Intelligence Artificielle Embarqué

Sebastian Marzetti & Valentin Gies

1 Présentation du TP

Le TP présente une initiation à l'intelligence artificielle (IA) embarqué. Un cas d'apprentissage non supervisé est développé à l'aide de l'algorithme Unsupervised Clustering with Expectation Maximization.

L'algorithme sera implémenté sur un microcontrôleur ARM Cortex M4F intégré dans un System on Chip (SoC) CC2652 de Texas Instruments. Le microcontrôleur dispose d'un système d'exploitation en temps réel (RTOS).

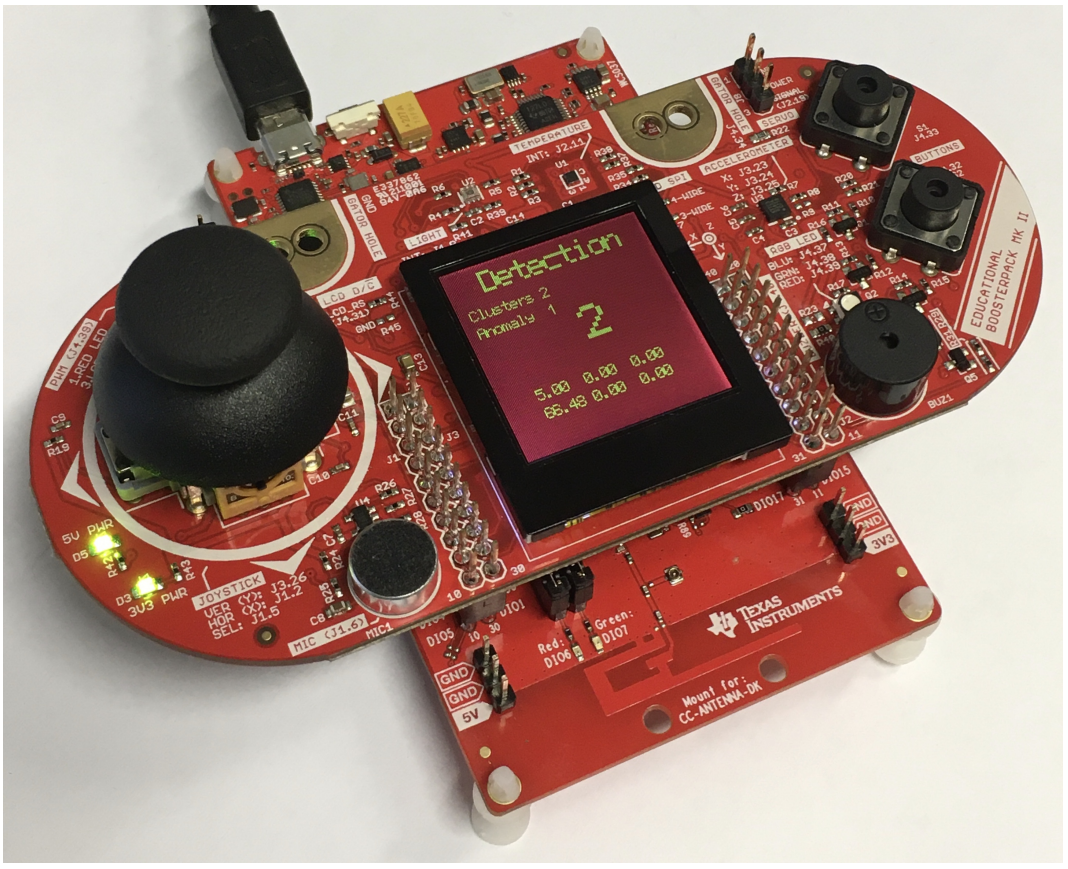

FIGURE 1 – TP : Intelligence Artificielle Embarqué

1.1 Système à implémenter

La figure 2 présente le schéma synoptique du système qui sera implémenté. L'algorithme d'IA va apprendre de façon non supervisée les modes de vibration du système.

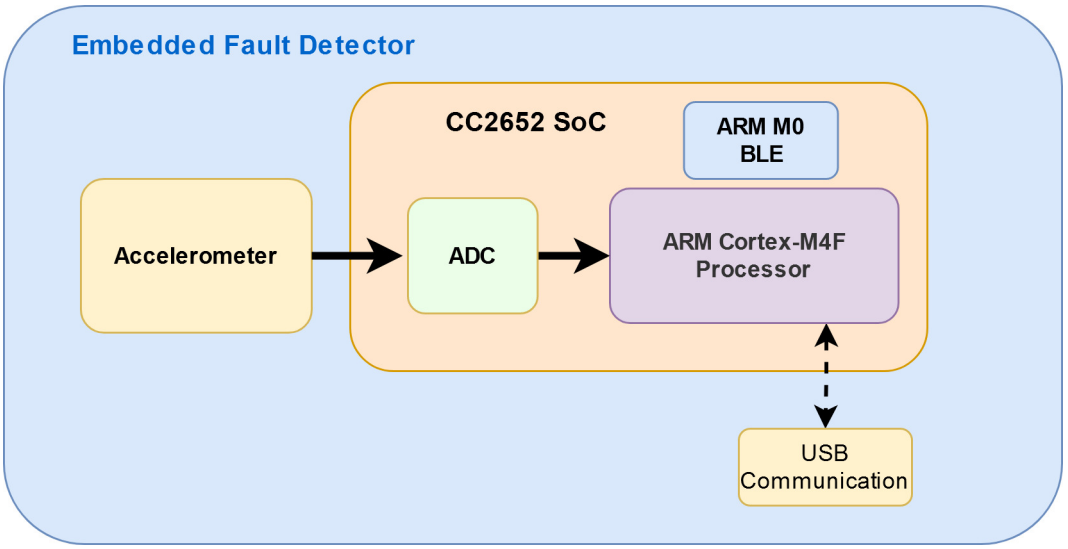

FIGURE 2 – Diagramme de bloques du système

Pour mesurer les vibrations, un accéléromètre analogique est utilisé. Les signaux de sortie mesurés par le capteur sont acquises à l'aide d'un convertisseur analogique numérique (ADC).a partir de ces acquisitions, des caractéristiques (features) sont extraites, elles servent d'entrée à l'algorithme d'apprentissage non-supervisé du système.

Un cas d'application typique pour ce TP est la détection des défauts sur une machine tournant. L'algorithme va apprendre les modes de fonctionnement de la machine, par exemple, une machine avec deux vitesses de fonctionnement va générer deux modes de fonctionnement dont est issue le nombre de clusters détectés par l'algorithme. Si des fonctionnement anormaux se produisent, des anomalies seront détectés, puisque les features extraites n'appartient pas aux clusters existants.

2 Installation et prise en main de l'environnement de développement

L'environnement de développement est composé de 2 logiciels couplés :

1. Code Composer Studio (CCS) : CCS permet de gérer des projets utilisant des microcontrôleurs de chez Texas Instruments.
2. SimpleLink : SimpleLink est une plateforme de microcontrôleurs qui dispose d'un portefeuille d'unités SoC (system-on-chip) Arm® câblées et sans fil dans un seul environnement de développement de logiciel à partir d'un code C.

Ces deux logiciels permettent donc de programmer des microcontrôleurs de Texas Instruments à l'aide d'un code en langage C.

2.1 Installation de Code Composer Studio

Cette partie peut être optionnelle si les logiciels sont déjà installés sur votre machine. Elle est toutefois présentée afin que vous puissiez réaliser l'installation sur vos ordinateurs (les logiciels sont gratuits) pour travailler sur vos projets.

$\Longrightarrow$ Téléchargez les fichiers d'installation des 2 logiciels dans leurs dernières versions à l'aide des liens suivants :

— Code Composer Studio
— SimpleLink CC13x2-26x2 SDK

$\Longrightarrow$ Installer si ça n'est pas déjà fait Code Composer Studio dans une version supérieure à la version 10.0.0, en laissant les options par défaut, sauf quand il demande sur la sélection de composants. Ajouter *SimpleLink CC13xx and CC26xx Wireless MCUs*, comme montre la figure 3. Redémarrer si besoin.

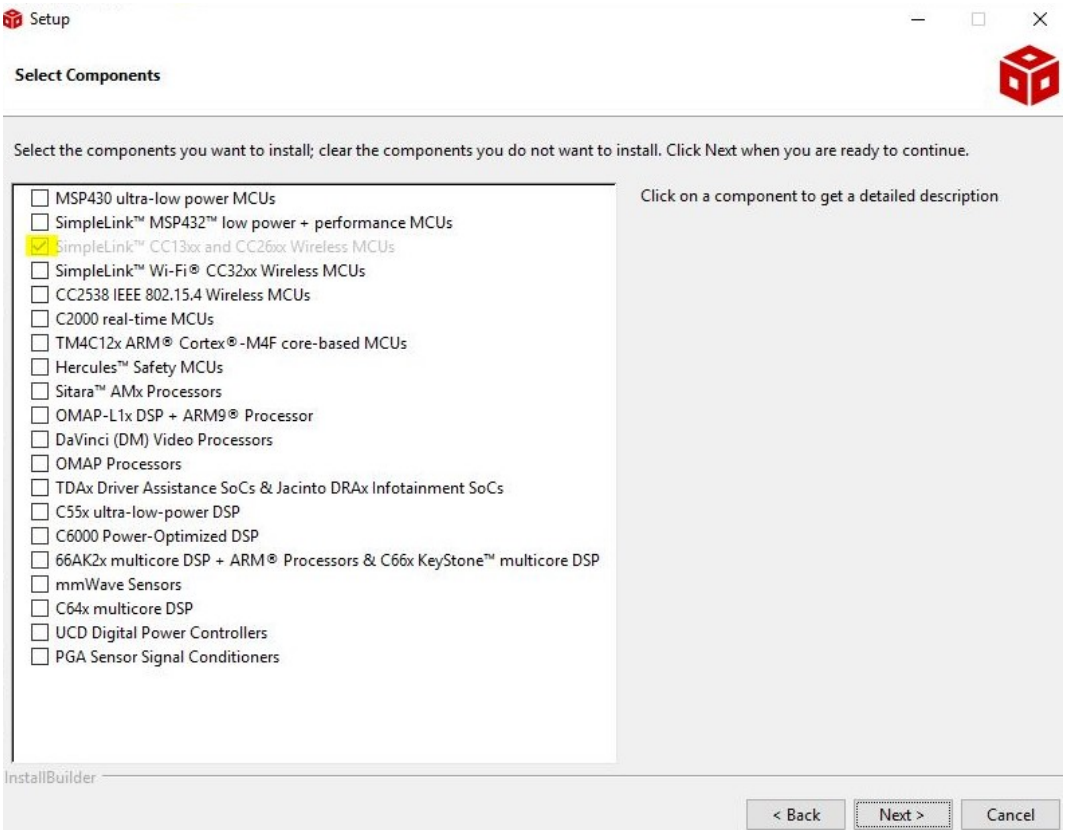

FIGURE 3 – Installation Code Composer Studio

⟹ Installer ensuite SimpleLink CC13x2-26x2 SDK dans une version supérieure à la version 3.40.00 si ça n'est pas déjà fait, en laissant également les options par défaut. SimpleLink sera intégré automatiquement à CCS, sans action de votre part.

⟹ Au lancement de CCS, un message peut vous demander le *workspace* à utiliser. Spécifiez *C :/Embedded_AI/*. Il est à noter qu'il ne faut surtout pas laisser d'espace dans le nom du répertoire de travail car sinon les compilations ne fonctionneront pas sous CCS.

⟹ Branchez à présent le kit de développement *LAUNCHXL-CC26X2R1* sur le PC. Il doit être reconnu par Windows. En cas de problème demandez au professeur.

3 Premier programme

Cette partie utilise le kit de développement *LAUNCHXL-CC26X2R1*, qui est décrit en détail sur le site internet de chez Texas Instruments LAUNCHXL-CC26X2R1. C'est donc cette carte qu'il faudra programmer, qui intègre le SoC CC2652R1. Le programmateur est intégré aussi dans la carte donc il faudra juste connecter le kit de développement par USB.

Vous allez utiliser aussi un deuxième kit de développement appelé *BOOSTXL-EDUMKII* décrit un détail aussi sur le site de chez Texas Instruments BOOSTXL-EDUMKII.

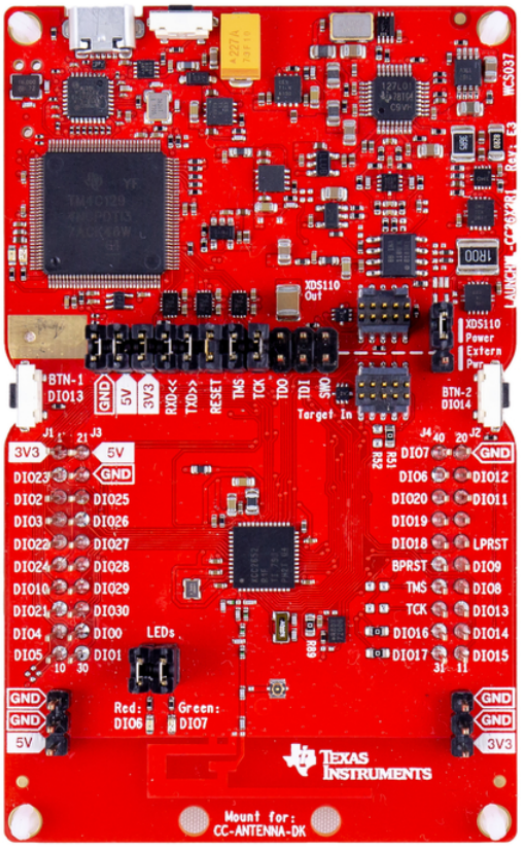

FIGURE 4 – LAUNCHXL-CC26X2R1

Le SoC (System on Chip) CC26x2R1 utilisé est décrit en détail sur le site internet de chez Texas Instruments : CC2652R.

Vous trouverez sur le site du constructeur toutes les informations nécessaires au fonctionnement de chacun des systèmes du SoC avec des exemples de code. Dans le dossier *SimpleLink* qu'on vient d'installer il y a aussi des exemples déjà implémentés permettant de tester tous les drivers.

3.1 Création du projet

$\Longrightarrow$ Vous allez utiliser un projet de base servant normalement à créer un objet connactable en BLE. Pour cela lancez CCS, puis allez dans :

Project → Import CCS Projects → Browse

$\Longrightarrow$ Cherchez le dossier *simplelink_cc13x2_26x2_sdk_X_XX_XX_XX*, par défaut il est installé sur *C :/ti*. puis allez sur le projet «simple peripheral» et sélectionnez le dossier *CCS* qui est dans le dossier : *tirtos* :

C :/ ti/ simplelink_cc13x2_26x2_sdk_X_XX_XX_XX/ examples/ rtos/
CC26X2R1_LAUNCHXL/ ble5stack/ simple_peripheral/ tirtos/ ccs

Le projet devrait être sur l'explorateur de projets comme il est montré dans la figure 5.

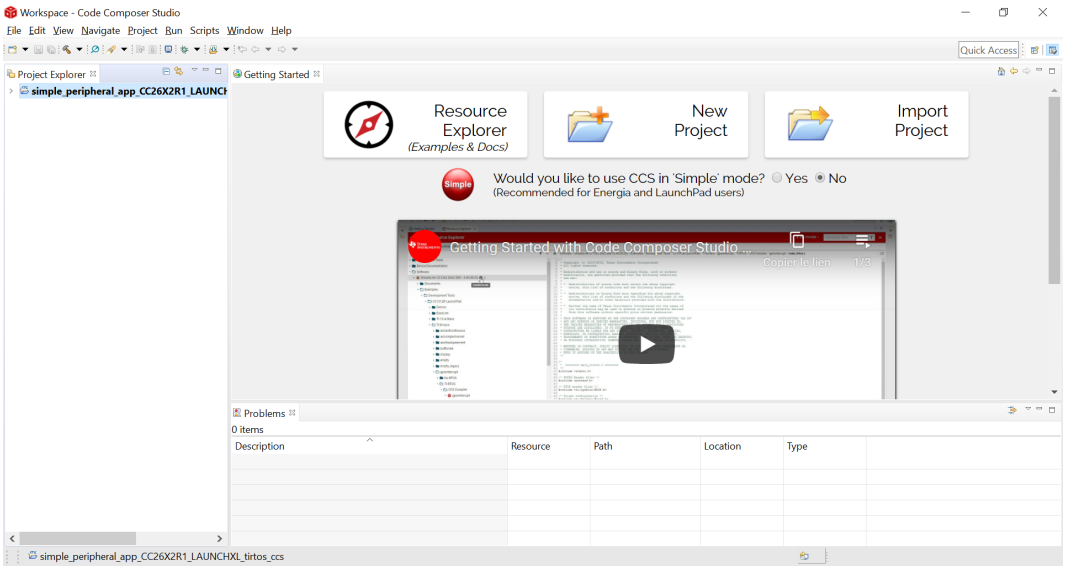

FIGURE 5 – Projet «simple peripheral» importé

Si votre projet a été bien importé vous pouvez le compiler avec l'icône représentant un marteau ou avec :

Project → Build Project

A l'issue de la compilation il ne devrait pas y avoir des erreurs donc flashez le SoC avec (icône a droite pour flash et icône a gauche pour debug). Si il y a des erreurs, demandez au professeur.

La fenêtre *Project Explorer* contient tous les projets ouverts ainsi que tous les fichiers de chaque projet. Le fichier *simple_peripheral.syscfg* nous permet de configurer facilement les drivers que nous allons utiliser (Fig. 6).

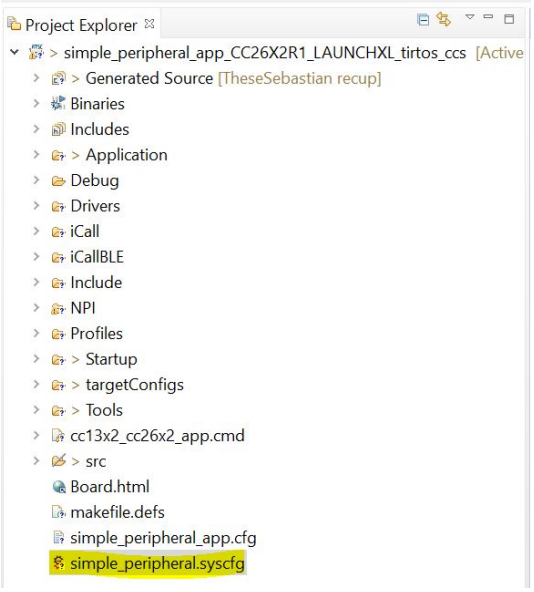

FIGURE 6 – Project Explorer

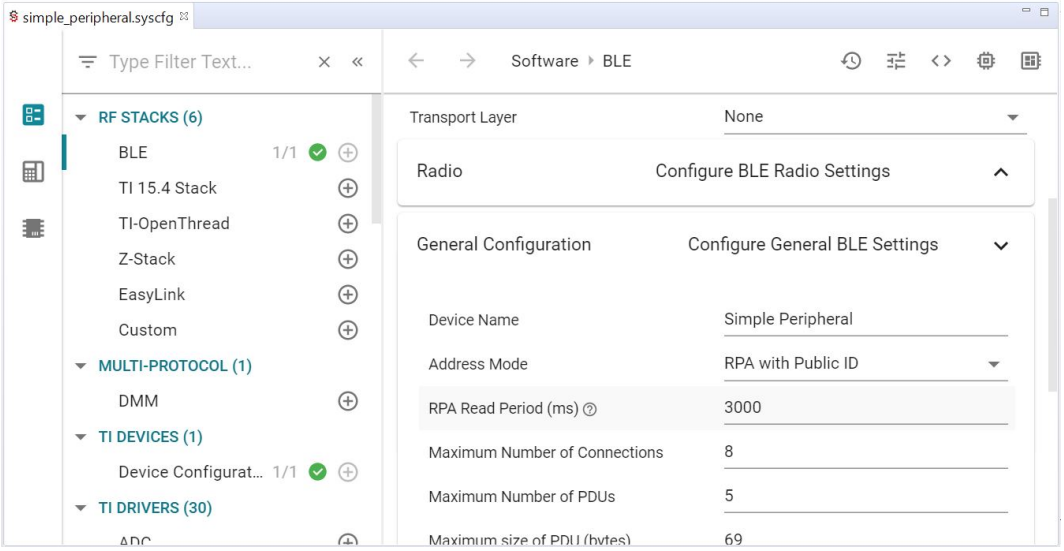

FIGURE 7 – Fichier Simple_Peripheral.syscfg

3.2 Création de la tâche ADC

Après avoir vérifié le bon fonctionnement du compilateur, vous allez créer une tâche pour l'échantillonnage de l'accéléromètre intégré sur le kit BOOSTXL-EDUMKII (figure 8), après avoir désactivé la tache BLE de base.

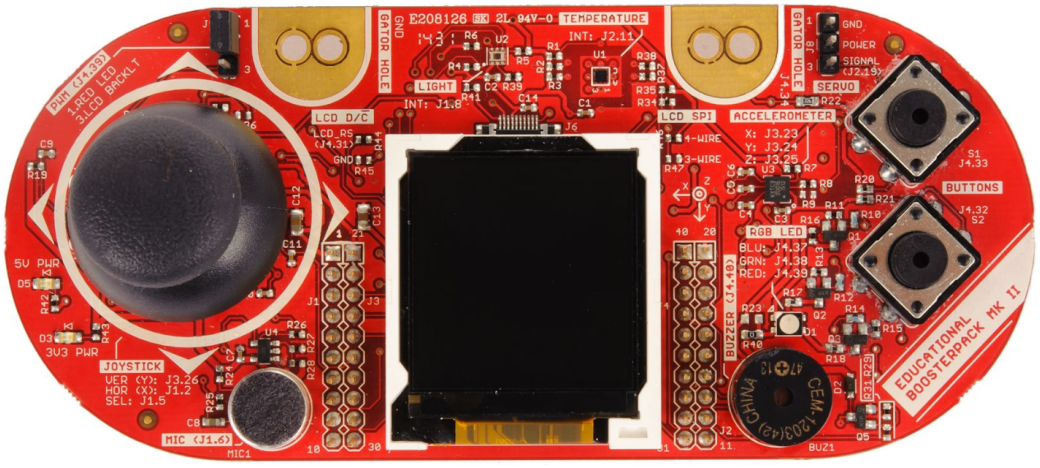

FIGURE 8 – BOOSTXL-EDUMKII

⟹ Désactivez l'appel de la fonction *SimplePeripheral_createTask* dans le fichier *main* du répertoire *Startup*.

⟹ Créez un nouveau dossier appelé *TacheADC* dans le projet en effectuant un click droit sur le projet et ensuite :

New → Folder
Folder Name : TacheADC → Finish

⟹ Créez un fichier .h et un fichier .c appelés *TacheADC* sur le dossier qu'on vient de créer en effectuant un click droit sur le dossier *TacheADC* et puis :

New → Header File
Header file : TacheADC.h → Finish
New → Source File
Source file : TacheADC.c → Finish

⟹ En commençant par le fichier source que vous venez de créer, inclure les lignes de code suivantes afin d'intégrer les appels aux bibliothèques standards et utilisées par le RTOS :

```
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <math.h>

#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Event.h>
#include <ti/sysbios/knl/Queue.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/BIOS.h>
#include <TacheADC/TacheADC.h>
```

⟹ A présent, configurez la tâche avec une priorité de 3 (Le maximum est 5) et une taille de stack de 1024 octets. Pour cela ajoutez les lignes de code suivantes au fichier source à la suite du code précédent. Ce code déclare la structure de la tâche *TacheADC* et sa stack, ainsi qu'un sémaphore permettant de déclencher la tâche lorsque cela est nécessaire :

```
        #define TacheADC_TASK_PRIORITY 3
        #define TacheADC_TASK_STACK_SIZE 1024

        Task_Struct TacheADC;
        uint8_t TacheADCStack[TacheADC_TASK_STACK_SIZE];

        Semaphore_Struct semTacheADCStruct;
        Semaphore_Handle semTacheADCHandle;
```

⟹ Ajoutez à présent la fonction principale de la tâche au même fichier source. Elle est constituée comme souvent pour une tâche en RTOS d'une boucle infinie :

```
void TacheADC_taskFxn(UArg a0, UArg a1)
{
        for (;;)
        {
        }
}
```

Une fois les structures de la tâche ADC et du sémaphore déclarées, il faut les instancier dans une fonction de création de la tâche qui sera appelé une seule fois à la création de celle-ci. Elle doit être appelée une seule fois depuis la fonction main() du code principal (dossier Startup).

⟹ Pour cela rajoutez le code suivant, toujours dans le fichier *TacheADC.c*, ainsi qu'un appel à cette fonction dans le fichier *main.c* après la ligne *SimplePeripheral_createTask()*. Vous penserez également à rajouter les prototypes de fonctions du fichier TachADC.c dans le header correspondant, et à inclure ce header dans le main à l'aide du code *#include "./TacheADC/TacheADC.h"* :

```
void TacheADC_CreateTask(void){

        Semaphore_Params semParams;
        Task_Params taskParams;

        // Configure task
        Task_Params_init(&taskParams);
        taskParams.stack = TacheADCStack;
        taskParams.stackSize = TacheADC_TASK_STACK_SIZE;
        taskParams.priority = TacheADC_TASK_PRIORITY;

        Task_construct(&TacheADC, TacheADC_taskFxn,
                       &taskParams, NULL);
```

```
        /* Construct a Semaphore object
        to be used as a resource lock, initial count 0 */
        Semaphore_Params_init(&semParams);
        Semaphore_construct(&semTacheADCStruct, 0, &semParams);
        /* Obtain instance handle */
        semTacheADCHandle = Semaphore_handle(&semTacheADCStruct);
}
```

Vous venez de créer une tâche avec utilisant un sémaphore, cette méthode est générique et vous permettra de créer n'importe quel type de tâche et l'adapter à vos besoins. Dans le cas où le sémaphore ne serait pas nécessaire, vous pouvez bien entendu retirer tout ce qui le concerne.

La tâche que vous avez créé est dédiée à l'échantillonnage d'un accéléromètre analogique, par conséquent vous devez pour que celui-ci fonctionne initialiser le driver ADC. L'accéléromètre dispose de 3 axes (X, Y et Z) donc 3 signaux analogiques sont générés.

⟹ Pour effectuer la déclaration et initialisation des drivers, allez dans notre fichier de configuration (*simple_peripheral.syscfg*) et nous ajoutons 3 drivers ADC avec le bouton *ADD* (Fig. 9) pour échantillonner ces signaux. Puis sur *Peripheral and Pin Configuration* on l'assigne a chaque voie un pin. Vérifier les pins de l'accéléromètre qui sont connectés aux pins *DIO25* (*DIO25/23 (Header)*), *DIO26* (*DIO26/24 (Header)*) et *DIO27* (*DIO27/25 (Header)*) du kit de développement.

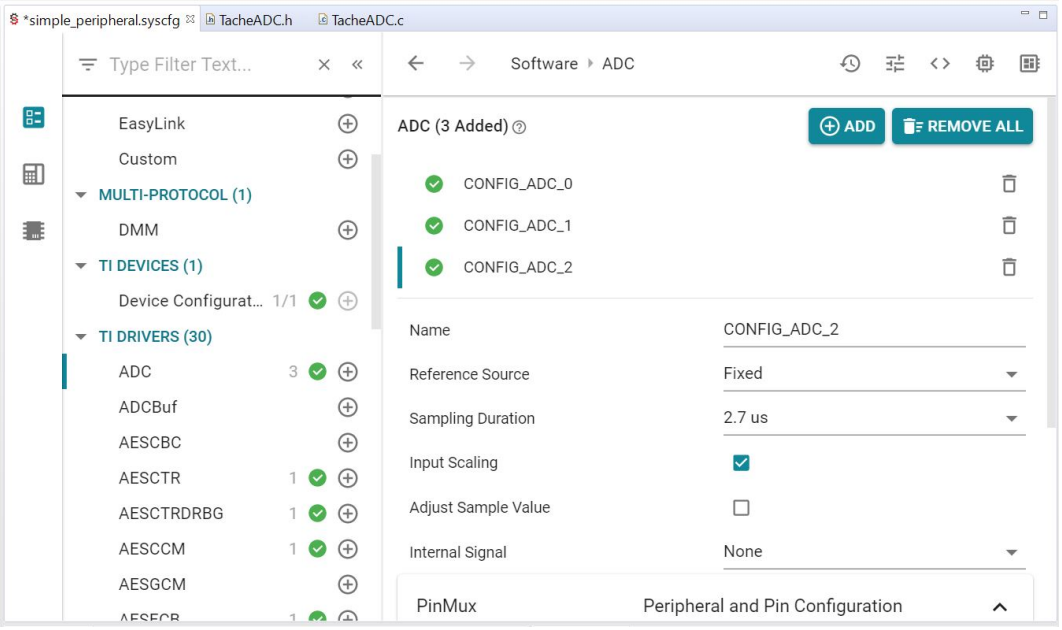

FIGURE 9 – Fichier Simple_Peripheral.syscfg add ADC driver

Une fois cette configuration effectuée, il est nécessaire de déclencher les acquisitions périodiquement. Pour cela un timer, dénommé *Clock* dans *TI RTOS* va périodiquement poster (déclencher) le sémaphore *semTacheADCHandle*. Les conversions, lancées dans la boucle infinie de la tâche *TacheADC* devront quand à elle attendre que le sémaphore soit déclenché pour que les conversions soient effectuées successivement sur les trois canaux ADC.

⟹ Le timer doit être instancié dans la fichier *TacheADC.c* juste après les sémaphores à l'aide du code suivant :

```
        static Clock_Struct myClock;
```

⟹ Le code permettant de configurer le Timer est le suivant, il est à placer avant la boucle infinie de la fonction *TacheADC_taskFxn* du fichier *TacheADC.c* :

```
        // Declaration d'une structure clock_Params
        Clock_Params clockParams;
        // Initialisation de la structure
```

```
        Clock_Params_init(&clockParams);
        // Reglage de la periode a 10 ms
        clockParams.period = 10 * (1000/Clock_tickPeriod);
        // Initialisation du timer (Clock en RTOS)
        Clock_construct(&myClock, myClockSwiFxn, 0, &clockParams);
        //Lancement du timer
        Clock_start(Clock_handle(&myClock));
```

⟹ Vous devez également ajouter au fichier *TacheADC.c* la fonction *myClockSwiFxn* lancée par le timer à chaque période. Elle post à intervalle régulier le sémaphore de lancement des conversions ADC. Pensez également à rajouter son prototype au fichier *TacheADC.h* :

```
void myClockSwiFxn(uintptr_t arg0)
{
        Semaphore_post(semTacheADCHandle);
}
```

A ce stade, le code doit compiler.

⟹ Testez le en debug et vérifiez que vous passer bien dans la fonction *myClockSwiFxn* en rajoutant un point d'arrêt. Ne continuez pas la suite tant que vous n'arrivez pas à atteindre le point d'arrêt.

Vous allez à présent utiliser les post effectués sur le sémaphore pour déclencher les conversions sur l'ADC.

⟹ Commencez par rajouter les appels aux bibliothèques de l'ADC en début de *TacheADC.c* à l'aide du code suivant :

```
        /* Driver Header files */
        #include <ti/drivers/ADC.h>
        /* Driver configuration */
        #include "ti_drivers_config.h"
```

⟹ Ajoutez également un appel à la fonction d'initialisation de l'ADC avant la boucle infinie de la fonction *TacheADC_taskFxn* du fichier *TacheADC.c* :

```
        //Initialisation du module ADC
        ADC_init();
```

⟹ Ajoutez à présent la fonction d'échantillonnage et conversion de l'ADC à *TacheADC.c*, en pensant à ajouter son prototype au fichier header. Son code est le suivant, il est à noter que les fonctions d'ouverture, fermeture et conversion de l'ADC sont inclues dans la bibliothèque du driver ADC :

```
uint32_t Sampling(uint_least8_t Board_ADC_Number){
        ADC_Handle adc;
        ADC_Params params;
        ADC_Params_init(&params);
        uint16_t adcValue;
        uint32_t adcValue1MicroVolt;

        adc = ADC_open(Board_ADC_Number, &params);
        ADC_convert(adc, &adcValue);
        adcValue1MicroVolt = ADC_convertRawToMicroVolts(adc, adcValue);
        ADC_close(adc);

        return adcValue1MicroVolt;
}
```

⟹ Pour lancer les conversions, il ne vous reste plus qu'à les déclencher sur arrivée d'un post sur le sémaphore dans la boucle infinie de la tâche, en intégrant le code suivant :

```
        Semaphore_pend(semTacheADCHandle, BIOS_WAIT_FOREVER);
```

```
uint32_t DatasampledX = Sampling(CONFIG_ADC_0);
uint32_t DatasampledY = Sampling(CONFIG_ADC_1);
uint32_t DatasampledZ = Sampling(CONFIG_ADC_2);
```

Vous devriez normalement avoir un code fonctionnel à présent pour réaliser les conversions ADC.

$\Longrightarrow$ Pour cela testez le code en ajoutant un breakpoint dans la fonction de conversion ADC à la ligne *ADC_close(adc) ;*, en ajoutant une watch expression (click droit sur la variable) sur *adcValue*. A chaque passage sur le point d'arrêt, vous devriez voir sa valeur changer de manière périodique toutes les 3 mesures, correspondant aux axes X, Y et Z. Vous pouvez vérifier que l'accéléromètre fonctionne bien en changeant l'orientation de la carte, la valeur correspondant à $0G$ étant autour de 1580.

3.3 Création de la tâche LCD

Dans cette partie, vous allez à présent afficher les informations issues de l'accéléromètre sur le LCD intégré dans le kit BOOSTXL-EDUMKII et présenté à la figure 8. L'implémentation de cette tache n'apportant en terme d'IA embarquée à ce TP, le code vous est donc fourni.

$\Longrightarrow$ Importez les fichiers relatifs au LCD téléchargeables ici (Lien de téléchargement) dans un nouveau dossier dénommé *TacheLCD* (Fig. 10).

$\Longrightarrow$ Appelez dans le *main* la fonction de création de la tâche dans *main.c* de la même manière que vous l'avez fait pour la tâche ADC, en n'oubliant pas d'ajouter un include vers le header *TacheLCD.h*.

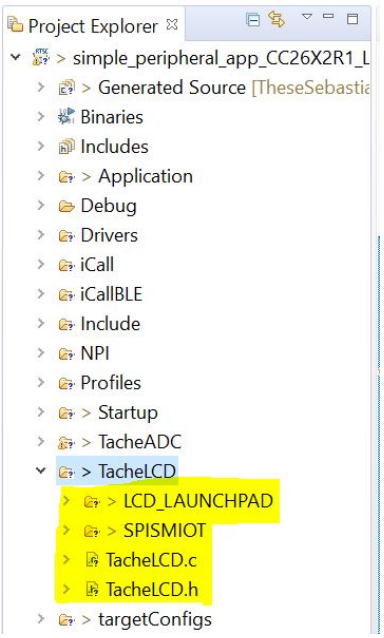

FIGURE 10 – Fichiers Tâche LCD

La communication avec le LCD est réalisée à l'aide d'une liaison SPI, vous allez donc configurer le driver de cette liaison pour pouvoir l'utiliser.

$\Longrightarrow$ Pour cela, ouvrez à nouveau le fichier (*simple_peripheral.syscfg*) et ajoutez un driver SPI (Fig. 11), en assignat les pins *SCLK* au DIO10, *MISO* au DIO8 et *MOSI* au DIO9 (Fig 12).

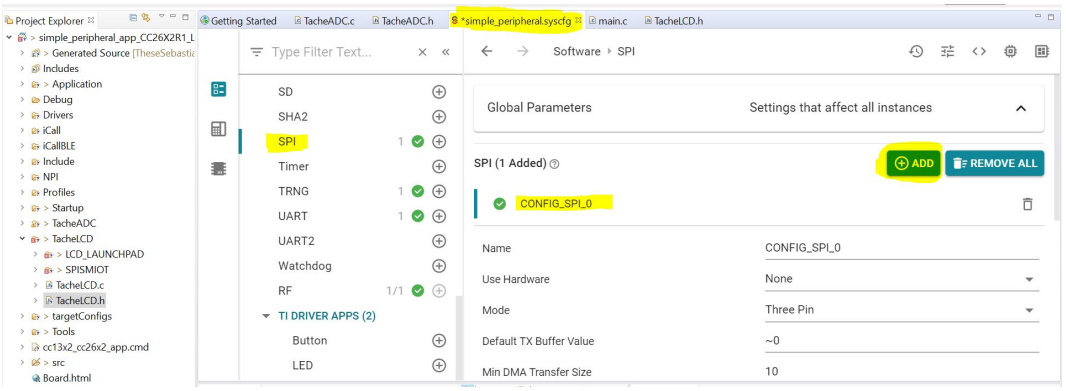

FIGURE 11 – Driver SPI

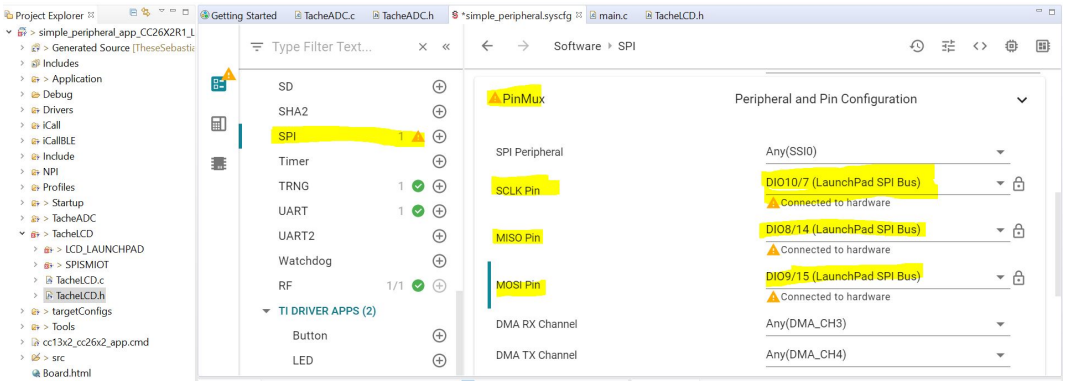

FIGURE 12 – Pins SPI

⟹ La liaison SPI et le LCD nécessitent également un *chip select* et un *chip reset*. Ajoutez ces deux sorties dans la partie GPIO du fichier de configuration : la première est appelée *SPI_LCD_CS* et utilise la pin DIO13 et la seconde est dénommée *SPI_LCD_RS* et est assignée à la pin DIO17 (Fig. 13). Toutes deux doivent être configurées en sortie (OUTPUT).

⟹ Un conflit avec la pin *CONFIG_GPIO_BTN1* devrait apparaitre. Pour le résoudre, il est nécessaire de désactiver cette entrée en passant sa priorité *Use Hardware* à *None*.

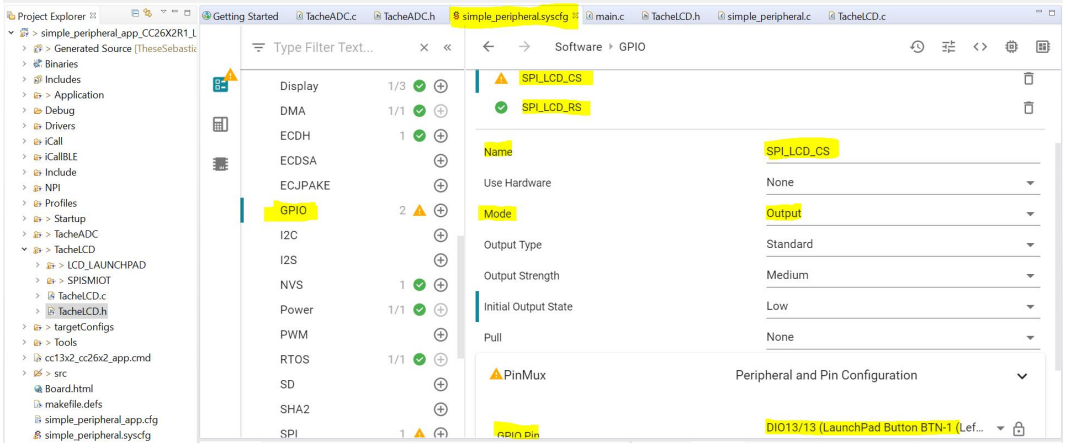

FIGURE 13 – Pins GPIO LCD

⟹ Testez votre code à ce stade, le LCD devrait afficher un message d'accueil *UTLN AI*.

L'affichage dynamique sur le LCD est basé sur une boucle infinie avec une attente sur un sémaphore comparable à celui utilisé dans la partie ADC. La différence est que le post du semaphore est effectué au sein de la fonction suivante :

```
LCD_PrintState(int State, float NumberOfClusters,
int ClusterDetected, float NumberOfAnomalies,
float Array[6], int ArraySize);
```

Elle vous permettra ultérieurement d'afficher l'état de votre algorithme d'apprentissage non supervisé, chaque paramétre de cette fonction étant affiché comme montre la figure **??**.

$\Longrightarrow$ Afin d'afficher les données de l'accéléromètre sur l'écran, vous allez tout d'abord convertir les données brutes issues de l'acquisition en G à l'aide de la fonction uVToGfloat est donné ci-dessous. N'oubliez pas de déclarer son prototype dans le fichier header.

```
float uVToG_float(uint32_t dataSampled)
{
float dataG = ((float)dataSampled - 1650000)/660000;
return dataG;
}
```

Les données accélérométriques converties en G seront appelées dans le code : xG, yG et zG.
$\Longrightarrow$ Pour transmettre les données au LCD, vous devez placer les données de l'accéléromètre converties en G dans un tableau de float dénommé *features* , avant d'appeler la fonction *LCD_PrintState* comme suit :

```
float features[6];
features[0]= xG;
features[1]= 0;
features[2]= yG;
features[3]= 0;
features[4]= zG;
features[5]= 0;
LCD_PrintState(0, 0, 0, 0, features, 6);
```

Ne pas oublier d'inclure le fichier header de la tâche LCD dans la tâche ADC (Fig. 14).

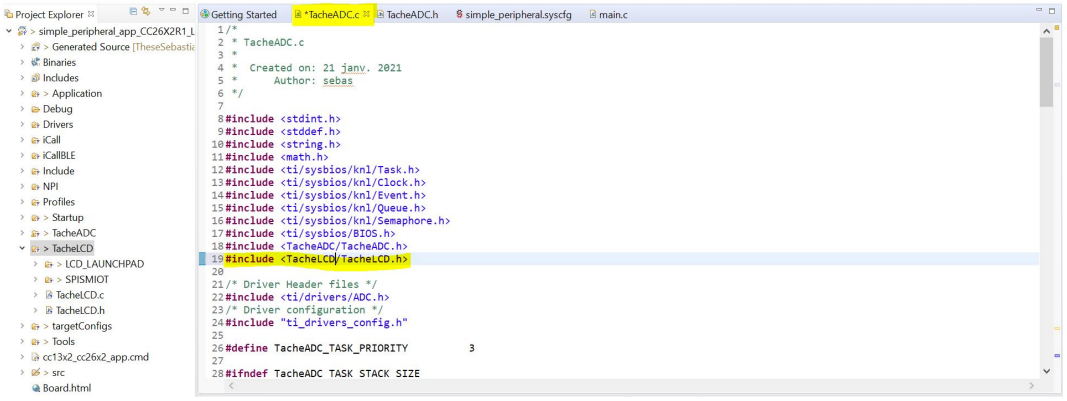

FIGURE 14 – Tâche ADC

A ce stade, votre projet devrait compiler et afficher la valeur de la gravité (en G) lue sur chacun des 3 axes de l'accéléromètre comme indiqué à la figure 15 quand la carte est statique.

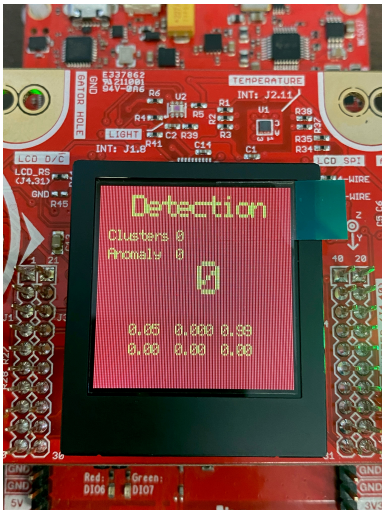

FIGURE 15 – Affichage des données ADC sur le LCD

⟹ Validez l'ensemble et observer que les changements de valeurs de l'accéléromètres sont instantanés si vous tournez la carte sur elle même.

3.4 Extraction de features pour la classification

Après avoir vérifié les données obtenues avec le convertisseur analogique numérique, il est à présent temps d'en extraire des informations utiles, les *features*, qui serviront à l'apprentissage non-supervisé dans la dernière partie.

Pour extraire les features nécessaires dans notre algorithme d'intelligence artificiel, nous allons utiliser les opérateurs suivants :
- Filtre passe haut
- Norme tri-axiale
- FFT
- Détection de pics sur la FFT

Ces opérateurs seront implémentés dans l'ordre montré dans la figure 16.

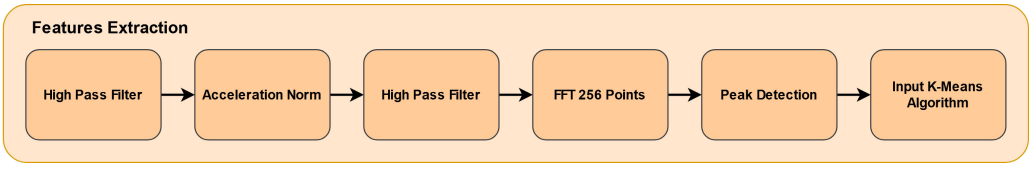

FIGURE 16 – Extraction de features

Vous allez coder certains de ces opérateurs pour vous familiariser avec le traitement de signal numérique embarqué. En particulier, vous allez coder les filtres passe-haut du premier ordre utilisés dans cet algorithme.

⟹ Pour vous entrainer et vérifier que tout se passe bien, vous allez tout d'abord implanter 3 filtres passe-bas numériques d'ordre 1 sur chacun des 3 signaux issus de l'accéléromètre. Pour cela commencez par ajouter à votre projet le répertoire *Filters* téléchargeable ici : Liens de téléchargement, et examiner comment sont implantés ces filtres. Vous penserez à bien décrire le fonctionnement de la bibliothèque dans votre rapport, et expliquer comment les formules utilisées ont été obtenues.

⟹ Dans le fichier *ADC.c*, déclarez les trois filtres à l'aide du code suivant :

```
Order1Filter LPFilterAccelX;
Order1Filter LPFilterAccelY;
Order1Filter LPFilterAccelZ;
```

⟹ Dans la partie initialisation de la tâche ADC du fichier *TacheADC_taskFxn*, initialisez les filtres en tant que filtres passe-bas du premier ordre comme suit :

```
// Initialisation des filtres
InitOrder1LPFilterEuler(&LPFilterAccelX, 1, 100);
InitOrder1LPFilterEuler(&LPFilterAccelY, 1, 100);
InitOrder1LPFilterEuler(&LPFilterAccelZ, 1, 100);
```

⟹ Dans la boucle infinie de la fonction *TacheADC_taskFxn*, après avoir converti les données acquises sur l'accéléromètre en G, filtrez-les et affichez-les à l'aide du code suivant :

```
//Filtre passe-bas sur les 3 axes
float AccelLPX = ComputeOrder1Filter(&LPFilterAccelX, xG);
float AccelLPY = ComputeOrder1Filter(&LPFilterAccelY, yG);
float AccelLPZ = ComputeOrder1Filter(&LPFilterAccelZ, zG);

float features[6];
features[0]= AccelLPX;
features[1]= 0;
features[2]= AccelLPY;
features[3]= 0;
features[4]= AccelLPZ;
features[5]= 0;

LCD_PrintState(0, 0, 0, 0, features, 6);
```

⟹ Validez le bon fonctionnement de l'ensemble : la valeur de l'accélération devrait varier très progressivement (la fréquence de coupure du filtre étant de 1Hz) au lieu de changer brusquement comme précédemment.

Vous allez à présent implanter la première étape de l'extraction de *features* : les filtres passe-haut.

⟹ Ajouter au fichier *Filter.c*, une fonction *InitOrder1HPFilterEuler* semblable à celle du filtre passe-bas mais permettant cette fois d'initialiser un filtre passe-haut d'ordre 1 généré à l'aide de l'approximation d'Euler. Pensez à ajouter son prototype au fichier header. Il est à noter qu'il ne sera pas nécessaire ensuite de modifier la fonction *ComputeOrderAFilter* pour l'utiliser...

⟹ Déclarez 3 filtres passe-haut dans le fichier ADC.c, puis initialisez-les avec une fréquence de coupure de 1Hz, et testez-les sur les signaux xG, yG, zG issus de l'accéléromètre après conversion.

⟹ Vous devriez normalement avoir des valeurs qui augmentent soudainement lors d'un changement d'orientation de la carte, avant de revenir à 0 si vos filtres sont bien implantés.

Vous allez à présent implanter le calcul de la norme d'accélération tri-axiale, filtrer la valeur à l'aide d'un filtre passe-haut et placer le résultat dans un tableau qui servira ensuite au calcul de la FFT.
⟹ Pour cela, déclarez dans le fichier *ADC.c* un nouveau filtre passe-haut du premier ordre dénommé *HPFilterAccelNorme* (en pensant à l'initialiser), ainsi que des tableaux destinés à bufferiser les données nécessaires au calcul ultérieur de la FFT à l'aide du code suivant :

```
Order1Filter HPFilterAccelNorme;

#define FFT_WINDOW_SIZE 256
float SerieNormeAccel[FFT_WINDOW_SIZE];
int indexFFT;
```

⟹ Effectuez à présent les calculs de la norme et le remplisage des buffers utilisés pour la FFT à l'aide du code suivant. L'appel de la fonction de calcul de la FFT et de la classification est commenté dans ce code, car pas encore implanté :

```
float normeAccel = sqrtf(AccelHPX*AccelHPX+AccelHPY*AccelHPY+AccelHPZ*AccelHPZ);
float normeAccelHP = ComputeOrder1Filter(&HPFilterAccelNorme, normeAccel);
```

```
SerieNormeAccel[indexFFT] = normeAccelHP;
indexFFT++;

if(indexFFT>=FFT_WINDOW_SIZE)
{
      //On lance la tache de calcul de la FFT et classification
      FFTClassificationTrigger(SerieNormeAccel);
      //Le resultat est recupere dans DataYFFT
      indexFFT = 0;
}
```

3.5 Création de la tâche de calcul de la FFT et classification

Dans cette partie, vous allez créer une tâche destinée à effectuer le calcul de la FFT, la détectiond des pics principaux de celle-ci et de leur amplitude, et la classification non supervisée. L'ensemble de ces opérations est rassemblé dans une seule et même tâche, différente de celle des acquisitions de données sur l'accéléromètre car ces opérations sont appelés 256 fois moins souvent que les conversion ADC et filtrages, mais elles durent beaucoup plus lontemps. Cette tâche doit donc être moins prioritaire que les précédentes.

$\Longrightarrow$ Dans un dossier *TacheFFTClassification* que vous allez créer, ajoutez des fichiers source *TacheFFTClassification.c* et header *TacheFFTClassification.h*.

$\Longrightarrow$ Dans le fichier source, ajoutez le code suivant, qui est un code minimal pour créer une tâche ayant un sémaphore de déclenchement. Examiner ce code et commentez le dans votre rapport :

```
#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Event.h>
#include <ti/sysbios/knl/Queue.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/BIOS.h>

/* Driver configuration */
#include "ti_drivers_config.h"

#include "TacheFFTClassification.h"

#define TacheFFTClassification_TASK_PRIORITY 2
#define TacheFFTClassification_TASK_STACK_SIZE 1024

Task_Struct TacheFFTClassification;
uint8_t TacheFFTClassificationStack[TacheFFTClassification_TASK_STACK_SIZE];

Semaphore_Struct semTacheFFTClassificationStruct;
Semaphore_Handle semTacheFFTClassificationHandle;

void TacheFFTClassification_CreateTask(void){

      Semaphore_Params semParams;
      Task_Params taskParams;

      // Configure task
      Task_Params_init(&taskParams);
      taskParams.stack = TacheFFTClassificationStack;
      taskParams.stackSize = TacheFFTClassification_TASK_STACK_SIZE;
      taskParams.priority = TacheFFTClassification_TASK_PRIORITY;

      Task_construct(&TacheFFTClassification, TacheFFTClassification_taskFxn,
      &taskParams, NULL);

      /* Construct a Semaphore object
```

```
        to be used as a resource lock, initial count 0 */
        Semaphore_Params_init(&semParams);
        Semaphore_construct(&semTacheFFTClassificationStruct, 0, &semParams);
        /* Obtain instance handle */
        semTacheFFTClassificationHandle = Semaphore_handle(&semTacheFFTClassificationStruct);
}

static void TacheFFTClassification_taskFxn(UArg a0, UArg a1)
{
        //Initialisations

        //Boucle infinie
        for (;;)
        {
                Semaphore_pend(semTacheFFTClassificationHandle, BIOS_WAIT_FOREVER);
        }
}
```

⟹ Dans le fichier header, ajoutez les prototypes des fonctions du fichier source.

⟹ Créez la tâche dans le fichier *main.c*, en pensant à inclure les fichiers Header nécessaires.

⟹ Rajoutez à présent la fonction *FFTClassificationTrigger* permettant de déclencher le sémaphore de la tâche *TacheFFTClassification* après avoir copié localement les tableaux nécessaires au calcul de la FFT. Le *define* et les tableaux devront être placé en début de fichier.

```
#define FFT_WINDOW_SIZE 256

float FFTSerieReal[FFT_WINDOW_SIZE];
float FFTSerieIm[FFT_WINDOW_SIZE];

void FFTClassificationTrigger(float serie[]){
        //Copie locale des buffer de data pour la FFT
        for (int i=0; i< FFT_WINDOW_SIZE; i++)
        {
                FFTSerieReal[i] = serie[i];
        }
        //Lancement de la tache FFT Classification
        Semaphore_post(semTacheFFTClassificationHandle);
}
```

⟹ Décommentez dans le code de la boucle principale de la *TacheADC* l'appel à la fonction *FFTClassificationTrigger*, et ajoutez un include vers son fichier header dans *TacheADC.c*.

⟹ Compilez et testez. Normalement, vous devriez déclencher le passage dans la boucle infinie de la tâche de classification toutes les 2.5 secondes environ. Vérifiez-le à l'aide d'un point d'arrêt.

⟹ Si tout est bon, ajoutez à présent au dossier *TacheFFTClassification* les fichiers sources et header *FFT* et *PeakDetector* téléchargeables ici : Lien de téléchargement.

⟹ Ajoutez après le *Semaphore_Pend* du fichier *TacheFFTClassification*, l'appel au calcul de la FFT à l'aide du code suivant, en veillant à inclure la librairie *math.h* :

```
//On utilise FFTSerieReal comme serie des parties reelles du signal temporel
//On utilise FFTSerieIm comme serie des parties imaginaires du signal temporel
//que l'on met a 0
for (int i=0; i<256; i++)
{
        FFTSerieIm[i] = 0;
}

//Calcul de la FFT
```

```
fft(FFTSerieReal, FFTSerieIm, 8, 1);

//Extraction de la norme de la FFT dans FFTDataY
for (int i=0; i<256; i++)
      FFTSerieReal[i] = sqrtf((FFTSerieReal[i]*FFTSerieReal[i]+FFTSerieIm[i]*FFTSerieIm[i]));
```

Vous allez à présent valider de manière opérationnelle le bon fonctionnement de l'algorithme de FFT proposé.

$\Longrightarrow$ Pour cela, branchez sur une alimentation de laboratoire le banc de mesure fourni par le professeur (Fig. 17), en veillant à ce qu'il fonctionne de manière équilibrée (sans ajout de balourd). Fixez votre *dev kit* dessus et faites tourner le banc à vitesse raisonnable (tension d'alimentation de $5V$ par exemple).

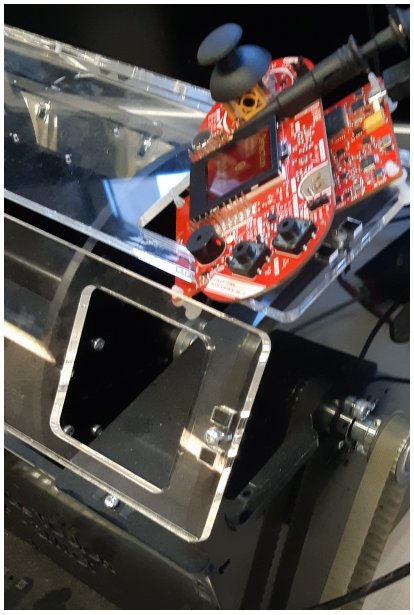

FIGURE 17 – Banc de Mesure avec dev kit

$\Longrightarrow$ Observez l'allure du signal analogique en sortie de l'accéléromètre à l'aide d'un oscilloscope. Pour cela vous devez regarder la 3^e (Accel X) pin en partant du dessus du connecteur $J3$ situé juste à gauche de l'écran LCD.

$\Longrightarrow$ Effectuez une analyse fréquentielle pour voir en temps réel les fréquences des pics obtenus sur l'oscilloscope.

$\Longrightarrow$ Dans le code embarqué, en plaçant un point d'arrêt en sortie du code de la FFT, vérifiez que vous obtenez des pics de FFT à des emplacements semblables à ceux de l'oscilloscope. On rappelle que la fenêtre étant de 256 échantillons et la fréquence d'échantillonnage de 100 Hz, l'incrément de fréquence dans le tableau de la FFT sera de $100/256 = 0.39$ Hz.

Vous allez à présent passer à l'extraction des pics à l'aide de la FFT de manière automatique.

$\Longrightarrow$ Pour cela, rajoutez après le calcul de la FFT le code suivant en n'oubliant pas d'ajouter les *include* nécessaire pour l'appel de la fonction *DetectPeak* et de la fonction *LCD_PrintState*. Pensez également à supprimer tout appel à la fonction *LCD_PrintState* à d'autres endroits du code :

```
//Determination des pics principaux
float OutPeakDetector[10];
DetectPeak(3, FFTDataY, 128, OutPeakDetector);
LCD_PrintState(0, 0, 0, 0, OutPeakDetector, 6);
```

$\Longrightarrow$ Analysez le code du *Peak Detector* afin de comprendre son fonctionnement. Vérifiez qu'il fonctionne bien sur le banc de mesure.

⟹ Afin de simplifier la lecture des fréquences des pics FFT, convertissez les index en fréquence avant leur affichage sur l'écran LCD.

3.6 Unsupervised learning

Après avoir récupéré les features, nous pouvons commencer à implémenter l'algorithme d'IA.

⟹ Pour cela importez dans le projet la structure de la tâche IA et les algorithmes de classification non supervisée à l'aide des fichiers téléchargeables ici : Lien de téléchargement.

⟹ Démarrez la tâche IA depuis le main, en pensant à inclure le fichier header correspondant.

Afin de pouvoir choisir dans quelle phase de l'algorithme vous êtes (apprentissage ou détection), vous allez ajouter une GPIO pour le bouton S2 du kit. Pour cela allez dans le fichier de configuration et ajoutez comme montre la figure 18 une entrée (INPUT) et connectée à *DIO16*. Au passage vous pouvez éteindre la LED verte du kit en déclarant une GPIO en sortie avec *Initial Output State* LOW et connectée à la pin *DIO20*.

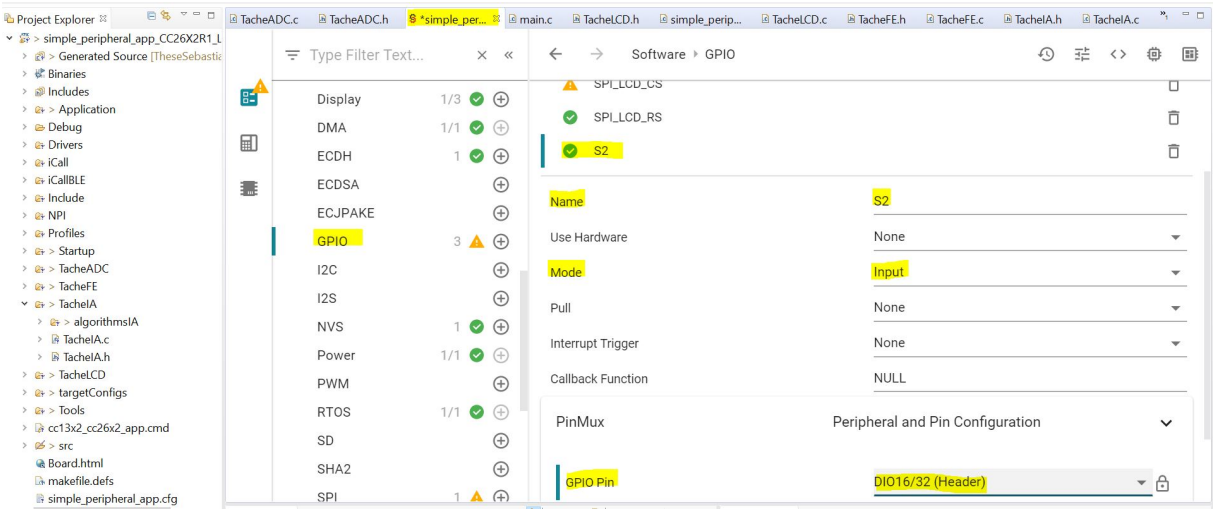

FIGURE 18 – Déclaration driver bouton S2

⟹ Après avoir déclaré le GPIO pour le bouton *S2*, ajoutez les lignes suivantes après le *semaphore_pend* de la tâche IA (Fig 19). Elles permettent de lancer l'entrainement ou la détection d'erreur selon le contexte (appui ou pas sur le bouton).

```
uint_fast8_t buttonS2 = GPIO_read(S2);

if(buttonS2 == 0){
        //Training
        Training(features);
        clustNumber = GetNumberOfClusters();
        int clusterDetected = GetLastClusterDetected();
        LCD_PrintState(1,clustNumber,clusterDetected,anomNumber,features,6);
        systemTrained = true;
}
else
{
        if(systemTrained)
        {
                //Anomaly Detection
                anomNumber = AnomalyDetection(features);
                clustNumber = GetNumberOfClusters();
                int clusterDetected = GetLastClusterDetected();
```

```
                LCD_PrintState(0,clustNumber,clusterDetected,anomNumber,features,6);
        }
}
```

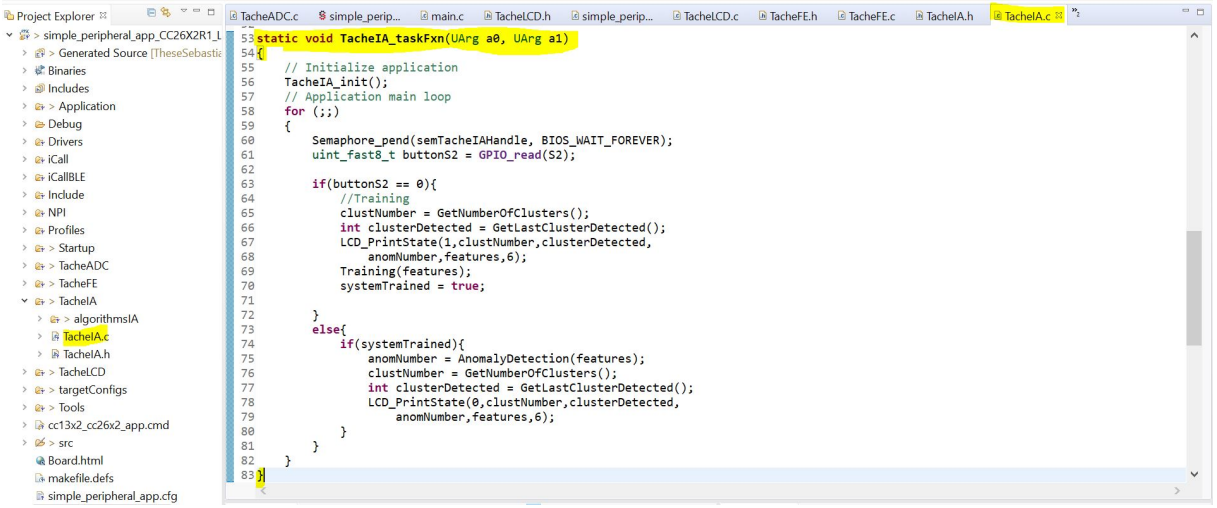

FIGURE 19 – Tâche IA

⟹ Vous devez à présent lancer l'appel à la fonction de classification ou détection d'anomalie depuis la tâche FFT, après la détection de pics dans la fonction *TacheFFTClassification_taskFxn*. Vous trouverez la fonction en question dans le header ou le source de *TacheIA*.

⟹ L'algorithme de classification non supervisée est à présent intégré sur le système embarqué.

⟹ Analysez le code de l'apprentissage non supervisé pour comprendre son fonctionnement.

⟹ Vous pouvez à présent le tester et également faire le tuning. Les paramètres à modifier sont les paramètres transmis quand on fait l'initialisation de l'algorithme d'IA (*DistanceMaximumConfigured* et *AlphaConfigured*). Vous pouvez les modifier dans le fichier *unsupervisedLearning.h*. Le paramètre *DistanceMaximumConfigured* configure le rayon des clusters, et modifie donc la sensibilité de l'algorithme.

⟹ Générer sur le banc de test des vibrations pour tester l'algorithme et vérifier quelle valeur de *DistanceMaximumConfigured* est idéal pour distinguer entre les modes de vibrations et détecter des anomalies.