UNIVERSITÉ SAVOIE MONT BLANC

Travaux Pratiques

Programmation C

pour les Systèmes Embarqués

1 Création d'un projet sur STM32

1.1 Projet "Hello world"

On rappelle qu'une application *hello world* doit réaliser une action simple (clignotement LED, transmission UART…) juste pour valider le bon fonctionnement de votre chaine de compilation. Pour cette partie, vous pourrez utilisez toutes les librairies HAL (**H**ardware **A**bstraction **L**ayer), proposées par ST. Voici un exemple pour gérer une GPIO d'un port :

```
HAL_GPIO_WritePin(GPIOx, GPIO_Pin, PinState)
```

- Avec STM32CubeIDE, créer une application "STM32F446_hello_world" (**File > New > STM32 Project > Board Selector > NUCLEO-446RE**).

A noter que nous avons utilisé une création de projet qui connait la carte Nucleo (**board selector**) sur laquelle nous travaillons, c'est-à-dire que les principaux périphériques de la carte sont déjà connus. Le STM32 est donc déjà configuré en conséquence : PA5 en sortie, PA3 et PA2 en RX/TX pour l'UART2, etc…

- Générer le code et tester une application simple (clignotement LED PA5).

1.2 Mémo des fonctionnalités STM32CubeIDE (Eclipse)

Eclipse est un environnement de développement puissant. Voici quelques-unes des fonctionnalités intéressantes :

Open declaration : Ctrl + clic sur une fonction, une variable, ou une MACRO. Pour revenir sur l'objet initial : Alt + ←

Search in project : Ctrl + H (select the radio button « enclosing project »)

Open call hierarchy : Clic droit > Open call hierarchy. Permet de connaitre tous les endroits de votre projet ou une fonction a été appelée.

1.3 Projet “From scratch”

1.3.1 Utilisation des MACROs ST et CMSIS

Maintenant que notre environnement de développement et notre carte STM32F446 sont opérationnels, nous allons réaliser un nouveau projet sans utiliser les librairies HAL fournies par ST.

Néanmoins, pour notre projet, nous nous accorderons le droit d’utiliser les MACROS de gestion des suivantes fournies par ST qui sont très équivalentes à celles que nous avons écrit en cours.

```
#define SET_BIT(REG, BIT)
#define CLEAR_BIT(REG, BIT)
#define READ_BIT(REG, BIT)
#define CLEAR_REG(REG)
#define WRITE_REG(REG, VAL)
#define MODIFY_REG(REG, CLEARMASK, SETMASK)
```

Si besoin, nous pourrons aussi utiliser les MACRO CMSIS pour par exemple activer une interruption :

```
NVIC_EnableIRQ(USART2_IRQn);
```

- Avec STM32CubeIDE, créer une application "STM32F446_from_scratch" (**File > New > STM32 Project > MCU/MPU selector > STM32446RE**).

A noter que cette fois-ci nous avons utilisé une création de projet qui connait uniquement le microcontrôleur STM32 et non pas la carte nucleo (**MCU/MPU selector**). Cela signifie que la configuration du STM32 devra être entièrement réalisée.

Nous ne modifierons pas la configuration par défaut. Cependant, dans l’onglet "clock configuration", vous devez noter la fréquence de cadencement affectée sur les différents bus car nous en aurons besoin plus tard.

- Générer le code sans modifier la configuration.

1.3.2 Nettoyage du code

- Dans les fichiers */core/Src* et */Core/Inc*, réorganiser votre code de façon à le rendre le plus clair possible, en supprimant tous les commentaires et les sauts de lignes inutiles. Vous ne pourrez plus générer du code avec CubeMX.

Vous mettrez la fonction *SystemClock_Config* dans un fichier *clock_config.c* . Puis, à chaque fois que vous utiliserez un périphérique (UART, GPIO, etc), vous mettrez le code dans des fichiers appropriés (uart.c, uart.h, ppio.c, gpio.h…).

1.3.3 Application Hello world from scratch

Nous allons écrire une application simple qui allume la LED PA5. Pour cela, vous définirez les adresses des registres dont vous avez besoin. Les valeurs de ces adresses se retrouve depuis le manuel de référence du STM32. Par exemple, le registre *ODR* pour piloter les sorties du GPIOA sera défini de la façon suivante :

```
uint32_t* const ptr_GPIOA_ODR = (uint32_t*) 0x40020014;
```

Voici la démarche pour réaliser l’application. Les étapes 1 et 2 seront à faire dans une fonction d’initialisation du type :

```
void GPIO_Init(void)
```

1. Pour utiliser un périphérique du microcontrôleur, il faut d'abord le cadencer. Pour cela, nous pouvons utiliser la MACRO suivante pour le GPIOA :

```
__GPIOA_CLK_ENABLE();
```

2. Mettre la broche PA5 en output.
3. Application : Mettre la broche PA5 à 1.

Une fois cette première application finalisée, réalisé un clignotement de LED toute les 100 ms. Pour cela vous pourrez ajouter la MACRO suivante à la liste des MACROS déjà proposées par ST:

```
#define TOGGLE_BIT(REG, BIT)
```

1.4 Evolution de l'application : librairie UART

Nous allons créer une librairie pour la gestion de la transmission de caractères sur l'UART2. Pour cela nous utiliserons la broche PA2 dont le câblage permet de récupérer les valeurs transmises via le connecteur USB ST-Link sans avoir connecter d'autre éléments. Voici la démarche :

1. Pour utiliser un périphérique du microcontrôleur, il faut d'abord le cadencer. Pour cela, nous pouvons utiliser la MACRO suivante pour l'USART2 :

```
__USART2_CLK_ENABLE();
```

2. Assigner PA2 à la fonction TX de l'UART2. Pour cela, la PIN doit être en *Alternate function mode* (registre *MODER*). Voir le manuel de référence.
3. Définir l'*Alternate function* comme étant l'UART2. (registre *AFR*). Voir le manuel de référence.

Suivre la procédure de configuration de l'UART (page 801 du manuel de référence) :

4. Activer l'UART2 : bit UE du registre *USART_CR1*.
5. Définir le registre *BRR* pour avoir un Baud Rate de 115200 bauds sur l'UART2 (voir chapitre 25.4.4 du manuel de référence).
6. Mettre le bit TE à 1 dans le registre *USART_CR1* pour préparer la transmission.

Enfin l'envoi se fait en écrivant dans le registre *USART_DR* et en vérifiant que celui-ci a bien été envoyé en scrutant le bit TXE du registre *USART_SR*.

1.5 Evolution de l'application : affichage par printf

- Ajouter la fonctionnalité d'un affichage d'une chaine de caractère sur la liaison série l'UART2 avec votre propre fonction printf dont le prototype est le suivant :

```
void myPrintf(uint8_t * text)
```

L'utilisation de *myPrintf* sera limitée car le nombre de paramètre ne permet pas de passer des arguments *%d*, *%s*… Nous allons donc mettre en place la fonction *printf* de la librairie standard en la redirigeant sur l'UART2.

- Implémenter la fonction *printf()* de la librairie standard *stdio.h*.

2 Algorithme de compression

2.1 Introduction

2.1.1 Objectifs

Le but de ce projet est d'écrire un programme sur microcontrôleur STM32F446 permettant de compresser du texte sans perte d'information, afin qu'il occupe moins d'espace. Après transmission sur un microcontrôleur cible (aussi un STM32F446), ce texte sera décompressé et affiché. Les algorithmes de compression sont nombreux. Celui proposé ici est l'algorithme de Huffman que nous allons coder en langage C.

2.1.2 Organisation et version du projet

Pour ce projet, nous repartirons de votre première application *from scratch* codée en première partie.

1. Nous vous recommandons d'utiliser un outil de versionning type "git". Chaque ajout de fonctionnalité de votre logiciel vous fera passer à la version supérieure.
2. Il est préférable que vous respectiez les propositions de prototypage des fonctions afin que le debugage soit plus aisé, mais vous pouvez les modifier si besoin.
3. Il est impératif que votre sortie Debug (sur l'UART2) soit la plus claire possible. Cela vous sera d'une grande utilité pour le dépannage de votre application.

2.1.3 Notation du projet

La grille de notation sur ce projet est la suivante :

Organisation du code :

1. Programmation multifichier/compilation séparée : 1 point
2. Organisation du fichier de sortie (Debug) et présentation des résultats : 1 point
3. Outils de commentaire Doxygene (1 point)
4. Outils de gestion de version (Github, etc…) : 1point

Avancement dans le code avant transmission :

1. Arbre de Huffman réalisé : 1 point
2. Affichage des codes de chaque caractère : 1 point
3. Création de l'entête : 1 point
4. Création du texte compressé complet : 2 points

Transmission

1. Transmission et réception du texte compressé

Avancement du code après réception :

1. Décompression de l'entête : 1 point
2. Décompression complète : 1 point

> **La note est ramenée sur 4 et compte pour l'examen.**

2.2 Principe de l'algorithme de Huffman

Une idée apparue très tôt en informatique pour compresser les données est basée sur la remarque suivante : les caractères d'un fichier sont habituellement codés sur un octet, donc tous sur le même nombre de bits. Il serait plus économique en termes d'espace disque de coder les caractères sur un nombre variable de bits, en utilisant **peu de bits pour les caractères fréquents** et **plus de bits pour les caractères rares**. Le codage choisi dépend donc du texte à compresser. Les propriétés d'un tel codage sont les suivantes :

1. Les caractères sont codés sur un nombre différent de bits (pas nécessairement un multiple de 8) ;
2. Les codes des **caractères fréquents sont courts** ; ceux des **caractères rares sont longs** ;
3. On doit pouvoir décoder le texte de façon unique.
4. Si c1 et c2 codent deux caractères différents, c1 ne doit pas être le début de c2 et c2 ne doit pas être le début de c1. Sinon il y aura confusion.

2.2.1 Calcul des occurrences des caractères

On prend l'exemple d'un texte dont le contenu est "Une banane".

On calcule tout d'abord le nombre d'occurrence de chaque caractère dans le texte à compresser. Dans l'exemple ci-dessous, on a le nombre d'occurrence du caractère 'espace', 'U', 'b', 'n', 'a' et 'e' :

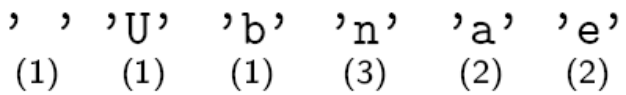


Figure 1 : Fréquence des caractères dans le texte

2.2.2 Création de l'arbre de Huffman

On prend ensuite les caractères qui possèdent la plus faible occurrence et on ajoute leur nombre d'occurrence pour réaliser un nouveau nœud.

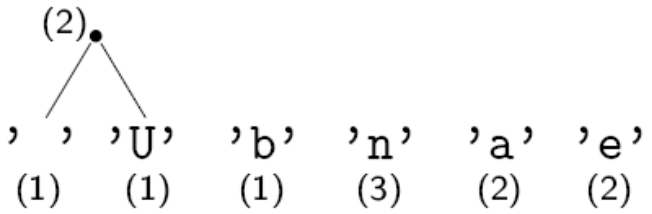


Figure 2 : Création d'un nouveau nœud

On fait exactement la même chose en reprenant en compte le nouveau nœud (et non plus les caractères espace et U). Dans l'exemple précédent l'occurrence la plus faible est celle de b et celle du nouveau nœud (on aurait pu prendre aussi 'a' ou 'e' puisque leur valeur est 2 aussi, cela revient au même à la fin).

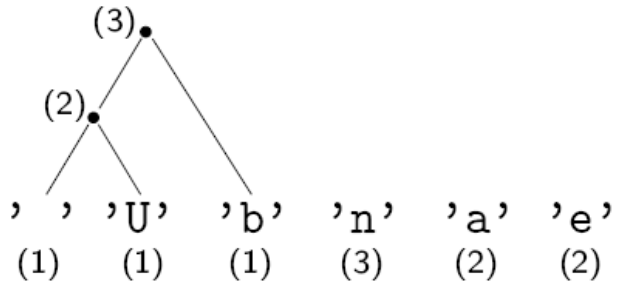


Figure 3 : Création d'un autre nouveau nœud

IMPORTANT : Dans toute la suite du TP, on appellera "une feuille" de l'arbre les éléments qui sont en bout de l'arbre (les caractères). On appellera un nœud les éléments regroupant deux feuilles (ou deux nœuds).

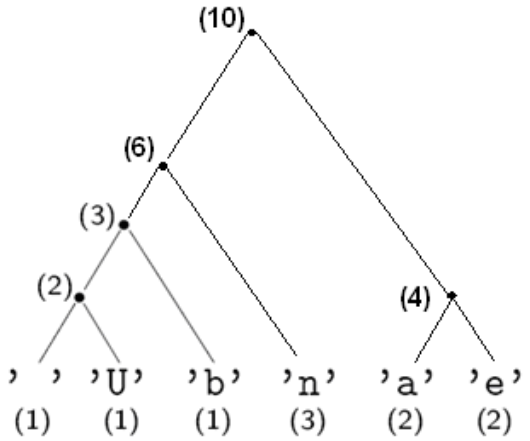


Figure 4 : Création complète de l'arbre de Huffman

Avec d'autres choix, on peut obtenir des arbres différents et tous sont correctes. À partir de l'arbre, on construit le code d'un caractère en lisant le chemin qui va de la racine au caractère. Un pas à gauche étant lu comme 0 binaire et un pas à droite comme 1 binaire.

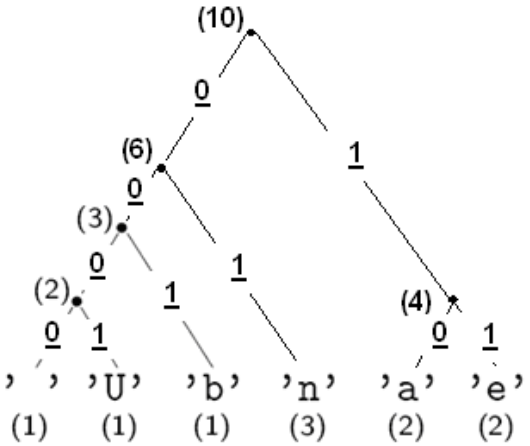


Figure 5 : Arbre de Huffman avec le codage des caractères

On lit le code des caractères en descendant depuis le haut jusqu'au caractère (en bas).

Caractère	' '	'U'	'a'	'b'	'e'	'n'
Fréquence	1	1	2	1	2	3
Code	0000	0001	10	001	11	01

Figure 6 : Code binaire de chaque caractère

Dans l'exemple précédent, on obtient la suite de bits **000101110000001100110011 1** pour le codage du mot "Une banane".

2.2.3 Taux de compression

"Une banane" est codé avec 10x8 bits = 80 bits en ASCII. Dans la version compressée, elle est codée avec 25 bits, nous avons donc économisé 55 bits, soit un taux de compression de 69%.

Cependant, avec cet algorithme, le décompresseur doit connaître les codes construits par le compresseur. Lors de la compression, le compresseur écrira donc ces codes en début de fichier, sous un format à définir, connu du compresseur et du décompresseur. Le fichier compressé aura donc deux parties disjointes :

- Une première partie permettant au décompresseur de retrouver le code de chaque caractère.
- Une seconde partie contenant la suite des codes des caractères du fichier à compresser.

Le fait de rajouter cet entête au fichier diminue considérablement le taux de compression pour les textes très court.

2.3 Travail demandé

La compression et la décompression se feront sur microcontrôleur STM32F446. Le texte d'origine sera écrit dans un tableau, mais on pourrait imaginer que celui-ci soit transmis à l'aide d'un autre bus de communication. Lorsque le texte sera compressé, il sera transmis par la liaison série UART1 à un autre microcontrôleur STM32F446.

Le nom des variables et des fonctions est souvent proposé afin d'avoir un code homogène et plus facilement debugable par l'enseignant. Nous allons donc respecter les variables suivantes :

```
#define TAILLE_MAX_COMPRESS 500   // A modifier si besoin.

// Texte non compressé
uint8_t     texte[]="aaaabbbccd";

// Texte compressé
uint8_t     texteCompress[TAILLE_MAX_COMPRESS];

// Tableau du nombre d'occurrence de chaque caractère
uint32_t    tabCaractere[256]={0};

// Nombre de caractère total dans le texte non compressé
uint32_t    nbrCaractereTotal = 0;

// Nombre de caractère différent dans le texte non compréssé
uint32_t    nbrCaractereDifferent = 0;

// Arbre de Huffman
struct noeud* arbreHuffman[256];
```

Le texte préconisé pour valider votre algorithme est "aaaabbbccd".

2.3.1 Comptage des occurrences des caractères

Nous nous intéressons au comptage des occurrences des caractères présents dans le texte non compressé. Pour cela nous utiliserons un tableau d'entiers :

```
uint32_t tabCaractere[256];
```

Ce tableau comporte 256 éléments. Dans chaque case de ce tableau, nous rentrerons le nombre d'occurrence du caractère dont le code ASCII est donné par l'index de tabCaractere.

Exemple : Le code ASCII du 'a' est 97. *tabCaractere*[97] comportera donc le nombre d'occurrence du caractère 'a' (4 dans notre exemple).

- Coder une fonction qui compte les occurrences de chaque caractère d'une chaine de caractère et qui stockera les occurrences dans un tableau.

```
void occurence( uint8_t* chaine , uint32_t tab[256] )
```

- *chaine* : adresse du texte
- *tab* : tableau des occurrences

- Afficher votre résultat sur votre interface de Debug UART2.

2.3.2 Création des feuilles de l'arbre de Huffman

Pour chaque caractère (ou nœud de l'arbre), nous allons créer une structure commune qui nous permettra de créer l'arbre et de créer le code. Nous avons appelé cette structure "nœud" mais en réalité elle sera aussi utilisée pour les feuilles de l'arbre.

La structure est donc définie comme suit :

```
struct noeud{
  uint8_t c;                    // Caractère initial
  uint32_t occurence;           // Nombre d'occurrences
  uint32_t code;                // Code binaire dans l'arbre
  uint32_t tailleCode;          // Nombre de bits du code
  struct noeud *gauche, *droite;// Lien vers les nœuds suivants
};
```

La seule différence entre un nœud et une feuille, est que pour une feuille, les pointeurs gauche et droite vers les nœuds suivant sont NULL. C'est de cette façon que nous les distinguerons. Voici la création des nœuds de l'arbre dans le cas de notre exemple "Une banane" :

' ' (1)	'U' (1)	'b' (1)	'n' (3)	'a' (2)	'e' (2)
struct nœud c=' ' occurrence=1 gauche=NULL droite=NULL code=0 tailleCode=0	**struct nœud** c='U' occurrence=1 gauche=NULL droite=NULL code=0 tailleCode=0	**struct nœud** c='b' occurrence=1 gauche=NULL droite=NULL code=0 tailleCode=0	**struct nœud** c='n' occurrence=3 gauche=NULL droite=NULL code=0 tailleCode=0	**struct nœud** c='a' occurrence=2 gauche=NULL droite=NULL code=0 tailleCode=0	**struct nœud** c='e' occurrence=2 gauche=NULL droite=NULL code=0 tailleCode=0

Figure 7 : Création des nœuds (struct nœud) de l'arbre

Pour connaitre les caractères présents dans le texte, il suffit de parcourir le tableau *tabCaractere*, et de vérifier les index pour lesquels il retourne un nombre d'occurrence non nul. Chacune de ces structures sera réservée en mémoire par la fonction *malloc()*. De plus, afin de conserver l'ensemble des structures créées, nous sauvegarderons les pointeurs sur ces structures nœud (*struct nœud**) dans un tableau :

```
struct noeud* arbreHuffman[256];
```

Ce tableau *arbreHuffman* aura donc le contenu suivant :

[0]	Adresse de la structure *nœud* du caractère ' '
[1]	Adresse de la structure *nœud* du caractère 'U'
[2]	Adresse de la structure *nœud* du caractère 'b'
[…]	…
[…]	…
[5]	Adresse de la structure *nœud* du caractère 'e'

La fonction aura le prototype suivant :

```
void creerFeuille(struct noeud * arbre[256], uint32_t tab[256] );
```

- arbre : Tableau de l'arbre de Huffman
- tab : Tableau d'occurrence des caractères

➡ Vous testerez votre programme en affichant depuis l'arbre de Huffman, le contenu de tous les champs des structures créées.

Nous avons alors créé toutes les feuilles de l'arbre de Huffman.

2.3.3 Affichage du tableau de l'arbre de Huffman

A de nombreuses reprises pour le Debug il sera intéressant de lire la partie significative du tableau arbreHuffman.

➡ Réaliser une fonction permettant de ligne "taille" case du tableau arbreHuffman. La fonction pourra avoir le prototype suivant :

```
void afficherTabArbreHuffman(struct noeud* arbre[256], uint32_t taille);
```

Vous pourrez afficher tous les champs de la structure et comparer l'évolution de votre tableau lors de la création de l'arbre.

2.3.4 Tri de l'arbre de Huffman

Nous allons maintenant faire apparaître des nouveaux nœuds dans l'arbre comme précisé dans la Figure 2. Pour cela, il est important de connaitre les deux caractères dont les occurrences sont les plus petites. Une façon simple pour connaitre les caractères dont les occurrences sont les plus faibles est de classer l'arbre de Huffman par ordre croissant : c'est-à-dire en plaçant en premier les pointeurs des structures dont les occurrences sont les plus les plus petites et en dernier les pointeurs des structures dont les occurrences sont les plus grandes. Il s'agit d'un simple tri de tableau. Les deux caractères dont les occurrences sont les plus petites se trouveront donc aux index 0 et 1.

Ce tri sera réalisé dans la fonction suivante :

```
void triArbre(struct noeud* arbre[256], uint32_t taille)
```

- arbre est l'arbre de huffman à trier
- taille est le nombre de case du tableau à trier

- Vous testerez votre programme en affichant depuis l'arbre de Huffman, le contenu de tous les champs des structures créées. Vous devriez avoir un affichage des caractères dont les occurrences sont les plus faibles en premier.

2.3.5 Création des nouveaux nœuds

Dès lors que nous avons trouvé les nœuds (ou feuille) dont les occurrences sont les plus faibles, nous allons créer un nouveau nœud dans l'arbre de Huffman. Celui-ci aura une valeur d'occurrence qui correspond à la somme des deux nœuds d'origine comme préciser dans la Figure 2. De plus, ce nouveau nœud remplacera les deux nœuds d'origine dans l'arbre de Huffman.

Note : Vous avez le choix entre deux emplacements possibles pour le nouveau nœud (on note '!' le caractère du nouveau nœud) :

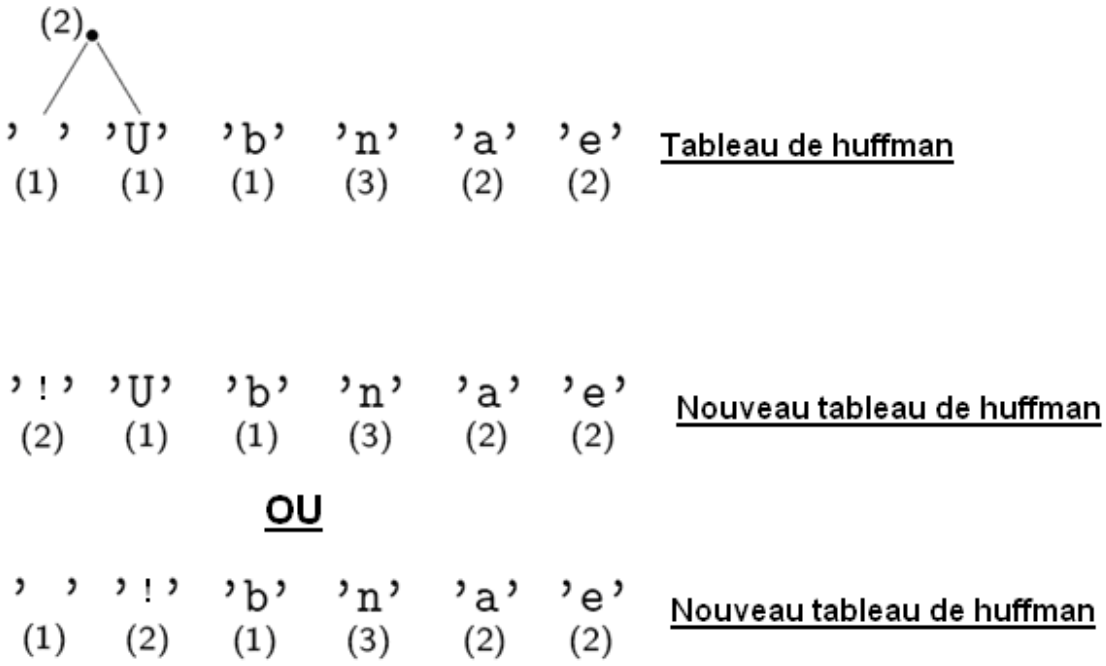


Figure 8 : Création du tableau de Huffman avec un nouveau nœud

La répétition de cette procédure de création de nœud jusqu'au sommet, vous permettra d'obtenir une seule et unique référence (pointeur sur structure) qui est celle du haut de l'arbre binaire, et qui possède dans les champs "droite" et "gauche" les références des deux nœuds suivant (qui eux même possèdent deux références des deux nœuds suivants, etc…)

Cette unique référence qui représente la racine de l'arbre sera stockée dans la variable suivante :

```
struct noeud * racine;
```

2.3.6 Parcours de l'arbre

Notre arbre est maintenant créé. Il ne nous reste qu'une seule référence qui est la racine de l'arbre. Bien entendu, les structures des nœuds et des feuilles existent toujours en mémoire, mais nous n'y avons pas directement accès. Il faut pour cela parcourir l'arbre en repartant toujours de la racine, puis aller à gauche et à droite jusqu'à ce que nous trouvions une feuille. Le code peut être le suivant. Il s'agit d'une fonction récursive, c'est-à-dire qu'elle s'appelle elle-même.

```
void parcourirArbre(struct noeud* ptrNoeud){
    if(ptrNoeud ->droite==NULL && ptrNoeud ->gauche==NULL){
      printf("Je suis une feuille\r\n");
    }
    else{
      printf("Je suis un nœud\r\n");
      parcourirArbre(ptrNoeud ->droite);    //      On      va      a
droite
      parcourirArbre(ptrNoeud ->gauche);    // On va a gauche
      }
}
```

Si nous souhaitons partir de la racine, alors il faut appeler la fonction :

```
parcoursArbre(racine);
```

- Valider cette fonction en notant tous les caractères et les occurrences présents dans votre texte.
- Lorsque cette fonction est validée, vous pouvez libérer l'espace utilisé pour la création de l'arbre de Huffman : *free()*.

2.3.7 Création du code

En parcourant l'arbre binaire depuis le haut de l'arbre, nous allons créer les codes comme nous l'avons exposé à la Figure 5. La méthode pour le parcours de l'arbre de la racine jusqu'au feuille sera structurée de la même façon que la fonction parcourirArbre() sauf qu'à chaque fois, on construit le code des caractères en injectant un '1' ou un '0' dans le code.

```
void creerCode(struct noeud* ptrNoeud,uint32_t code,uint32_t taille){

if(ptrNoeud ->droite==NULL && ptrNoeud ->gauche==NULL){
   ptrNoeud ->tailleCode=taille;
   ptrNoeud ->code=code;
   printf("%c \t code : %d \t taille :%d \r\n", ptrNoeud ->c, ptrNoeud
->code, ptrNoeud->tailleCode);
}
else{
   //On va a droite (on injecte un 0 à droite dans le code)
   creerCode(ptrNoeud ->droite,code<<1,taille+1);
   // On va a gauche (On injecte un 1 à droite)
   creerCode(ptrNoeud ->gauche,(code<<1)+1,taille+1);
         }
}
```

- ptrNoeud: pointeur vers un nœud ou une feuille
- code : code du caractère

- taille : nombre de bit du code

➡ Réaliser la création du code de votre arbre binaire

> **Toujours en parcourant l'arbre, vous pouvez réaliser d'autres fonctions suivants vos besoins.**

2.3.8 Compression du texte

En parcourant l'arbre, on peut retrouver l'adresse de la structure nœud correspondant au caractère qui nous intéresse :

```
struct noeud* getAddress(struct noeud* ptrNoeud, uint8_t caractere);
```

Une fois que nous avons récupéré l'adresse de la structure, on peut alors récupérer :

- le code
- la taille du code

Il nous reste à ajouter le codage de chaque caractère les uns à la suite des autres dans une zone spécifique de la mémoire RAM.

```
uint8_t texteCompress[TAILLE_MAX_COMPRESS];
```

➡ En visualisant la mémoire à l'adresse texteCompress, vérifier que les éléments binaires du code soient bien écrits correctement en mémoire.

2.3.9 Réalisation d'un entête

Le système qui gérera la décompression n'a aucun moyen de connaitre les caractères et leur code Huffman associés. Il est donc indispensable que le fichier compressé possède un entête contenant ces informations.

L'entête comportera :

- La taille de l'entête : 2 octets
- La taille du fichier compressé : 2 octets
- Le nombre total de caractère du fichier d'origine (nbrCaractereTotal) : 2 octets

Puis pour chaque caractère :

- Son code ASCII
- Son code de Huffman
- La taille de son code

➡ Réaliser la fonction de création de l'entête et vérifier sa validité.

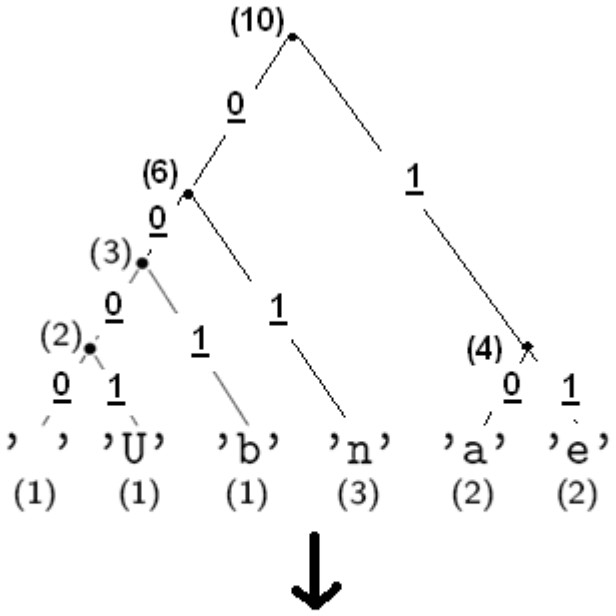
2.3.10 Transmission

La transmission se fera par l'UART1 du microcontrôleur. La transmission devra permettre le transfert du fichier compressé (avec l'entête).

Dans un premier temps, pour s'affranchir d'un deuxième microcontrôleur, on peut reboucler UART1 TX et UART1 RX pour s'envoyer le code à soit même.

2.3.11 Décompression

Le travail que nous avons effectué au début de l'algorithme était :



Caractère	' '	'U'	'a'	'b'	'e'	'n'
Fréquence	1	1	2	1	2	3
Code	0000	0001	10	001	11	01

Il nous faut maintenant réaliser le travail inverse.

Grâce à l'arbre reconstitué, lorsque nous allons lire chacun des bits du fichier compressé, nous descendrons dans les nœuds de l'arbre, jusqu'à ce que nous trouvions une feuille (un nœud avec des valeurs NULL pour les pointeurs droite et gauche).

- Recréer l'arbre de Huffman d'après les informations de l'entête.
- Parcourir votre arbre pour chacun des bits lus et afficher la valeur du caractère décompressé.