou de vider les tampons de transmission et de réception au fur et à mesure que des données seront disponibles ou requises. Les données seront finalement lues à partir des tampons de réception et affichées dans le terminal.

La condition de l'affichage est la suivante : les booléens "~~*acquireDone*~~" et "*analysisDone*" doivent ~~tous les deux~~ être définis à "vrai"  ce qui veut dire que les données ont été ~~acquises et~~ analysées, ce qui permettra donc l'affichage des données.

Pour signaler aux autres tâches de faire une autre collection/analyse des données, on ~~peut utiliser~~ les variables booléennes "*acquireDone*", "*startAnalysis*" et "*analysisDone*" des structures *PORT_DATA*. On change l'état de ces variable  pour signaler aux autres tâches que la collection/analyse actuelle est terminée et qu'une nouvelle peut commencer. Par exemple, lorsque l'acquisition de données est terminée, on peut définir "*acquireDone*" à vrai,  ce qui indique aux autres tâches que l'analyse peut commencer. Une fois que l'analyse est terminée, on peut définir "*analysisDone*" à vrai, ce qui indique aux autres tâches que la nouvelle collection de données peut commencer.

Le terminal VT100 est un terminal d'ordinateur couramment utilisé pour se connecter à des systèmes de mainframe ou de serveur. Il fournit une interface utilisateur graphique avec un

certain nombre de fonctionnalités pour permettre l'affichage et la saisie de données. Il est utilisé dans le code d'exemple de la fonction display task .

3.2.5 Freeze Task Function

La fonction **Freeze** du module **task** est un sous-module qui nous permet de changer l'état du système. La fonction nous permet de passer d'un état figé à un état actif. La fonction nous permet d'activer les sous modules de gestion des relais et d'acquisition des données. La machine d'état suivante nous permet de modéliser le comportement du système avec les spécifications suivantes: modifier l'état de sous-modules externes et prendre des entrées du terminal à partir du module contrôlant les entrées UART.

Nous implémentons ce sous-module en utilisant l'une des fonctions fournies par le module **UART**.

uart_getchar () nous retourne le caractère présent dans le tampon ou **EOF (end of file)**. Une structure **switch case** nous permet de changer l'état du système dépendamment de la valeur retourner par **uart_getchar()** avec '**f**' pour figer le système et '**c**' pour continuer. Toutes autres valeurs y compris **EOF** exécute le code **default** qui sera laissé vide.

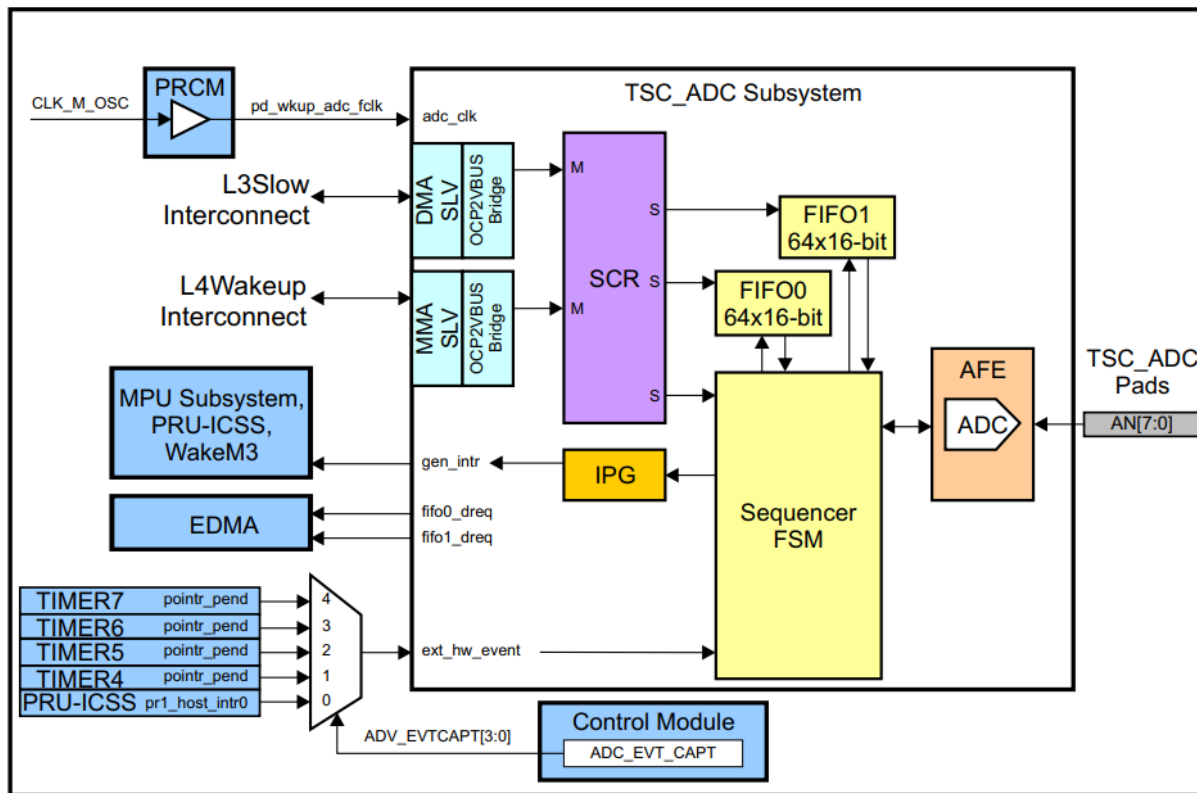
Une variable globale, **frozen_flag**, est utilisée pour la synchronisation ou l'activation des modules d'acquisition de données, d'affichage et de contrôle des relais.

Section 3.7 ADC Module Design

Section 3.7.1 ~~IDP ADC and AM355x~~ ADC Hardware Module Overview

Notre microcontrôleur AM355x nous permet d'utiliser l'ADC avec des interruptions logicielles et matérielles en utilisant le statut d'un tampon **FIFO** ou un **drapeau** (bit) pour le statut de conversion de l'ADC. Nous avons alors un module présent dans le micro contrôleur pour intégrer l'ADC avec des composants externes. La figure 12. 1 nous présente une vue de cette intégration.

Figure 12-1. TSC_ADC Integration



pr1_host_intr[0:7] corresponds to Host-2 to Host-9 of the PRU-ICSS interrupt controller.



Nous utilisons des registres supplémentaires pour configurer et interagir avec l'ADC, plus d'informations sur des registres spécifiques dans le manuel d'utilisateur du microcontrôleur, *chapitre 12.5.1*.

Les registres font 32-bit et nous permettent de configurer des interruptions, le facteur de division, le délai d'échantillonnage ainsi que la résolution.

12.5.1 TSC_ADC_SS Registers

Table 12-4 lists the memory-mapped registers for the TSC_ADC_SS. All register offset addresses not listed in Table 12-4 should be considered as reserved locations and the register contents should not be modified.

Table 12-4. TSC_ADC_SS Registers

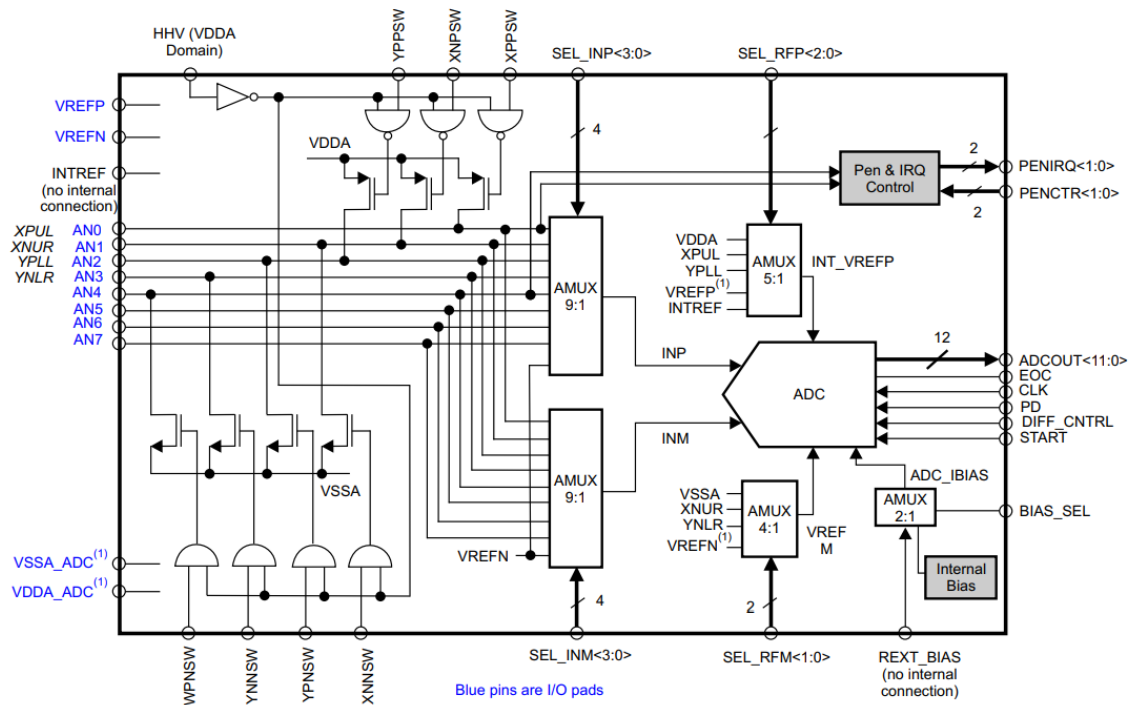
Offset	Acronym	Register Name	Section
0h	REVISION		Section 12.5.1.1
10h	SYSCONFIG		Section 12.5.1.2
24h	IRQSTATUS_RAW		Section 12.5.1.3
28h	IRQSTATUS		Section 12.5.1.4
2Ch	IRQENABLE_SET		Section 12.5.1.5
30h	IRQENABLE_CLR		Section 12.5.1.6

Pour notre conception nous utiliserons les registres suivant pour la configuration de l'ADC: *IRQSTATUS*, *IRQENABLE_SET*, *CTRL*, *STEPCONFIG1*, *STEPCONFIG2*, *STEPDELAY1* et *STEPDELAY2*, *FIFO0COUNT*, *FIFO0THRESHOLD*.



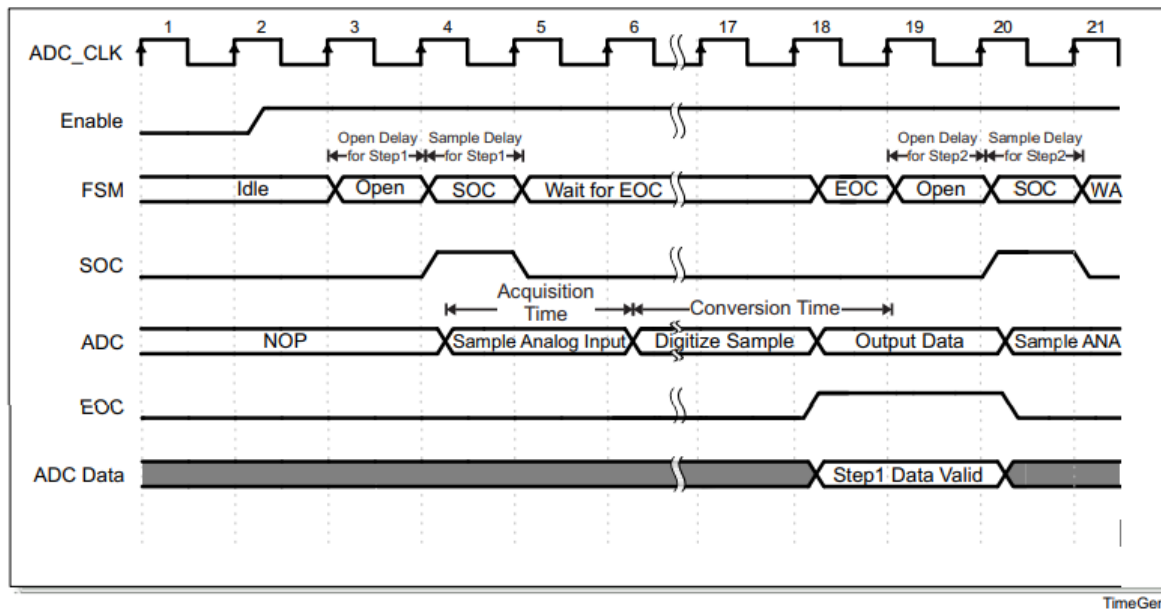
La figure 12-2 est une représentation schématique du bloc fonctionnel, le bloc **AFE** de la figure 12-1. Cette figure nous permet d'avoir une meilleure compréhension de la disposition physique des composants et leur interactions avec des signaux externes. Nous remarquons la présence des différents canaux de lectures analogiques sur la gauche, des multiplexeurs permettant leur sélection ainsi que 2 autres multiplexeurs pour la sélection des voltages de références. Des bits de contrôles ainsi que le registre de sortie sur la droite. ~~Le module **Pen&IRQ** est utilisé pour des évènements Touch-Screen.~~

Figure 12-2. Functional Block Diagram



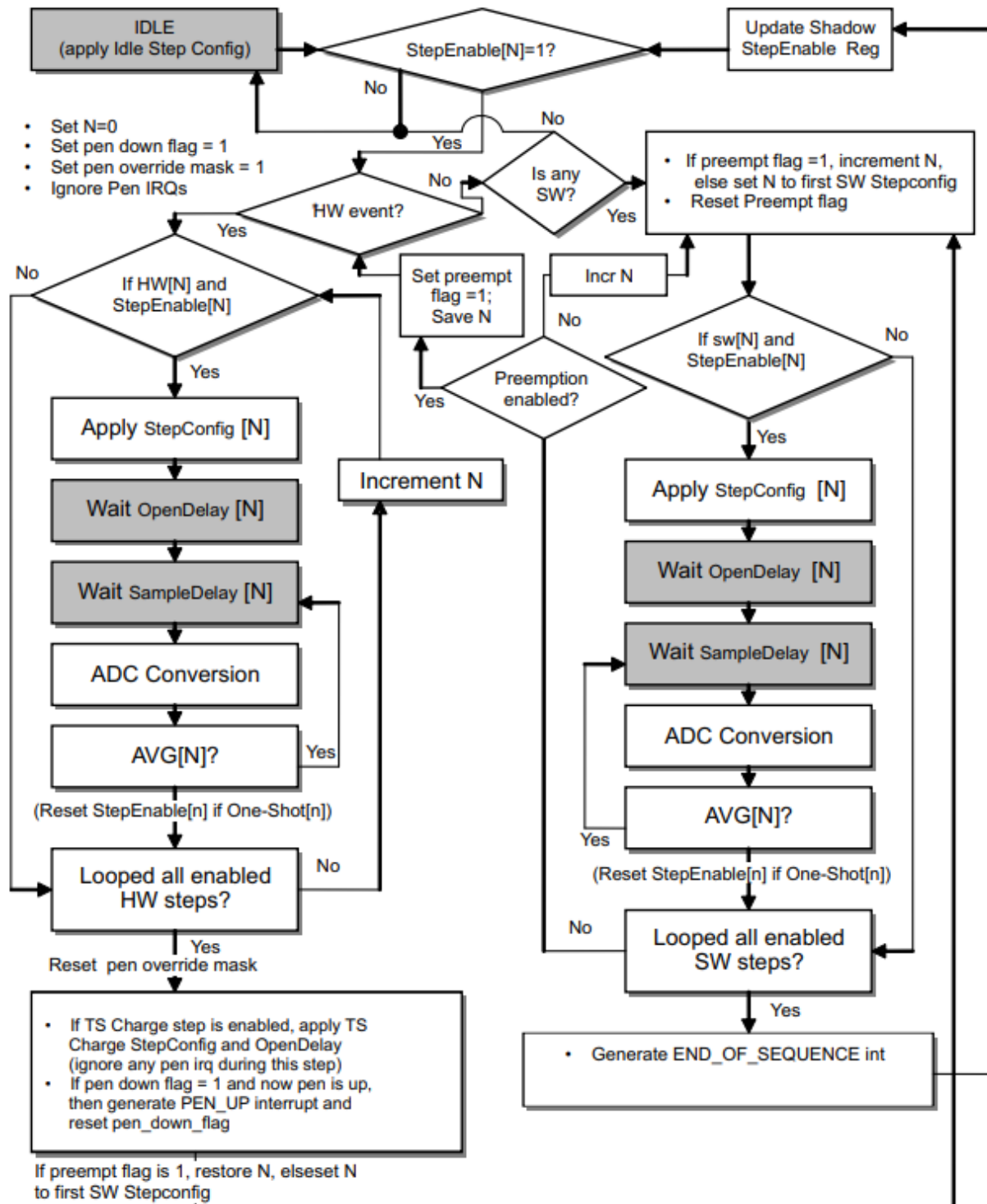
En configurant les registres spécifiés plus haut, le diagramme de temps dans la figure 12-4 nous permet d'avoir une idée des étapes pour la conversion d'une entrée analogique pour notre système PDI ainsi que les délais nécessaires pour la configuration de notre système, nous avons besoin de convertir deux valeurs de voltages (*Hot and neutral voltages*)

Figure 12-4. Example Timing Diagram for Sequencer



Nous pouvons alors diviser la procédure de conversion en deux étapes, *step1* et *step2*, présentent avant et après l'événement **EOC (end of conversion)**. Le mécanisme pour les deux étapes est identique et nous nous référons à la machine d'état présente dans la figure 12-3 pour la définir.

Figure 12-3. Sequencer FSM



Section 3.7.2 Design

Pour l'implémentation du module design, nous aurons besoin de compléter une fonction qui enregistre les données brutes lus par l'ADC interne de la carte dans un tampon. ~~starAdeAcq (~~), modifier l'initialisation de l'ADC ainsi que implémenter l'ISR (interrupt service routine) utilisé pour remplir des tampons.

Pour l'initialisation nous aurons besoin de déterminer des délais précis pour le fonctionnement du systèmes avec les paramètres suivants:

- 512 échantillons pour le voltage *hot* et *neutral*
- Le signal a une fréquence de 60Hz donc une période de 16.7ms
- Nous échantillons alors pour une durée de 16.7ms/512 pour chaque échantillons, **32.6us**

En faisant référence à la figure 12-4 notre temps total d'échantillonnage de 32.6us correspond au temps écoulé entre le premier délai *open* pour *step1* et la fin de *step2*. En assumant une période de conversion de 13 cycle d'horloge pour 16 séries de lectures par échantillon et en configurant un délai d'échantillonnage d'un cycle d'horloge nous obtenons le délais suivant pour arriver à *EOC* : $14c * 16 = 224$ cycles avec une horloge de 24 MHz nous avons une durée de 9.333 us pour *step1* et *step2* (**18.66 us total**). Le "*open delay*" pour *step2* est de 0 car nous désirons que la conversions juste après *step1*, alors le *open delay* (délai calculé pour obtenir le timing désiré) pour *step1* est de 13.94 us ou **335** cycles d'horloges

Nous utilisons *FIFO0* en tant que registre de réception, nous levons une interruptions lorsque le tampon *FIFO0* reçoit 32 échantillons avec 16 bits par échantillons.

Nous avons alors les configuration suivantes pour les registres de notre module ADC:

- **IRQSTATUS** : Les différents champs de ce registre nous permettent de déterminer et modifier le statut des différentes interruptions
- **IRQENABLE_SET** : Nous permet d'activer les interruptions et de définir le seuil auquel les interruptions sont générées avec le champ *FIFO0_threshold*. Pour notre conception nous utilisons un seuil de 32 échantillons pour générer l'interruption.
- **CTRL** : Écrire dans le *BIT0* nous permet d'activer les fonctions de l'ADC
- **STEPCONFIG1 et STEPCONFIG2**: Nous permet de choisir le registre *FIFO* dans lequel nous enregistrons les données collectées. Nous permet de choisir le canal de lecture, pour les différentes étapes de lectures nous choisissons les canaux *AN0 AN1*. Nous pouvons choisir la moyenne d'échantillonnage, nous choisissons une moyenne de 16 échantillons.

Le champ *Mode* nous permet de choisir le mode d'échantillonnage, nous choisissons le mode *continuous* logiciel avec l'utilisateur de notre module timer.

- **STEPDELAY1 et STEPDELAY2** : Nous permet de choisir le nombre de cycles pour échantillonner, nous utilisons 1 cycle par échantillonnage.

- ***FIFO0COUNT*** : Nous permet de déterminer le nombre d'échantillons présent dans ***FIFO0***
- ***FIFO0THRESHOLD*** : Nous permet de définir le seuil pour lancer une interruptions avec le champ ***FIFO0_threshold_level*** avec un seuil de 32 échantillons.
- ***FIFO0DATA*** : Nous permet d'obtenir un echantillon lu

Dans notre implémentation du module ***ADC***, le module ***dmtimer*** et plus précisément le ***timer7*** est utilisé pour un temps de début et de fin pour l'opération d'acquisition des données.

La fonction ***startAdcAcq(...)*** nous permet de lancer une conversion des voltages present dans la grille et spécifier l'adresse de réception des données digitalisées. La fonction renvoie un int pour indiquer le statut de l'opération change le statut du drapeau ***acquireDone*** à la fin de la conversion. Une ***ISR (interrupt service routine)*** nous permettra d'enregistrer les résultats de l'ADC dans un tampon spécifié par ***starAdcAcq()***, cette ISR est déclenchée par l'une des files de type ***Hwi (hardware interrupt)*** contrôlé par le module ***Acquisition***. L'interruption se charge de changer le statut de ***acquireDone*** et de modifier le statut de l'interruption à l'aide du registre ***IRQSTATUS***.

3.9 GPIO Module Design

3.9.1 GPIO Hardware Module Overview

Les **GPIO** (General Purpose Input Output) ont comme but de contrôler les relais et adresser les ADC de la carte SRC. Comme on peut le voir dans le tableau ci-dessous, chaque relai est contrôlé par un canal différent d'un GPIO. Comme nous pouvons voir dans la section du design des modules relai.

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)

Le relai utilise des table 3 dimensions qui permettent le « **mapping** » d'un relai de port du côté source ou côté source ou côté charge en utilisant un numéro de port.

Les fonctions principales d'un module GPIO incluent l'initialisation de la bibliothèque, la configuration des pour l'entrée ou la sortie, la lecture ou l'écriture de données sur les broches, et la gestion des interruptions sur les broches d'entrée.

Dans notre conception, le GPIO est utilisé pour faire la gestion des broches. Il permet aussi de configurer les broches de 3 GPIOs (0, 1 et 2) pour contrôler les relais et adresser les ADC sur les relais et de définir les bits à activer, désactiver et tester les bits à l'aide de fonctions telles que bitSet, bitClear et isBitSet.

Les 4 registres pertinents pour configurer et manipuler les broches de sortie :

Premièrement, nous avons le registre oe (Output Enable) qui permet de configurer la broche de sortie car par défaut toutes les broches sont des broches d'entrées. Ensuite, nous avons le registre dataout permettant de définir les valeurs des broches de sorties du GPIO. En d'autres mots, ce registre permet de contrôler les broches de sorties. Cependant, il n'est pas nécessaire de manipuler directement dataout du coup on utilise cleardataout et setdataout. cleardataout a pour fonction de set à 0 tous les bits du dataout et enfin setdataout permet de définir à 1 bits dans dataout.

Conception :

1) L'accès aux registres en utilisant une variable de structure :

Les adresses des 4 GPIO sont déjà finies (GPIO0_REG 0x44E07000, GPIO1_REG 0x4804C000, GPIO2_REG 0x481AC000, GPIO3_REG 0x481AE000) dans le fichier gpio.Definitions.h. Pour accéder aux registres on doit donc trouver l'adresse physique du GPIO qu'on veut utiliser.

2) le « mapping » de la variable de structure à une adresse de GPIO (indice – casting à un pointeur). Accès aux registres des 4 périphériques (indice tableau de pointeur)

Pour faire le mapping de la variable de structure à une adresse de GPIO, on utilisera le casting à un pointeur. On va créer quatre variables GPIO0_REG0 , GPIO1_REG1, GPIO2_REG2, GPIO3_REG3 qui seront des pointeurs qui pointent à des adresses de contenus de types GPIO_REG (par exemple: (GPIO_REG *) GPIO0_REG = GPIO0_REG) d'où on pourrait alors créer un tableau de pointeur qui va stocker ces quatre variables pour accéder aux différents registres.

3) Broches de sorties utilisées dans le projet

Le module GPIO a quatre fonctions :

1. **Void initGPIO() :** Initialise les GPIOs

- Cette fonction active GPIO0, GPIO1 et GPIO2.

- Elle configure également les broches de sortie utilisées dans l'IDP pour contrôler les relais et adresser les ADC sur les relais.

Pour initialiser les GPIOs, on doit accéder à l'adresse mémoire des registres des broches et les définir comme entrée ou sortie. Pour initialiser des dispositifs GPIOs, on a l'ensemble d'identifiants suivant (défini dans le type énuméré PERIPHERAL_ID) :MC_GPIO0, MC_GPIO1, MC_GPIO2.

// Base address for each gpio hardware module.

#define GPIO0_REG 0x44E07000 //<! gpio0 hardware module address.

#define GPIO1_REG 0x4804C000 //<! gpio1 hardware module address.

#define GPIO2_REG 0x481AC000 //<! gpio2 hardware module address.



- 1) **SPI (Serial Port Interface):** The SPI ports are used to communicate with the relay cards. The SPI0 is connected to the source relays, while SPI1 is connected to the load relays.
- 2) **Void bitSet(int gpioNum, uint32_t bits) :** Définit les bits identifiés dans le paramètre bits. Définit les bits du registre de sortie de données GPIO à l'aide de setdataout.

Paramètres :

- **gpioNum :** GPIO0 ou GPIO1 ou GPIO2
 - **bits :** les bits mis à 1 dans le paramètre (bits) entraîneront la mise à 1 des mêmes bits dans le registre de sortie de données GPIO. Les bits à 0 dans le paramètre n'affectent pas le registre de sortie de données.
-
- 3) **Void bitClear(int gpioNUM, uint32_t bits):** Efface les bits identifiés dans le paramètre bits. Efface le registre de sortie de données GPIO à l'aide de cleardataout.

Paramètres :

- **gpioNum :** GPIO0 ou GPIO1 ou GPIO2
 - **bits :** les bits mis à 1 dans le paramètre (bits) entraînent l'effacement (mise à 0) des mêmes bits dans le registre de sortie de données GPIO. 0 bits dans le paramètre n'affecte pas le registre de sortie de données.
-
- 4) **Bool isBitSet(int gpioNUM, uint32_t bits):** Teste si les bits sont activés.

Vérifie si un ensemble de bits de sortie de données (un ou plusieurs) sont effacés. Test si les bits sont effacés dans le registre de données.

Paramètres :

- **gpioNum :** GPIO0 ou GPIO1 ou GPIO2
- **bits :** Un ou plusieurs bits sont définis dans le paramètre "bits"

Retours :

Retourne 0 (faux) si tous les bits définis (à 1) dans le paramètre "bits" sont effacés (définis à 0) dans le registre de données, sinon retourne une valeur non nulle (vraie).

Références

- [1] IDP Project Deployment File
- [2] AM355x Sitara Processor Datasheet

[3]AM335x and AMIC110 Sitara™ Processors Technical Reference Manual