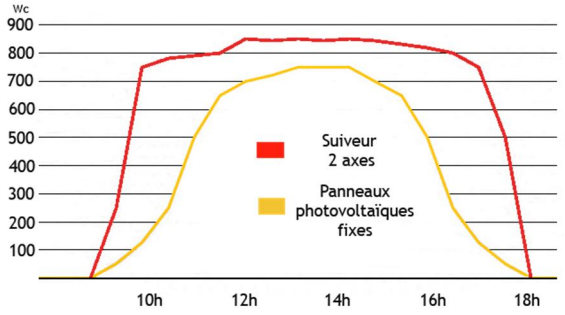
Mission de découverte du langage C – Le suiveur solaire

Les des nombreux défis du 21ème siècle concerne les ressources énergétiques. Parmi les énergies non fossiles, on trouve notamment les apports solaires, que l'on peut convertir en énergie électrique ou thermique avec des panneaux solaire (photovoltaïques ou thermiques).
Vous n'êtes pas sans savoir que la terre tourne autour du soleil ; ce qui provoque, depuis notre point de vue de terrien une rotation du soleil autour de la terre.

Il est trivial de constater qu'un panneau solaire parfaitement orienté face au soleil présentera un rendement supérieur à celui d'un panneau mal orienté (ou bien dont l'orientation n'est bonne qu'à un instant donné).

Pour maximiser le rendement d'un panneau solaire (et minimiser l'effort en énergie grise pour produire et acheminer le panneau) ; il est énergétiquement pertinent de chercher à orienter un panneau solaire en permanence face au soleil[1].



Comparaison de la production électrique d'un panneau solaire, avec et sans suiveur. Le gain est estimé autour de 40%.

L'utilisation de capteurs est une option (permettant notamment de rechercher à chaque instant la source lumineuse la plus forte)... mais c'est plus difficile par temps nuageux, où de nombreux rayons infrarouges traversent pourtant les nuages !
Nous proposons ici une méthode différente, basée sur un algorithme de sun tracking.

Cette mission s'appuie très fortement sur le document « C embarqué par l'exemple » ; et vous invite à le parcourir en accompagnant cette lecture d'applications concrètes dans le cadre du projet proposé. Tout au long de ce document, vous êtes invités à compléter le code source du sun tracker. L'ossature de ce code est fournie (nom des fonctions, fichiers, variables...).

Ce sujet sera abordé sous un angle simplifié, en tant que « maquette preuve de concept »... bien loin de toute considération d'industrialisation et de la rigueur scientifique qu'il faudrait exiger d'un tel sujet dans un contexte plus sérieux.

Laissez-vous guider, soyez attentifs et rigoureux dans la lecture des documents fournis.

[1] Les illustrations présentées sont des données issues d'une présentation commerciale.
https://www.kitsolaire-autoconsommation.fr/product/suiveur-photovoltaique-autoconsommation-deux-axes/255.html

Table des matières

Objectifs pédagogiques de la mission :

- Parcourir le document « le C embarqué par l'exemple »
- Prendre en main le langage C, par la pratique, et dans le cadre d'un exemple concret
- Découvrir l'utilisation de périphériques matériels
- Rédiger un module logiciel

1. Introduction

Au fur et à mesure de cette mission, il est demandé de lire les chapitres du document « le C embarqué, par l'exemple ». Ces instructions sont indiquées en bleues.

Chaque étape à réaliser est indiquée par un 'point' : •

Des instructions permettant de tester et valider vos codes sont proposées. Ces éléments sont indiqués en vert.

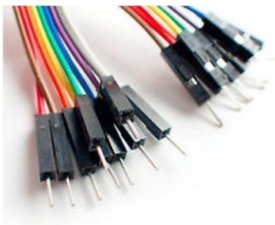
Tout au long des exercices proposés, il est fait référence à des fichiers et fonctions que vous devez localiser dans le code. Une maitrise de **l'outil de recherche** est indispensable :

- CTRL+H
- Onglet 'file search' (pour rechercher dans l'ensemble des fichiers)
- Indiquez le texte recherché
- Précisez éventuellement le lieu de la recherche (tout le workspace / projet courant seulement...)
- (l'onglet C/C++ peut être utilisé en première approche, il restreint la recherche aux mots clés définis d'une façon correcte en C, et ne cherche pas dans les commentaires par exemple, ou en dehors des fichiers sources. Il ne trouve pas non plus des 'morceaux de mots')

Nous vous conseillons également un usage intensif du « CTRL+clic » qui permet de **naviguer dans le code.** Essayer c'est l'adopter.

Matériel nécessaire pour aborder cette mission :

- Une carte Nucleo F103
- Un câble USB mini-B
- Un servomoteur
- Un potentiomètre
- 6 fils duponts mâle-mâle



Dans le laboratoire :

- Un ordinateur sur lequel sont installés :
  - System Workbench for STM32
  - Docklight ou terminal série équivalent
- Un oscilloscope et une sonde

Sur les tabourets :

- Binôme d'étudiants motivés et méthodiques.

2. Prise en main du procédé et découverte du langage C

- Commencez par lire l'introduction et les chapitres 1, 2 et 3. – fonctions, variables, paramètres, valeurs de retours, main()

- Ouvrez *OpenWorkbench for STM32*, dans un workspace qui peut être le même que pour la précédente mission
    - (rappel : pas d'accent ni d'espace dans le chemin. Par exemple Z:\workspace_pse)
    - (rappel : on peut avoir une multitude de projets dans un unique workspace.)

- Vous pouvez maintenant importer le projet **sun_tracker**, disponible sur le campus, et localiser sa fonction main() du fichier main.c
    - Si besoin, consultez la mission précédente pour savoir comment importer un projet !

Vous y trouvez notamment :
- Quelques initialisations diverses
- L'appel de la fonction `SUN_TRACKER_init();`
- L'appel de la fonction `SUN_TRACKER_process_main();`

- Parcourez ces fonctions
    - Localisez notamment l'appel à `HAL_GPIO_TogglePin(LED_GREEN_GPIO, LED_GREEN_PIN);`

- Compilez le programme en l'état, et envoyez-le sur la cible
    - (Consultez ci besoin les explications de la mission précédente)
    - Vous devez observer un changement d'état de la led verte à chaque seconde.

- Lisez le chapitre 4 – Types
- Lisez le chapitre 5 – Transtypage
- Lisez le chapitre 6 – Portion conditionnelle de code
- Lisez le chapitre 7 – Opérateur ternaire

- Corrigez le contenu de la fonction `static void process_ms(void)` en remplaçant la variable déclarée en `int` par un type 'moralement acceptable'.
    - Il est utile de parcourir cette fonction pour savoir quelle peut être la valeur maximale de cette variable.
    - (Le mot clé `static` sera abordé ultérieurement.)

- Lisez le chapitre 8 – Commentaires

Prenez l'habitude de nourrir vos codes de commentaires riches et pertinents.

Indépendamment des commentaires textuels qui accompagnent les lignes de codes, on utilise couramment la syntaxe des commentaires pour « désactiver » du code que l'on ne veut pas supprimer pour autant. Cela est plutôt destiné à un usage temporaire, mais c'est très pratique.

Astuce : pour commenter/décommenter un bloc de code, il y a le raccourci clavier **CTRL+SHIFT+/**
- Essayez cette astuce sur une portion de code quelconque. (le '/' n'est pas celui du pavé numérique)

- Lisez le chapitre 9 – Débogueur

- Appelez la fonction **`useless_function()`** et exécutez là en pas à pas à l'aide du débogueur.
- Observez ligne après ligne l'évolution des variables :
    - Deux méthodes sont disponibles :
        - En plaçant le curseur de la souris sur la variable
        - En consultant l'onglet 'variables'
    - (Certaines variables ne sont accessibles qu'en les nommant dans l'onglet 'expressions')

    - La variable **`c`** doit être observée au format binaire.
    - Les variables **`d`** et **`e`** doivent être observées au format hexadécimal.
    - Ne vous préoccupez pas du mot clé '**`volatile`**'.

- Placez un breakpoint à la ligne **`n++`** et appuyez sur le bouton poussoir pendant que le code tourne.
- Sans modifier le programme, et sans appuyer sur le bouton, trouvez un moyen de rentrer dans le **`if(b)`**
    - Un indice : le débogueur fournit plus qu'un accès en lecture aux variables !

- Lisez le chapitre 10 – Opérateurs arithmétiques et logiques
- Exécutez la fonction **`useless_function()`** en pas à pas avec le débogueur pour comprendre le rôle des opérateurs logiques bit à bit.

  Complétez ce tableau :

	Exercice pratique : Est-ce Vrai ou Faux ?
`int8_t a;`	on ne sait pas encore !
`int8_t b;`	on ne sait pas encore !
`int8_t c;`	on ne sait pas encore !
`a = 42;`	a est _____
`b = 0;`	b est _____
`c = !a;`	c vaut ___ → c est donc_____
`c = !b;`	c vaut ___ → c est donc_____
`b = 1`	b est vrai
`a && b`	vrai ET vrai donne _____
`a || b`	vrai OU vrai donne _____
`a || 0`	vrai OU faux donne _____
`a || 1`	vrai OU vrai donne _____
`a && 0`	vrai ET faux donne _____
`a && 1`	vrai ET vrai donne _____

    - Assurez-vous d'avoir bien compris le rôle de chaque opérateur. Notez d'éventuelles questions en cas de doutes.

- Désactivez l'appel de la fonction **`useless_function().`** Nous en aurons plus besoin.

3. Tableaux, chaines de caractères, communication série, ...

- Lisez le chapitre 11 – Définition de tableaux
- Lisez le chapitre 12 – Boucles for

Patience, nous allons bientôt pouvoir créer un tableau et le parcourir avec une boucle for.

- Lisez le chapitre 13 – Boucles while

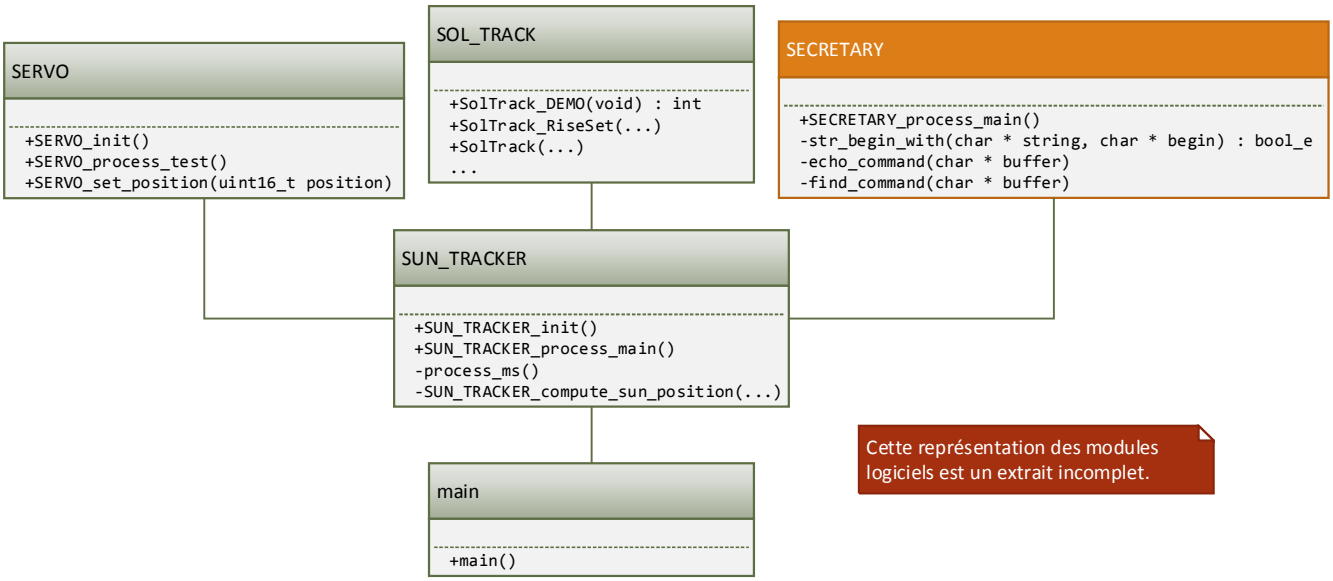
A l'allumage de notre dispositif, on souhaite afficher une information de sécurité à destination de l'utilisateur, et lui imposer l'appui sur un bouton pour démarrer le système.
Nous ne disposons pas encore de la fonction permettant d'afficher ce message, mais nous pouvons dors-et-déjà assurer le blocage en attente de l'appui bouton.

- Dans la fonction main, avant la boucle while(1) (que l'on nomme généralement 'boucle de tâche de fond'), insérez une boucle while qui bloque le programme tant que l'utilisateur n'a pas pressé le bouton bleu.
    - Vous devez utiliser la fonction **`readButton()`**

Vérifiez que la led verte ne clignote pas tant qu'on a pas appuyé sur le bouton bleu.

- Lorsque cette fonctionnalité est validée, mettez cette fonctionnalité en commentaire... car c'est vraiment trop pénible de devoir appuyer sur le bouton à chaque lancement du programme.

- Observez cette représentation partielle des modules logiciels. On s'intéresse au module **SECRETARY**.
    - On remarque qu'il possède une seule fonction publique, et 3 fonctions privées.



Lorsque l'on allume le tracker solaire, il ne peut pas connaitre le jour et la date courants.
Afin de lui apporter ces informations, on souhaite utiliser une liaison série ; c'est-à-dire 3 fils qui nous permettront de communiquer des octets envoyés bit par bit : **RX**, **TX** et la **masse**.

Le périphérique interne qui gère cette liaison série se nomme UART. (ou USART : Universal Synchronous/Asynchronous Receiver Transmitter)

Le STM32F103 dispose de plusieurs périphériques UART. Dans notre exemple, on utilise l'UART2.
Ce périphérique est déjà initialisé (voir au début de la fonction `main()`)

Le module logiciel fourni qui exploite ce périphérique est capable de réagir à chaque caractère reçu et de les stocker dans un tableau (par un mécanisme d'interruption que nous verrons plus tard).
Notre code peut alors :

- consulter l'état de ce tableau en utilisant la fonction : `UART_data_ready(UART2_ID).`
- récupérer le prochain caractère reçu avec : `c = UART_getc(UART2_ID)`

- Relisez le chapitre 8 – Commentaires
- Ajoutez ces commentaires en préambule de la fonction **`void SECRETARY_process_main(void)`**

```
/**
* @brief        Récupère les octets reçus sur l' UART2, les rassemble dans un buffer et les traite si on reçoit '\n'
* @pre          cette fonction doit être appelée régulièrement dans la tâche de fond
* @pre          les trames reçues doivent se terminer par un '\n' et ne pas dépasser BUFFER_SIZE
* @post         la taille du buffer est limitée. Si de trop nombreux caractères sont reçus, ils s'écrasent à la fin.
* @post         le traitement des trames reçues est sous-traité
*/
```

- Complétez la fonction **`void SECRETARY_process_main(void)`** selon ce pseudo code :

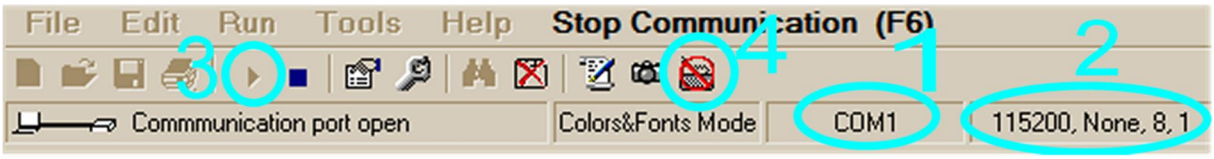
```
void SECRETARY_process_main(void)
{
      Définition d'un tableau de BUFFER_SIZE caractères, de type 'static char'
      Définition d'un entier de type 'static uint8_t', nommé index (initialisé à 0)
      Définition d'une variable de type char, nommée c

      Tant que (Des données sont disponibles sur l'UART2, cf fonction UART_data_ready)
      {
            c = prochaine donnée disponible sur l'UART2, cf fonction UART_getc

            ranger c dans la case index du tableau buffer

            si (c vaut '\n')
            {
                  //(on considère que c'est le dernier caractère de la chaine)
                  Ecraser la case index du tableau buffer avec un zéro : '\0'
                  Appeler la fonction echo_command() en lui fournissant buffer
                  Remettre la variable index à 0
            }
            Sinon si (index est inférieur à BUFFER_SIZE - 1)
            {
                  Incrémenter la variable index pour le prochain caractère
            }
      }
}
```

Pour tester cette fonction, on a pré rempli la fonction echo_command() avec un code temporaire qui se contente de renvoyer la trame envoyée lorsqu'elle est reçue.

L'UART2 a été choisi car c'est celui qui est physiquement relié au microcontrôleur de la sonde de débogage. Chaque octet envoyé est rendu disponible par la sonde de débogage sur un port série virtuel que votre ordinateur peut ouvrir. (C'est ici la liaison USB qui véhicule – entre autres – ces données.)

A l'aide d'un logiciel hyperterminal série (par exemple docklight) ; ouvrez le port série COM**x** qui correspond à la sonde de débogage de la carte Nucleo. (le nombre **x** dépend de votre PC).

Utilisation de Docklight.



1- Ouvrez le bon port COM (Sur les PCs de labos, ce n'est pas le port COM1 !)
2- Liaison paramétrée à 115200 bps / 8N1
3- Ouvrez la communication en appuyant sur Start (ou F5)
4- Activez le clavier pour pouvoir envoyer des commandes dans le terminal

- On vous propose ensuite le jeu de test suivant :

Chaine à envoyer (en terminant par le caractère '\n' que l'on peut saisir en appuyant sur entrée.)	**Comportement attendu**
`Test`	Renvoie la chaine 'Test'
`Une chaine`	Renvoie la chaine 'Une chaine
`Cette chaine compte plus de 128 octets et doit logiquement être tronquée ! C'est super bien de travailler sur microcontrôleur pour avoir de bonnes notes.`	Renvoie les 128 premiers octets de la chaine

Les chaines renvoyées doivent l'être en 'un seul bloc', à chaque appui sur entrée.

- Lisez le chapitre 14 – Boucles do...while()

Soyons clairs, c'est pas tous les jours qu'on peut sortir un beau do...while().
On se contentera à ce stade de savoir que ça existe.

- Lisez le chapitre 15 – Boucles imbriquées
- Lisez le chapitre 16 – Chaines de caractères, code ASCII
- Profitez du chêne de caractère pour prendre une grande respiration et attaquer la suite.

Il faut maintenant analyser les trames récupérées sur la liaison série afin de comprendre correctement les commandes transmises par l'utilisateur.
Pour cela, la fonction `find_command()` compare le début de la trame reçue avec une liste de commandes.
Cette fonction ne rentre pas dans le cadre de cette étude, mais vous pouvez y jeter un œil... !

- Remplacez l'appel de la fonction `echo_command(buffer)` par un appel à `find_command(buffer)`

`find_command()` sous-traite chaque comparaison à la fonction suivante que vous devez compléter :

```
static bool_e str_begin_with(char * string, char * begin);
```

- Complétez selon l'algorithme fourni :

```
static bool_e str_begin_with(char * string, char * begin)
{
	Définition d'une variable booléenne de type bool_e : ret
	Définition d'une variable de type uint16_t : i

	//on fait l'hypothèse naïve que string commence par le contenu de begin
	Initialiser la variable ret à TRUE;

	Pour(i allant de 0 ; tant que begin[i] est différent de '\0' ; incrémenter i)
	{
		//si c'est déjà la fin inattendue de la chaine string ou que les chaines
		présentent une différence
		Si(string[i] vaut 0 ou que string[i] est différent de begin[i])
		{
			//l'hypothèse était fausse, les chaines diffèrent !
			Affecter FALSE à la variable ret
			Sortir de la boucle (avec un break;) (car continuer est inutile)
		}
	}
	Retourner la variable ret
}
```

- Validez le bon fonctionnement en utilisant les commandes suivantes :

Commandes (en terminant par 'entrée')	Comportement attendu
`help`	Affiche la liste des commandes
`getdate`	Affiche la date
`gettime`	Affiche l'heure
`setdate 25/12/19`	Modifie la date
`settime 17:30:00`	Modifie l'heure
`settime 7:30:00`	Refus de modification de l'heure (0 manquant)
`setdate 31/04/21`	Refus de modification de la date

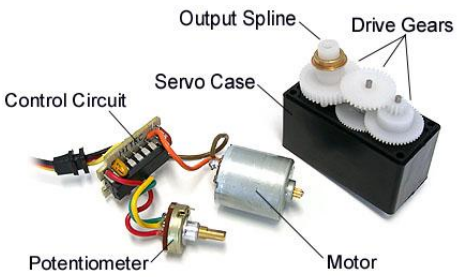
Attention, les commandes sont à indiquer en minuscules, et sans oublier les zéros (08:35:00...)

Il serait bien sûr possible de rendre le code plus tolérant, mais ce n'est pas demandé.

4. Ecriture d'un logiciel suivant la bonne pratique des modules

- Lisez les chapitres 20 – Module Logiciel et 21 - #include "fichier.h"

On doit maintenant ajouter au suiveur solaire un module logiciel gérant un servomoteur.
Un servomoteur est un ensemble contenant : un moteur + un réducteur + un capteur de position + l'électronique de pilotage.

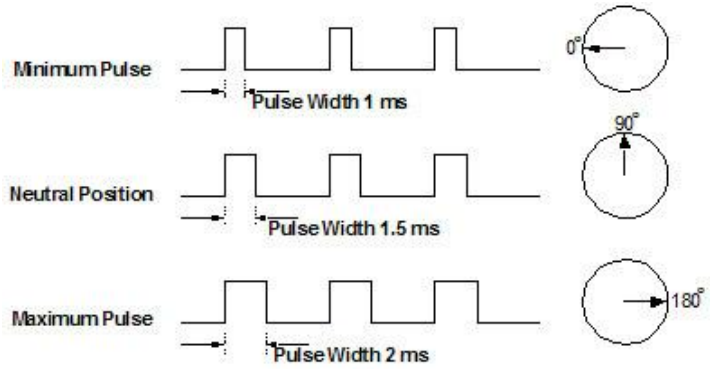


Images : https://www.servocity.com/how-does-a-servo-work

Bien qu'on puisse considérer que cet ensemble dispose d'une « forme d'intelligence » il ne s'agit pas d'un « cerveau-moteur[2] ». L'étymologie du mot 'servo' vient de 'service'. Ce moteur est 'asservit', il rend 'service' et **répond à une consigne.**

Cette consigne[3] doit lui être présentée sous forme d'une impulsion dont la durée informe le servomoteur de la position souhaitée. Comme il a la mémoire courte, il faut lui répéter régulièrement cette impulsion (typiquement toutes les 10 à 20ms).

Durée de l'impulsion	Position demandée
1ms[4]	à fond à gauche !
...	...entre les deux...
1,5ms	...au milieu...
...	...entre les deux...
2ms[4]	à fond à droite !



Une méthode naïve pourrait consister à fabriquer ce signal de la façon suivante :

```
while(1){
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 1);
        HAL_Delay(1);
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 0);
        HAL_Delay(10);
}
```

Cette méthode présente un inconvénient majeur : on ne peut faire que ça ! (ou presque)

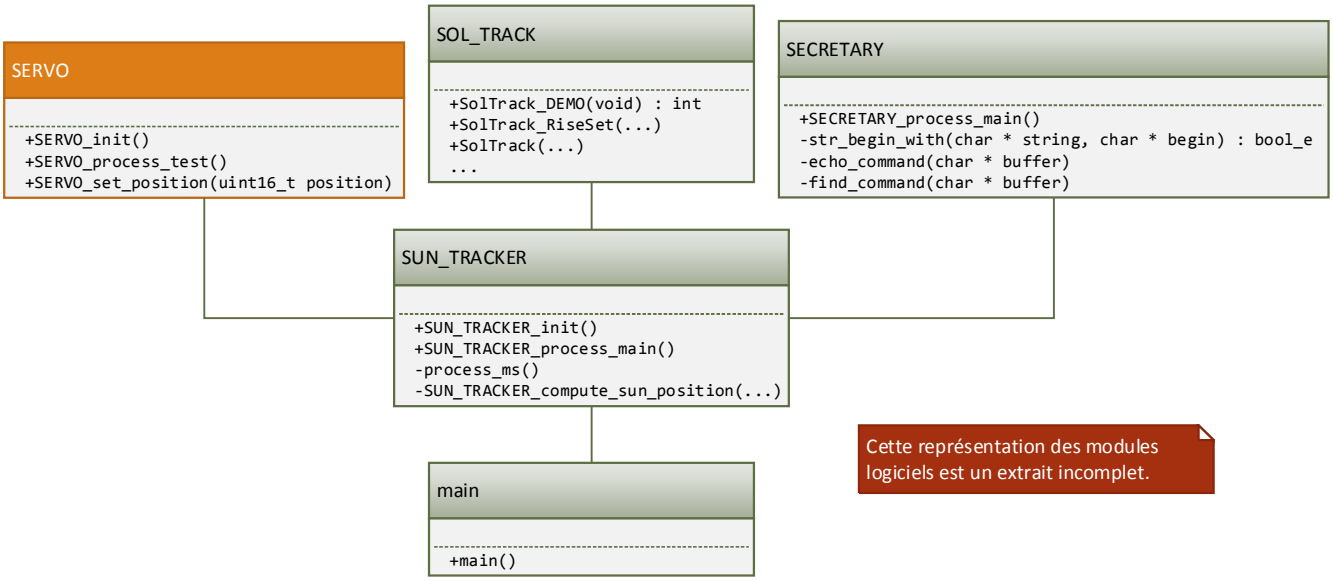
En effet, on occupe ici 100% de notre processeur. Si on veut demander à notre microcontrôleur de réaliser d'autres tâches, cela aura un effet substantiel sur la consigne envoyée au servomoteur. (dysfonctionnements ou vibrations garantis !)

Il convient donc de sous-traiter la fabrication du signal à un **périphérique timer,** en particulier en générant un **signal PWM.**

- Créez le module logiciel servo en suivant les instructions ci-après. Les fonctions de pilotage du timer1 sont disponibles dans le module suivant : **lib/bsp/stm32f1_timer.c/h**
- Pour les utiliser, il faut mettre à 1 la macro USE_BSP_TIMER dans config.h

[2] ça, c'est vous, là, maintenant !
[3] Il existe de nombreux types de servomoteurs, on s'intéresse ici aux petits servomoteurs de modélisme.
[4] Selon le modèle exact de servomoteur, les butées peuvent être 'un peu au-delà ces valeurs standards'.

On s'intéresse maintenant au module SERVO :



Créer les fichiers du module logiciel servo : servo.c et servo.h. Et remplissez-les.

- Fonctions publiques :
  - **void SERVO_init(void);**
  - **void SERVO_set_position(uint16_t position);**
  - **void SERVO_process_test(void);**
- Fonctions privées :
  - (aucune)
- Dépendances :
  - config.h, stm32f1_timer.h, stm32f1_gpio.h, macro_types.h
- Constante privée (privée car elle se trouve dans le .c et non dans le .h !)
  - `#define PERIOD_TIMER 10 //ms`

Contenu des fonctions :

Nous vous fournissons le contenu de la fonction de test, que vous devez temporairement appeler (lors de la phase de test) dans la boucle de tâche de fond du main (le fameux 'while(1)').

```
void SERVO_process_test(void)
{
	static uint16_t position = 50;
	static bool_e previous_button = FALSE;
	bool_e current_button;

	//lecture du bouton bleu
	current_button = !HAL_GPIO_ReadPin(BLUE_BUTTON_GPIO, BLUE_BUTTON_PIN);
	if(current_button && !previous_button)		//si appui bouton
	{
		position = (position > 99)?0:(position+5);	//de 0 à 100%, par pas de 5%
		SERVO_set_position(position);
	}
	previous_button = current_button;	//sauvegarde pour le prochain passage
	HAL_Delay(10);	//anti-rebond "de fortune" en cadencant la lecture du bouton
}
```

La fonction d'initialisation doit être appelée avant la boucle de tâche de fond du main.

```
void SERVO_init(void)
{
	//initialisation et lancement du timer1 à une période de 10 ms
	TIMER_run_us(TIMER1_ID, PERIOD_TIMER*1000, FALSE);  //10000us = 10ms

	//activation du signal PWM sur le canal 1 du timer 1 (broche PA8)
	TIMER_enable_PWM(TIMER1_ID, TIM_CHANNEL_1, 150);

	//rapport cyclique reglé pour une position servo de 50%
	SERVO_set_position(50);
}
```

```
//position est exprimée de 0 à 100.
void SERVO_set_position(uint16_t position)
{
	if(position > 100)
		position = 100;	//écretage si l'utilisateur demande plus de 100%

	//TODO : mise à jour du rapport cyclique.
	//duty doit être exprimé ici de 100 à 200 (sur 1000) (pour un rapport cyclique
de 10% à 20%, c'est-à-dire une durée de pulse de 1 à 2ms dans la période de 10ms)
	//Donc on additionne 100 à position.
}
```

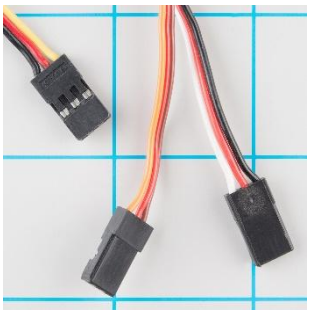
- Complétez la fonction `SERVO_set_position()` afin qu'elle sous-traite au module timer la mise à jour du rapport cyclique souhaité.
  - (Trouvez la bonne fonction par vous-même...elle se trouve dans stm32f1_timer.c)
  - Ce rapport cyclique est « position + 100 » ; de telle sorte que pour une position variant de 0% à 100%, on ait un rapport cyclique variant de 100 'pour 1000' à 200 'pour 1000'.

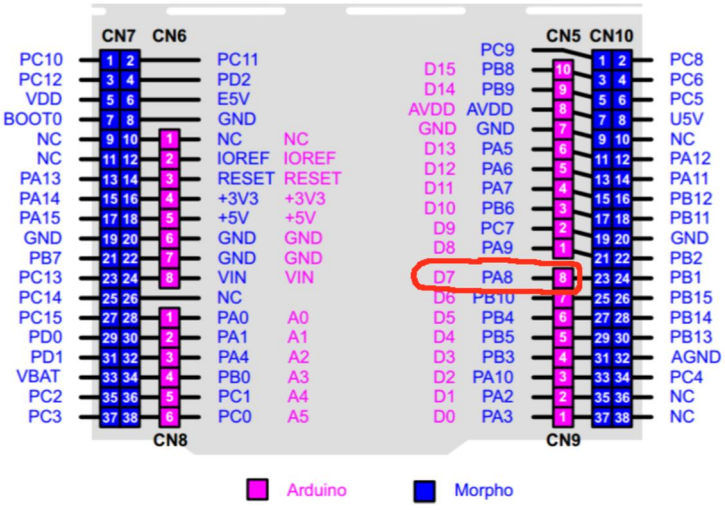
Tests du module logiciel servo.c/h :

1- Observez à l'oscilloscope la broche PA8 (l'utilisation d'un petit fil peut aider).
On doit y voir un pulse de période 10ms et de durée 1 à 2ms.
La durée de ce pulse doit évoluer à chaque appui sur le bouton bleu.

2- Lorsque ce pulse est correctement observé, on peut brancher le servomoteur :

GND	5V	Signal PA8 (Timer1 – CH1)
Noir	Rouge	Jaune
Marron	Rouge	Orange
Noir	Rouge	Blanc

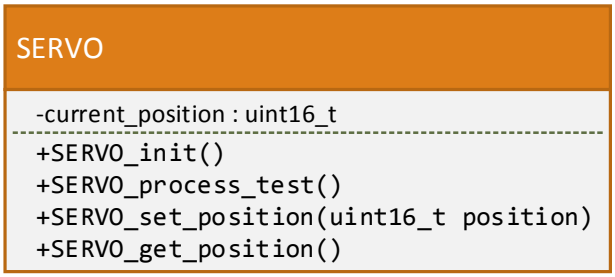




3- Vérifiez les mouvements du servomoteurs lors des appuis sur le bouton bleu.

- Lisez le chapitre 22 – Public/Privé
- Lisez le chapitre 23 – static
- Lisez le chapitre 24 – Accesseurs et variables privées

- Dans le module servo.c/h, on souhaite ajouter une fonction permettant de récupérer la position demandée :



  - Ajoutez une variable privée au début du fichier servo.c :
    ```
    static uint16_t current_position;
    ```
  - Ajoutez l'affectation de cette variable dans la fonction **SERVO_set_position()** :
    ```
    current_position = position;
    ```
  - Ajoutez la fonction d'accès en lecture à cette variable (pensez à recopier le prototype de cette fonction dans servo.h et à définir le contenu de la fonction dans servo.c)
    ```
    uint16_t SERVO_get_position(void)
    ```

- Dans le module sun_tracker.c/h, on souhaite ajouter une fonction permettant de modifier la localisation géographique.
  - Ajouter la fonction suivante :
    ```
    void SUN_tracker_set_new_pos(double longitude, double latitude)
    ```
  - Cette fonction doit mettre à jour les variables `loc.longitude` et `loc.latitude`.

Vous pouvez tester le bon fonctionnement de cette fonction avec la destination vacances de votre choix, et en utilisant le mode debug pour vérifier que la structure loc a bien été mise à jour :
Il vous suffit pour cela d'appeler la fonction où vous le souhaitez, un couple de paramètres.

Quelques exemples :

Destination vacances	**Paramètres**
Tahiti	`SUN_tracker_set_new_pos(-17.619774, -149.484625);`
Base antarctique Dumont d'Urville	`SUN_tracker_set_new_pos(-66.662970, 140.002640);`
New York	`SUN_tracker_set_new_pos(40.689219, -74.044613);`
Very Large Telescope (Chili)	`SUN_tracker_set_new_pos(-24.627481, -70.404476);`
ESEO Velizy	`SUN_tracker_set_new_pos(48.783990, 2.210231);`

Ces coordonnées pourront également servir plus tard pour comparer les suivis solaires dans différents lieux à la même heure.

- Lisez le chapitre 18 – Structures

Dans la suite de la mission, nous allons utiliser plusieurs structures.
Souvenez-vous de ceci :

- Accès au champ d'une structure qui nous appartient : s.champ
- Accès au champ d'une structure dont on ne connait que l'adresse : s->champ

Eclipse est toutefois doté d'un automatisme intéressant qui remplace automatiquement les '.' en '->' lorsque c'est nécessaire.

5. Quelques notions clés du fonctionnement du langage

- Lisez le chapitre 25 – #define
- Lisez le chapitre 26 – #ifdef et #if

Les algorithmes fournis par le module logiciel SolTrack sont capables de nous indiquer la position du soleil à partir de notre position géographique, de l'heure et de la date.
On souhaite afficher les données calculées par ces algorithmes.
Cependant, on veut pouvoir activer/désactiver cet affichage à l'aide d'une macro `ENABLE_DISPLAY`

- Définissez la macro `ENABLE_DISPLAY` à **1** au début du fichier sun_tracker.c

A la fin de la fonction SUN_TRACKER_compute_sun_position, ajoutez une portion conditionelle :

```
#if ENABLE_DISPLAY
      //nous mettrons ici les affichages
#endif
```

Observez ce qui se passe lorsque vous basculez à 0 la macro `ENABLE_DISPLAY`. La portion de code concernée doit se 'griser'. Dans ce cas, elle n'est plus prise en compte à la compilation.

- Lisez le chapitre 27 - printf
- Lisez le chapitre 28 - float

- Dans la portion conditionnelle `ENABLE_DISPLAY` définie précédemment, utilisez la fonction printf pour chacun de ces affichages (vous pouvez adapter le contenu des chaines affichées) :

Exemple d'affichage :

```
Date          :    1  4 2020
Heure         :    8:30:40.4
Jour julien   :    2458940.85

Heure levé    :   5.6722,     azimuth  :  82.08
Heure zénith  :   12.0988,    altitude :  47.34
Heure couché  :   18.5407,    azimuth  :  278.23

Ecliptique longitude :                12.07°
ascension droite et déclinaison :     11.10°       4.77°
altitude non corrigée :               27.31°

azimuth et altitude corrigées :       115.13°      27.34°
```

A vous d'indiquer correctement les chaines de format pour les variables indiquées

Données	**Variables concernées**
Date	time->year, time->month, time->day
Heure	(int)time->hour, (int)time->minute, time->second
Jour Julien	pos->julianDay
Heure levé, azimuth	riseSet->riseTime, riseSet->riseAzimuth
Heure zenith, altitude	riseSet->transitTime, riseSet->transitAltitude
Heure couché, azimuth	riseSet->setTime, riseSet->setAzimuth
Longitude de l'écliptique	pos->longitude
ascension droite et déclinaison	pos->rightAscension, pos->declination
altitude non corrigée	pos->altitude
azimuth et altitude corrigées	pos->azimuthRefract, pos->altitudeRefract

- Lisez le chapitre 29 – pointeurs

Il est temps de faire varier le temps. (Au temps pour moi, vous m'en direz tant...)

Le périphérique RTC (Real Time Clock, en français horloge temps réel) permet de compter le déroulement du temps en utilisant les périodes bien connues : secondes, minutes, heures, jours, mois, année. (La RTC connaît notamment le nombre de jour par mois selon l'année ! ...)

On peut l'utiliser avec une source d'horloge interne (peu précise), ou bien reliée à un quartz externe de 32768 Hz. On atteint alors une précision de l'ordre de 10 à 30 ppm (1 ppm = 1 partie par million), soit une déviation annuelle de l'ordre de 5mn.

- Dans la fonction **void SUN_TRACKER_init(void) :**

```
RTC_init(TRUE); //laissez bien cette ligne en place

//Choisissez l'horaire de départ arbitraire au démarrage du programme...
RTC_DateTypeDef currentDate;
RTC_TimeTypeDef currentTime;
currentDate.Date = 1;
currentDate.Month = 4;
currentDate.Year = 20;
currentTime.Hours = 8;          //attention à renseigner l'heure GMT
currentTime.Minutes = 0;
currentTime.Seconds = 0;
RTC_set_time_and_date(&currentTime, &currentDate);
```

On remarque que la fonction `RTC_set_time_and_date()` attend en paramètres deux pointeurs vers des structures. On fournit donc bien ici les adresses de ces structures que nous avons définies.

- Dans la fonction **void SUN_TRACKER_process_main(void)**, remplacez les affectations temporaires de la structure time par :

```
RTC_DateTypeDef currentDate;
RTC_TimeTypeDef currentTime;
RTC_get_date(&currentDate);     //on récupère la date courante
RTC_get_time(&currentTime);     //on récupère l'heure courante (GMT !)

//recopie des données de la RTC au format attendu par la librairie SolTrack
time.year =         (int32_t)(currentDate.Year) + 2000;
time.month =        (int32_t)(currentDate.Month);
time.day =          (int32_t)(currentDate.Date);
time.hour =         (int32_t)(currentTime.Hours);
time.minute =       (int32_t)(currentTime.Minutes);
time.second =       (double)(currentTime.Seconds);
```

On remarque que les structures `currentDate` et `currentTime` seront remplies par les fonctions `RTC_get_date()` et `RTC_get_time()`.

On remarque également les transtypages pour faire correspondre les types des données manipulées.

- Lisez le chapitre 30 – pointeur sur fonction

- Observez l'appel `Systick_add_callback_function(&process_ms);` qui est fait dans la fonction **`SUN_TRACKER_init()`**.

On fournit ici l'adresse de la fonction process_ms afin qu'elle soit sauvegardée dans un tableau de pointeurs sur fonction (qui sera parcouru ensuite chaque milliseconde).

- Lisez le chapitre 31 - volatile

Sans nécessairement prendre le temps de tester cela, sachez que le mot clé 'volatile' utilisé dans la définition du `flag_1s` est primordial.

En effet, si l'on demande au compilateur de produire un code optimisé (-O2, -O3), ce dernier pourrait se contenter d'une copie locale de cette variable, et ne pas se rendre compte qu'elle est modifiée par une routine d'interruption !

Toute variable dont l'utilisation est partagée entre des tâches de niveau de préemption différent mérite ce mot clé.

Il en est de même pour des registres de périphériques dont la valeur peut changer à chaque instant à cause du périphérique concerné.

Quel temps met le code à s'exécuter ?

Il est assez difficile de calculer précisément la durée d'exécution d'une portion de code.
Cela dépend notamment des options de compilation (optimisation...), des variables, des interruptions éventuelles, ...
Bien souvent, la meilleure méthode consiste à effectuer une mesure :

On s'intéresse ici à la durée nécessaire pour effectuer l'ensemble des calculs déterminant la position du soleil.

- Pour mesurer cette durée, c'est très simple :
  - Commentez la ligne `HAL_GPIO_TogglePin(LED_GREEN_GPIO, LED_GREEN_PIN);`
  - Ajoutez un allumage de la led verte avant l'appel à `SUN_TRACKER_compute_sun_position`
  - Ajoutez une extinction de la led verte après cette ligne
  - Observez et mesurez la durée d'allumage de la led à l'oscilloscope

6. Suivi du soleil

- Afin de faire fonctionner le suivi solaire selon l'azimuth, ajoutez à votre code le pilotage du servomoteur selon le paramètre `pos.azimuthRefract`.
    - Côté nord (<90 ° ou >270°) : servomoteur en position 0
    - Plein Est : (90°) : servomoteur en position 0
    - Plein Ouest (270°) : servomoteur en position 100
    - Côté sud (>90° et <270°) : servomoteur entre 0 et 100 ; proportionnellement

    - Utilisez pour cela une variable locale `uint16_t position`

- Pilotez le servomoteur avec `SERVO_set_position(position);` en vous assurant qu'aucun autre appel à cette fonction n'entre en conflit. (si nécessaire, il est tant de désactiver tout appel à `SERVO_process_test()`).

- Afin de tester plus facilement, il est utile « d'accélérer la RTC ». Ajoutez ces deux lignes dans la fonction `void SUN_TRACKER_init(void)` (après l'initialisation de la RTC !) :
    - `#define TIME_ACCELERATION (60*5) //5mn par seconde`
    - `RTC_set_time_acceleration(TIME_ACCELERATION);`

Testez en observant les mouvements réguliers du servomoteur selon l'heure.

Vous pouvez également faire varier l'horaire et la date avec les commandes settime et setdate envoyées via la liaison série.

Vous pouvez également faire varier le lieu.

7. Ajout d'un mode manuel – utilisation du périphérique ADC

On souhaite, à des fins de maintenance, ajouter un mode de pilotage manuel du servomoteur.
Pour cela, nous devons implémenter deux fonctionnalités :

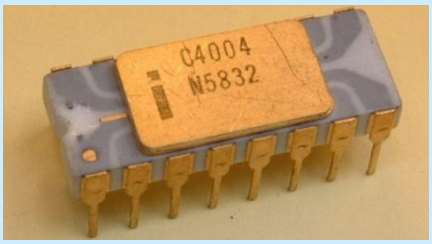
- ➔ Lecture de la tension analogique fournie par un potentiomètre pour piloter le servomoteur
- ➔ Passage du mode manuel au mode auto lors de chaque appui sur le bouton bleu

---

Afin de piloter le mode manuel à partir d'un potentiomètre, mettons en œuvre le périphérique ADC.
Le périphérique ADC (Analog to Digital Converter) permet d'acquérir une tension analogique et de la convertir en un nombre, exploitable dans le programme. En théorie, il est impossible de refléter dans une quantification **numérique** (à résolution limitée) la grandeur analogique **exacte**. Exemple : si vous mesurez la longueur d'un crayon avec un double décimètre et que vous obtenez 14,7cm ; c'est faux. Peut-être fait-t-il 14,7000000cm. Ce qui est encore faux... Il faudrait sans doute compter jusqu'aux atomes pour obtenir une juste longueur.
Expliquons comment la mesure s'effectue.

Vous connaissez le principe du juste prix ?
Quel était le prix de ce magnifique microprocesseur i4004, en 1971, premier né d'une longue série. (740kHz, 2300 transistors, 92 000 opérations par secondes !)

$512 ?	C'est moins.	→ 0
$256 ?	C'est moins.	→ 0
$128 ?	C'est plus.	→ 1
$192 ?	C'est plus	→ 1
$224 ?	C'est moins.	→ 0
$208 ?	C'est moins.	→ 0
$200 ?	C'est plus.	→ 1
$204 ?	C'est moins.	→ 0
$202 ?	C'est plus.	→ 1
$203 ?	Gagné !	→ 1



(on remarque que 0b0011001011 = 203)
(aujourd'hui, cet objet de collection est négocié à plus de $1500)
Ce même procédé d'**approximation successive** est appliqué à la tension analogique lorsqu'elle est **échantillonnée** par le périphérique ADC.

1- un 'petit' condensateur (intégré dans le microcontrôleur) est chargé par la tension d'entrée souhaitée. (cela permet de stabiliser la valeur mesurée). On parle d'échantillonneur-**bloqueur**.
2- successivement, des **comparaisons** sont effectuées, selon la **dichotomie** présentée ci-dessus. La résolution de l'ADC donne le nombre de comparaisons. (souvent 8, 10 ou 12 bits par exemple).
3- on obtient un nombre qui est mis à disposition par le périphérique, dans un registre.
Deux références (+ et -) encadrent ces grandeurs. Ces références ne sont pas forcément 0V et VDD... On peut souvent choisir d'amener des tensions références.

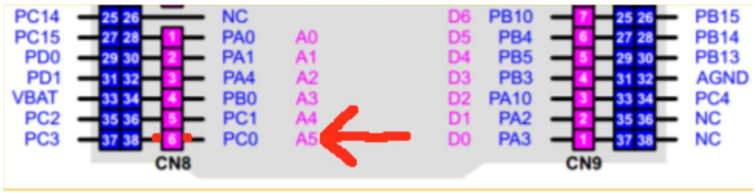
Le STM32 dispose d'un ADC sur 12 bits (références : 0V et 3V). Le nombre obtenu va donc de 0 à 4095.
Nombre_obtenu = tension * 4096 / 3V.
Attention, il ne faut pas confondre résolution et précision.
Une balance de cuisine peut être résolue au microgramme, alors que sa précision est de 10g. C'est idiot. A l'inverse, une précision de 10g pour une résolution de 100g permet d'être sûr que le résultat présenté est correct. Une résolution de x bits nous donne x étapes pour s'approcher du résultat.

Le module logiciel stm32f1_adc.c/h qui vous est fourni s'appuie sur le module hal_stm32f1xx_adc.c/h.
Il vous permet de :

- Initialiser l'ADC1 du microcontrôleur, dont la lecture des entrées analogique sera cadencée au rythme d'1 mesure par ms (grâce au timer 3).
  - `void ADC_init(void)`
- Consulter à tout moment la valeur du dernier échantillon disponible pour le canal souhaité
  - `int16_t ADC_getValue(adc_id_e channel)`
- Savoir si une nouvelle donnée est disponible depuis notre précédente demande
  - `bool_e ADC_is_new_sample_available(void)`

- Activez les modules logiciels Timer et ADC fournis via les macros disponibles dans config.h :

```
#define USE_BSP_TIMER                   1       //Utilisation de stm32f1_timer.c/h
#define USE_ADC                         1       //Utilisation de stm32f1_adc.c/h

//On active le canal AN10 qui correspond à la broche PC0, notée A5 sur la Nucleo
#define USE_AN10                        1       //Broche correspondante : PC0
```



Lorsque vous modifiez le contenu de config.h, prenez l'habitude de demander ensuite à eclipse de rafraîchir sa lecture des fichiers pour qu'il prenne conscience des changements impliqués (cela impacte les soulignements en rouge dans Eclipse, mais n'a aucune influence sur la compilation) :

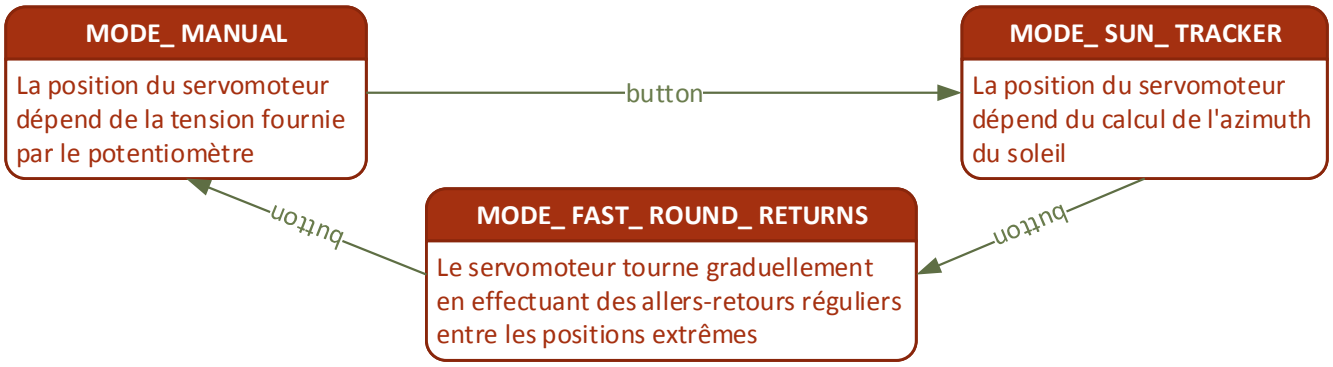
- Bouton-droit sur le projet -> Index -> Freshen All Files
  (Ou : Project -> C/C++ Index -> Freshen All Files)

- Dans **`SUN_TRACKER_init()`**, appelez ADC_init();

- Dans **`SUN_TRACKER_process_main()`** : Affectez le résultat fourni par `ADC_getValue(ADC_CHANNEL_10)` dans la variable `position` qui pilote le servomoteur, juste avant l'appel de la fonction `SERVO_set_position(position)`.
  - Attention, une mise à l'échelle s'impose ; ADC_getValue renvoie un résultat sur 12 bits (nombre de 0 à 4095) alors que l'on attend une position de 0 à 100.

8. Vers plusieurs modes de fonctionnements

On souhaite maintenant être en mesure de changer de mode à chaque appui sur le bouton.



- Dans le fichier sun_tracker.c, créez la variable statique suivante :

```
static uint16_t sun_tracker_position = 50;
```

- Remplacez le pilotage du servomoteur qui était fait dans **SUN_TRACKER_process_main** par une simple affectation de la variable `sun_tracker_position` à la valeur souhaitée.

Il faut maintenant gérer le bouton poussoir pour changer de mode à chaque appui.

Attention, il y a une différence entre :
- Un appui sur le bouton (c'est un évènement qui se produit une fois à chaque nouvel appui)
- Le bouton est appuyé (cela peut être vrai en continu tant que le bouton est pressé)

Pour détecter un appui sur le bouton poussoir, on propose cette fonction :

```
/*
 * @brief    détecteur d'appui sur le bouton bleu
 * @ret      cette fonction détecte chaque nouvel appui sur le bouton et renvoit vrai
à chaque appui
 * @pre      pour éviter les effets dûs aux rebonds, cette fonction doit idéalement
être appelée toutes les 10ms
 */
bool_e button_press_event(void)
{
	static bool_e previous_button = FALSE; //état précédent du bouton
	bool_e ret = FALSE;
	bool_e current_button;                 //état actuel du bouton

	//bouton en logique inverse, d'où le '!'
	current_button = !HAL_GPIO_ReadPin(BLUE_BUTTON_GPIO, BLUE_BUTTON_PIN);

	//si le bouton est appuyé et ne l'était pas avant, champomy !
	if(current_button && !previous_button)
		ret = TRUE;

	//on mémorise l'état actuel pour le prochain passage
	previous_button = current_button;
	return ret;
}
```

- Lisez le chapitre 17 – Enumérations
- Lisez le chapitre 19 – Switch

Il nous faut maintenant prendre en compte cet appui bouton pour changer de mode.

- Ajoutez en début de fichier :

```
static uint32_t t_read_button = 0;
```

- Ajoutez dans la fonction process_ms :

```
      if(t_read_button)
            t_read_button--;
```

- Ajoutez une fonction et complétez là selon les instructions indiquées :

```
static void SUN_TRACKER_mode_management(void)
{
      //Definition d'une énumération mode_e définissant 3 modes :
      //MODE_SUN_TRACKER,MODE_MANUAL, MODE_FAST_ROUND_RETURNS

      //créez la variable 'mode', static de type mode_e, initialisé à MODE_SUN_TRACKER
      //créez un entier 'position', non signé sur 16 bits

      if(!t_read_button)        //le chronomètre est épuisé
      {
            t_read_button = 10;//on recharge le chronomètre pour 10ms

            //ajoutez ici un switch sur la variable mode
            //ce switch reprend l'ensemble des valeurs de l'énumération
            /*
            dans le MODE_MANUAL :
                  //TODO acquérir la valeur du potentiomètre
                  //position = ...;
                  Si(évènement bouton)
                  {
                        //passer au MODE_SUN_TRACKER
                  }

            dans le MODE_SUN_TRACKER :
                  position = sun_tracker_position;
                  Si(évènement bouton)
                  {
                        //passer au MODE_FAST_ROUND_RETURNS
                  }
            dans le mode MODE_FAST_ROUND_RETURNS :
                  imaginez un algo pour faire évoluer la position de 1 à chaque appel
                  avec changement de sens quand on arrive aux extrêmes (0 ou 100)
                  Si(évènement bouton)
                  {
                        //passer au MODE_MANUAL
                  }
            */
            //Pilotage du servomoteur avec l'argument 'position'
      }
}
```

- Appelez cette fonction dans la fonction SUN_TRACKER_process_main
    - en dehors du `if(flag_1s)`, (la lecture du bouton doit être réalisée bien plus souvent)

```
SUN_TRACKER_mode_management();
```

A ce stade, on peut tester l'application qui nous permet de naviguer entre les 3 modes.
L'usage de débogueur (breakpoint, pas à pas, ...) sera probablement nécessaire en cas de problème.

9. Bonus : téléportation

- Ajoutez une fonctionnalité au SECRETARY, permettant à l'utilisateur de saisir une nouvelle localisation GPS.
    - Par exemple en indiquant dans le terminal : `setloc 48.783990, 2.210231`
    - La fonction atof() pourrait s'avérer utile