TP 2 : Emetteur / récepteur lumineux

Programmation temps-réel

Objectif pédagogique

Sensibilisation aux problèmes de synchronisation et de temps lors de la transmission de données entre un émetteur et un récepteur indépendants dans le cadre d'un projet complet. Utilisation d'un environnement multi-tâche (RTOS) pour la gestion de tâches asynchrones.

Préparation et contenu de l'archive initiale du projet

- A l'aide de LPCXpresso, décompressez l'archive tp2_light_com_RB.zip fournie. Cette archive contient 2 projets : un pour l'émetteur l'autre pour le récepteur.
- Renommez G***x***_TP2_light_com_etu avec ***x*** correspondant au numéro de votre groupe.
- Renommez G***x***_TP2_light_tx_etu avec ***x*** correspondant au numéro de votre groupe.

Contenu initial des projets :

- G***x***_TP2_light_com_tx :
    - Fonction *calc_checsum()* servant à calculer le checksum sur une chaîne de caractères
    - Librairie ***MyLab_lib.a*** et ***ledrgb_dma.h*** : routine de gestion de la led RGB
    - *main()* illustrant l'utilisation de la routine *calc_checsum()* et de la led RGB
    - ***TP2_light_Tx_ref.axf*** *:* l'exécutable d'un émetteur fonctionnel pour tester le récepteur (dans le répertoire *Debug*)

- G***x***_TP2_light_com_rx :
    - Librairie ***MyLab_lib.a*** et ses headers :
        - ***lcd.h*** : routines de gestion de l'écran
        - ***traces_ref.h*** : routines de gestion des traces
        - ***uart.h*** : routines de gestion de l'UART
    - FreeRTOS et son fichier de configuration : ***FreeRTOSConfig.h***
    - ***debug.c, debugh*** : module de debug permettant de sauver un signal dans un fichier texte (pour lecture avec Octave)
    - ***light_com.c*** : contient le *main()* avec un exemple d'acquisition d'échantillons provenant du capteur RGB, ainsi que l'initialisation des traces et de l'UART.
    - ***ext_color_sensor.h***, ***ext_color_sensor.o*** : routines de gestion du capteur RGB
    - dans /scripts ***:***
        - ***uart2file_and_sending :*** script permettant d'envoyer des commandes par UART et de recevoir et de convertir les traces de timings des tâches
        - ***tp2_corr.m*** : script Matlab/Octave permettant de décoder le signal acquis par le récepteur
        - ***tp2_ref.txt*** : signal d'exemple (à visualiser avec ***tp2_corr.m*** comme démo)

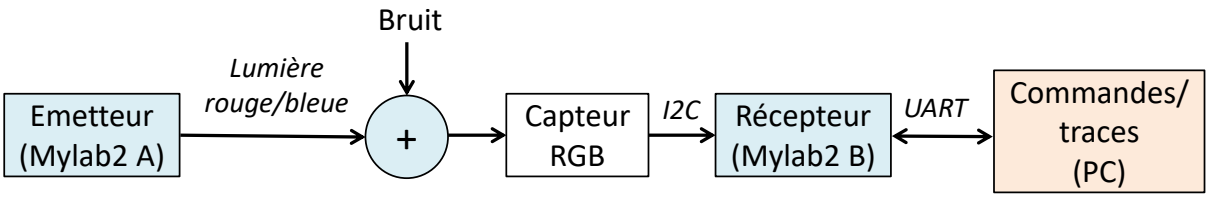
Note : la documentation des routines utilitaires fournies se trouve dans chaque header.

Spécification

Description générale

L'objectif de ce TP est de construire un petit système contenant un émetteur de lumière (constitué par une carte LPCXpresso et de sa carte d'extension) d'une part et d'autre part un récepteur de lumière indépendant, constitué d'une seconde carte. Le récepteur doit aussi pouvoir exécuter des commandes simples provenant d'un PC (par l'UART) et évacuer les traces des timings des tâches vers ce même PC.

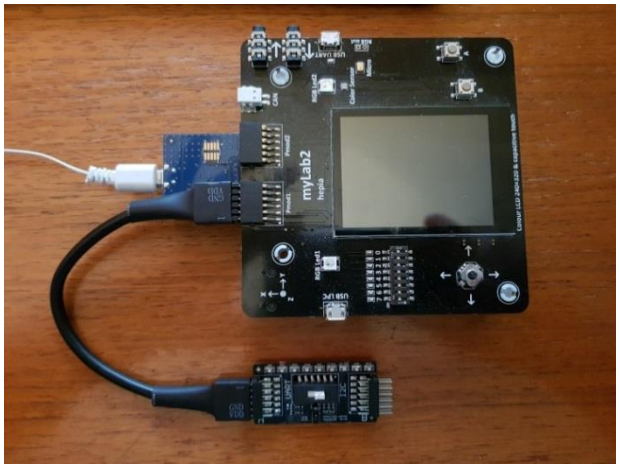
Le système est illustré schématiquement ci-dessous.



L'émetteur (carte Mylab2 A) produira un signal lumineux rouge et bleu à l'aide de la led RGB que le récepteur (carte Mylab2 B) recevra à l'aide de son capteur RGB. Contrairement à l'exercice de la série 6b, ici le bruit n'est pas ajouté volontairement, mais par l'imperfection d'une transmission réelle, qui est d'autant plus marquée lorsque l'émetteur s'éloigne du récepteur. Mais grâce au traitement de signal, il est possible de limiter l'effet de ce bruit de façon significative afin de permettre au récepteur de comprendre la teneur du signal émis à une distance raisonnable.

L'émetteur transmettra régulièrement des chaînes de caractères codées que le récepteur affichera sur son écran LCD de façon asynchrone. Cela signifie que le récepteur peut rater une ou plusieurs phrases, mais lorsqu'il décide d'afficher une phrase, il doit l'afficher correctement du début à la fin.

Le détail de la transmission et les commandes provenant du PC que récepteur doit pouvoir interpréter sont décrites dans le chapitre qui suit. Le branchement du capteur RGB sur le récepteur doit se faire comme ci-dessous avec la carte d'extension pmod. Positionnez l'interrupteur sur la position I2C. Il s'agira de placer le capteur de cette carte en face de l'émetteur (carte A) pour assurer la transmission.



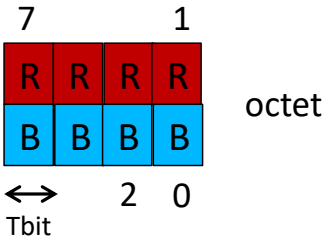
Description détaillée

L'émetteur

L'émetteur doit émettre des trames de 18 octets à un rythme régulier. Chaque octet est transmis du bit de poids fort (MSB ou *Most Significant Bit*) au bit de poids faible (LSB ou *Least Significant Bit*) de la manière suivante :

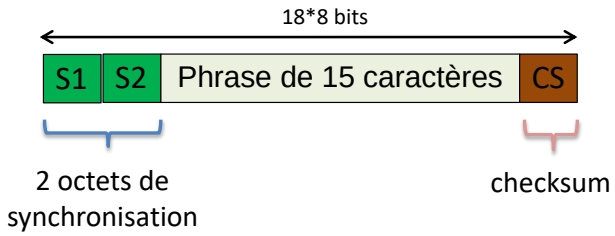
- Un '1' doit allumer la led rouge ou bleue selon la figure ci-dessous pendant une période Tbit
- Un '0' éteint la led rouge ou bleue pendant une période Tbit
- La période Tbit d'un bit est de 8 ms

Chaque octet doit alterner la couleur qui représente ses bits ainsi :



Comme 2 couleurs sont utilisées en même temps, **2 bits peuvent être codés à la fois dans un intervalle Tbit**. Le vert a été volontairement écarté, car sa longueur d'onde se trouve entre celle du rouge et du bleu et le capteur n'étant pas très sélectif, il pourrait mal interpréter cette couleur.

Le format d'une trame est le suivant :



Les 2 octets en début de chaque trame serviront à la **synchronisation** du récepteur (S1 et S2). Les valeurs de S1S2 sont constantes et imposées à `"\x95\x1B"`. Le dernier octet est un *checksum* qui sert au récepteur pour vérifier l'intégrité du contenu du message (la phrase ici). Le calcul du checksum est un simple « ou » logique des 15 caractères émis. La routine qui le calcule vous est fournie.

La taille d'une trame est donc de 2 + 15 + 1 = 18 octets.

L'émetteur doit envoyer successivement des phrases différentes (au moins 6), mais celles-ci peuvent être envoyées en boucle.

La librairie permettant de contrôler la led RGB est incluse dans le projet G***x***_TP2_light_com_tx: la fonction **`set_rgb_led_color(int color)`** permet d'allumer la led RGB avec la couleur choisie, codée ainsi : 0xRRGGBB. Dans votre cas les 4 seules couleurs à utiliser sont le rouge (0xFF0000), le bleu (0xFF), rouge et bleu (0xFF00FF) ou noir (0x0).

Note importante: pour obtenir les couleurs correctes sur la led RGB, branchez aussi un câble USB sur le connecteur USB de Mylab2 afin d'alimenter correctement la led.

Conseils de développement

- **La précision temporelle de la formation des bits est cruciale**
- Rappel : n'utilisez pas de fonction printf ou sprintf dans la boucle qui gère l'envoi des bits : ces fonctions peuvent fausser les timings !
- N'utilisez pas FreeRTOS
- Ce code doit rester simple et fonctionnel : il n'a pas besoin d'être « esthétique »

Le récepteur

Le récepteur est constitué de 4 tâches qui sont décrites ci-dessous. La tâche 4 doit avoir la plus haute priorité et toutes les autres ont la même priorité (intermédiaire). Le système de traces de timings vous est fourni.

Configuration de FreeRTOS :

FreeRTOS doit être configuré en mode préemptif avec un *tickrate* de 100 Hz pour ce TP.

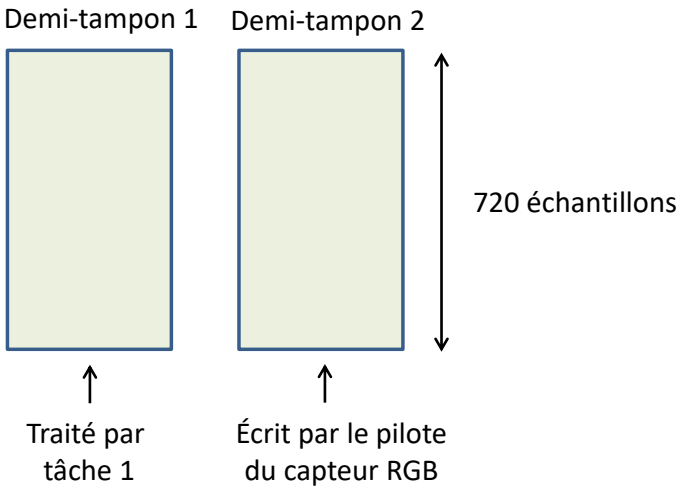
A part l'accès aux signaux (qui sont stockés dans des tampons globaux), utilisez les primitives de synchronisation standard du RTOS pour communiquer : sémaphores, queues, etc… Notez que le nom des primitives appelées depuis une interruption finit par FromISR (voir documentation de FreeRTOS).

Les délais doivent être passifs et utilisent aussi les primitives de FreeRTOS dans ce but.

1. Tâche de réception du signal lumineux

C'est la tâche principale du récepteur. Le signal lumineux est traduit en signal électrique par le capteur RGB de la carte d'extension pmod. Cette carte doit être branchée sur le connecteur pmod1 du récepteur. Faites attention de bien connecter les pins 1 ensemble !

Le signal électrique du capteur est ensuite converti en nombres, transféré par port I2C, puis stocké dans un double tampon par le pilote du capteur RGB (fourni, voir **`ext_color_sensor.h`**). Les données sont écrites dans un tampon par le pilote et ce dernier appelle une fonction de « callback » par interruption, chaque fois qu'un demi-tampon est rempli. Le corps de la fonction de callback peut être rempli par le programmeur pour avertir la tâche 1 qu'un tampon est plein (voir annexe pour l'utilisation d'une fonction de *callback*). Cela permet à la tâche 1 d'effectuer la détection du message reçu sur le (demi) tampon qui vient d'être rempli, alors que l'autre est en cours de remplissage.



Chaque échantillon reçu par le capteur est une structure qui contient les données suivantes :

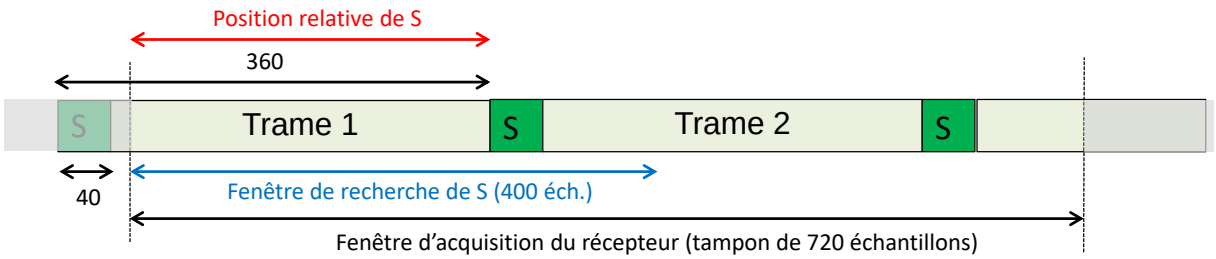
```
typedef struct
{
	uint16_t clear; // sorte de somme de toutes les composantes de couleur
	uint16_t red;
	uint16_t green;
	uint16_t blue;
} ext_cs_t;
```

Ainsi seuls les champs *red* et *blue* nous intéressent. La configuration du capteur limite leur dynamique à 1025. Ils peuvent donc contenir des valeurs entre 0 (absence de composante) et 1025 (couleur vive).

Les composantes rouges et bleues nous servent à reconstituer le signal envoyé par l'émetteur. La fréquence d'échantillonnage choisie du récepteur est de 625 Hz, ce qui signifie que chaque paire de « bits » reçue est représentée **sur 5 échantillons**. Comme la taille d'un demi-tampon est de 720 échantillons, il faudra donc 720/625 = 1.152 seconde pour le remplir.

Synchronisation

Comme l'acquisition du signal commence à un temps arbitraire, il est probable que les trames reçues soient décalées par rapport au début du tampon, comme illustré ci-dessous.

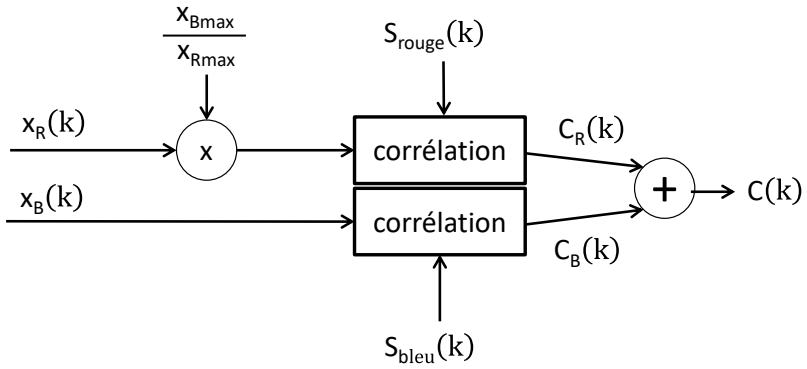


Mais comme la durée de l'acquisition couvre 2 trames, l'une d'elles sera toujours valide et il sera possible de la décoder.

Le but ici est de rechercher la position de S sur la moitié des échantillons acquis. En effet : en recherchant S sur 400 échantillons, on est sûr de pouvoir trouver l'une de ses 2 apparitions sur cette plage. Une fois cette position déterminée, les bits de données peuvent être lus correctement, sachant qu'ils suivent S.

Afin de détecter la position de S sur le signal perturbé reçu (exemple ci-dessous), chaque composante du signal reçu (R et B) doivent être corrélées avec les bits R et B de la séquence connue. La position du maximum de corrélation sur R et B (somme) donne la position de S.

Voici le schéma de traitement à appliquer au signal d'entrée x(k) :



$x_R(k)$ et $x_B(k)$ est le vecteur d'échantillons stocké dans le tampon

$S_R(k)$ est formé sur la base des 8 bits impairs de S (sur les 16 qui constituent S), chaque bit à 1 étant représenté par une suite de 5 échantillons à 1 et chaque bit négatif par une suite de 5 échantillons à -1 . De façon identique, $S_B(k)$ est la suite de bits pairs de S. Afin de donner le même poids aux 2 signaux entrants, on mesure la valeur maximum des 2 signaux sur les 720 échantillons reçus et on normalise l'un des signaux (par exemple le rouge, comme sur la figure ci-dessus).

La position de S peut être déterminée grâce au résultat de la corrélation entre la séquence S et le signal reçu en considérant l'indice k_{max} pour lequel elle est maximum :

$$pos(S) = k_{max} = \arg\max(C(k)) \ \ pour\ k \in \{0,1,\dots,399\}$$

Une fois la position de S connue, les bits de données (rouge et bleu) peuvent ensuite être reconstitués à la position pos(S) + Tbit/2 + Tbit * n simplement en repérant si la valeur des échantillons dépasse la moitié de la valeur maximum reçue: si le seuil est dépassé, cela indique un '1', le contraire un '0'. Autrement dit, les bits de la composante rouge sont recouvrés avec la formule suivante:

$$bit_R(n) = x_R\left[k_{max} + \frac{T_{bit}}{2} + nT_{bit}\right] > \frac{x_{Bmax}}{2}\ pour\ n \in \{0,1,\dots,18*8-1\}$$

Tbit = 5 échantillons

Ou, plus généralement pour reconstituer tous les bits de la trame avec les 2 composantes rouge et bleue :

$$bit(2n+1) = x_R\left[k_{max} + \frac{T_{bit}}{2} + nT_{bit}\right] > \frac{x_{Bmax}}{2}$$

$$bit(2n) = x_B\left[k_{max} + \frac{T_{bit}}{2} + nT_{bit}\right] > \frac{x_{Bmax}}{2}$$

Une fois que les bits de données ont été reconstitués, rassemblez-les par 8 pour reformer la trame complète envoyée par l'émetteur.

On estime qu'une chaîne de caractère est « correctement » reçue que lorsque la séquence S est parfaitement décodée par le récepteur. 3 cas peuvent se présenter et doivent être traités :

1) réception correcte : S est bien décodé, et le checksum appliqué sur les 15 caractères de données est correct

2) réception corrompue : S est bien décodé, mais le checksum appliqué sur les 15 caractères de données n'est pas correct

3) pas de réception : S n'est pas décodé correctement

La tâche de réception doit envoyer une chaîne de caractères comportant un message à la tâche 2 (qui s'occupera de l'afficher) selon le cas :

1) envoi de la chaîne de caractère décodée

2) envoie « Bad checksum »

3) envoie « Nothing received »

2. Tâche d'affichage et d'interprétation de commandes

Cette tâche a deux rôles :

a) Elle s'occupe d'afficher les chaînes de caractères reçues par l'émetteur sur l'écran LCD de la carte d'extension LPCXpresso.

b) Elle interprète les commandes reçue du PC par UART, les affiche et les exécute.

Les chaînes de caractères arrivent de façon asynchrone, par contre leur affichage doit être synchrone.

a) Par défaut, l'affichage consiste à écrire une chaîne de caractères toutes les **450 ms** (*scroll_delay*). Utilisez ***lcd_printf()***. Cette chaîne de caractère doit être dans l'ordre de priorité :

1) celle qui a été envoyée par la tâche 1 (si il y en a une)

2) celle qui provient d'une commande (voir b) ci-dessous)

3) un retour de ligne (“\n “) si rien n'a été reçu en 1) et 2)

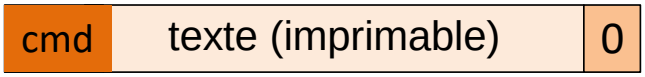
Ainsi, l'affichage par défaut fait défiler régulièrement les chaînes de caractères reçues du bas vers le haut de l'écran. Les couleurs d'affichage sont les suivantes :

1) blanc (LCD_WHITE), mais cette couleur peut être changée par la commande 'color'

2) violet (LCD_MAGENTA) pour les commandes et blanc pour le contenu de la commande 'send' (voir b))

b) La ligne de réception de l'UART permet de recevoir des chaînes de caractères qui doivent être interprétées par un interpréteur de commandes. L'initialisation de l'UART se fait en appelant la fonction existante `uart0_init_ref()` définie dans ***uart.h***. le débit de l'UART doit être fixé à 115200 bits/s. Une fonction de « *callback* »[1] doit être définie à l'appel de cette fonction : celle-ci sera appelée chaque fois qu'un caractère est reçu sur l'UART. Le caractère reçu peut être lu dans le registre `LPC_UART0->RBR`.

Chaque commande commence (sauf *send*) commence par un caractère « non-imprimable » et se termine par un zéro. Le format des commandes est le suivant :



Les commandes peuvent être envoyées par le PC (avec l'UART connecté) grâce au script ***uart2file_and_cmd_sending***. Une fois lancé, celui-ci permet d'envoyer des commandes parmi celles définies dans le tableau suivant :

Commande (syntaxe du script)	**cmd**	**Corps de la commande (texte imprimable)**	**Fonctionnalité côté LPC1769**
send <text_file>	rien	Tout le texte du fichier *text_file* [2]	Afficher le texte par segment. Un segment est défini par maximum 30 caractères successifs ou par le caractère '\n' ou '\x0'. Un seul segment doit être affiché toute les *scroll_delay* [ms].
color <col>	0x2	Les mots clés reconnus sont : *white, yellow, red, green, blue*	Change la couleur des phrases reçue par la carte émettrice.
scroll <mode>	0x3	Les mots clés reconnus sont : *slow, fast, stop*	*slow* : impose *scroll_delay* = 450 [ms] *fast* : impose *scroll_delay* = 220 [ms] *stop* : arrête le défilement automatique, c'est à dire cesse d'insérer des retours de ligne (point a.3)
leds <mode>	0x4	Les mots clés reconnus sont : *on, off*	*on* : active la tâche des leds *off* : désactive la tâche des leds

[1] voir description en annexe
[2] qui doit contenir uniquement du texte et des retours de ligne

load <delay>	0x5	Nombre entre 0 et 15 compris	*delay* : délai d'attente active de la tâche de charge CPU en [ms]

3. Tâche de gestion des leds

Cette tâche allume puis éteint successivement chacune des 8 leds de la carte Mylab2 de droite à gauche avec un intervalle de 100 ms entre chaque allumage. Par défaut, cette tâche ne fait rien. Elle ne s'active que lorsque la commande « leds on » est reçue (voir tâche 2). La commande « leds off » stoppe le défilement d'allumage des leds.

4. Tâche de charge du CPU

Cette tâche consiste à charger le CPU sur demande, afin de simuler un calcul prioritaire. Pour cela, elle doit avoir la plus haute priorité. Par défaut, sa charge est de 0 ms / 20 ms, mais la commande « load » permet de changer cette valeur pour atteindre jusqu'à 15 ms de charge active toutes les 20 ms. Implémentez une boucle vide de la bonne durée pour cela.

5. Les traces de timing des tâches

Le système de traces est déjà implémenté : appelez

```
init_traces(115200, 2, true);
```

au début du programme pour l'activer et définissez

```
#define configHEPIA_TRACING   MYLABLIB_TRACES
```

dans FreeRTOSConfig.h. `configHEPIA_TRACING` doit être mis à 0 si les traces ne sont pas utilisées.

Les traces pourront être récupérées côté PC avec le script ***uart2file_and_cmd_sending*** qui effectuera directement la conversion de celles-ci au format VCD. Une brève analyse des traces est demandée dans le mini-rapport à fournir.

Contraintes d'implémentation

- Les seules variables globales admises sont :
    - La déclaration des variables contenant des signaux (ex : échantillons RGB)
    - Les variables utiles au RTOS (ex : sémaphore, mutex, queue)
    - Les constantes (variables avec le mot-clé ***const***)
- Utilisez des primitives du RTOS pour communiquer entre les tâches/callbacks

- N'utilisez pas de timers dans le code du récepteur, sauf pour la tâche 4. Note : les timers 2 et 3 sont déjà utilisés par les pilotes fournis.

Démarche de développement conseillée

1. L'émission : Codez l'émetteur. Pour valider son fonctionnement, utilisez un récepteur simple qui acquiert 720 échantillons et les stocke dans un fichier (comme codé dans le *main()* d'origine du récepteur). Le fichier peut directement être lu par le script Matlab/Octave **tp2_corr.m**: ce dernier effectuera le calcul du récepteur et le décodage. Si votre émetteur fonctionne, le script doit pouvoir retrouver la chaîne de caractères que vous avez émise.

2. La réception :

    a. Travaillez **sans FreeRTOS** au départ. Appliquez au signal le traitement décrit dans la tâche 1. Pour la corrélation, convertissez en C la version que vous avez déjà développée sous Matlab/Octave et adaptez-la pour permettre la détection de S. Travaillez sur **une seule** fenêtre d'acquisition et comparez votre résultat avec celui du script tp2.m. Vous pouvez également sauver vos signaux avec la fonction **`int2file`**( ) et les visualiser ensuite sous Matlab/Octave. Utilisez les tampons prédéfinis dans le projet et laissez leur définitions en variables globales. Faites attention à la dynamique des nombres lors des calculs !

    b. Effectuez le décodage des bits de données à partir de la position de S

    c. Regroupez les bits en caractères et affichez la chaîne reçue sur l'écran LCD (sans défilement, juste avec *lcd_printf()*)

    d. Otez les fonctions **`int2file`**() et faites tourner le programme en boucle : les chaînes de caractères doivent s'afficher les unes après les autres sur l'afficheur LCD.

3. Développez le code qui sera le cœur de la tâche 2 : interprétation des commandes UART et affichage (sans FreeRTOS)

4. Implémentez la gestion des leds et de charge CPU

5. Intégrez l'ensemble des fonctions ci-dessus dans FreeRTOS et ajoutez les moyens de communication nécessaires

6. Testez toutes les commandes possibles côté PC, dont la commande « send » avec le fichier texte fourni (*test.txt*)

7. Analysez vos traces VCD avec gtkwave afin de pouvoir les commenter dans le mini-rapport

Travail à rendre

- Une archive générée par LPCXpresso (*File->export->archive file*) contenant 2 projets : celui de l'émetteur et celui du récepteur. Appelez l'archive ***Gx_TP2.zip***, où x est votre numéro de groupe
- Un mini-rapport dont la consigne est donnée en annexe. Nommez ce fichier ***Gx_rapport_TP2.pdf*** *(x est le N° de groupe)*
- Si vous n'arrivez pas à faire fonctionner l'ensemble du projet, envoyez les « briques » de code que vous avez construites séparément selon la démarche de développement proposée ci-dessus et indiquez dans le rapport ce qui fonctionne (et ce qui ne fonctionne pas).

Chaque projet rendu doit pouvoir compiler sans erreur, sans warning et exécuter au minimum une partie du TP.

ANNEXE

Fonction de callback

Une fonction de « callback » ou de rappel permet à une fonction d'en appeler une autre dont le prototype est connu, mais pas le contenu. Ce procédé permet une grande souplesse d'utilisation, car l'utilisateur pour utiliser un code à priori inconnu (par exemple une librairie), tout en y rajoutant le sien, mais sans avoir à modifier le code fourni. Dans notre cas, la librairie de l'UART et du capteur appellent leur fonction de *callback* depuis une routine d'interruption.

Il faut juste se faire à la syntaxe, car elle fait appel aux pointeurs de fonction et est un peu particulière. Exemple :

```
// définition d'un type de fonction qui retourne un entier et qui passe un entier en paramètre

typedef int (*callback_t)(int buffer_index);

// la fontion suivante d'une librairie inconnue a besoin d'une fonction de rappel

void unknown_func(int a, callback_t user_func);

// on peut définir une fonction ayant le bon prototype et la passer en paramètre à
// unknown_func. Par exemple :

int ma_function(int b) { printf("b=%d\n", b); }

// L'appel de unknown_func se fera alors ainsi :

unknown_func(10, ma_fonction);
```

Voir http://en.wikipedia.org/wiki/Callback_%28computer_programming%29 pour plus d'information.

Contenu et format du mini-rapport à rendre

Ce rapport a pour objet de décrire très brièvement l'état du/des projet(s) rendu(s), de mentionner les anomalies observées (s'il y en a) et de les commenter. **Le rapport ne doit pas dépasser 4 pages A4**. Si la complexité d'un morceau de code ou d'un algorithme doit être éclaircie, cela peut être fait dans le rapport mais de façon succincte et si possible illustrée (organigramme).

- Titre : mention du cours, numéro du groupe, noms des participants et date
- Indiquez l'état général de votre développement au moment où le code a été rendu : état de l'émetteur, de chaque tâche du récepteur, etc. Si le code n'est pas terminé, expliquez pourquoi et quelle aurait été votre démarche pour pouvoir le finir.
- Expliquez les anomalies que vous avez observées (sur le code rendu uniquement) et donnez une piste pour les expliquer
- Montrez un extrait des traces des tâches que vous obtenez lorsque le récepteur est très sollicité et justifiez-le
- **Ne répétez pas ce qui est décrit dans la présente spécification !** Faites-y référence si nécessaire.