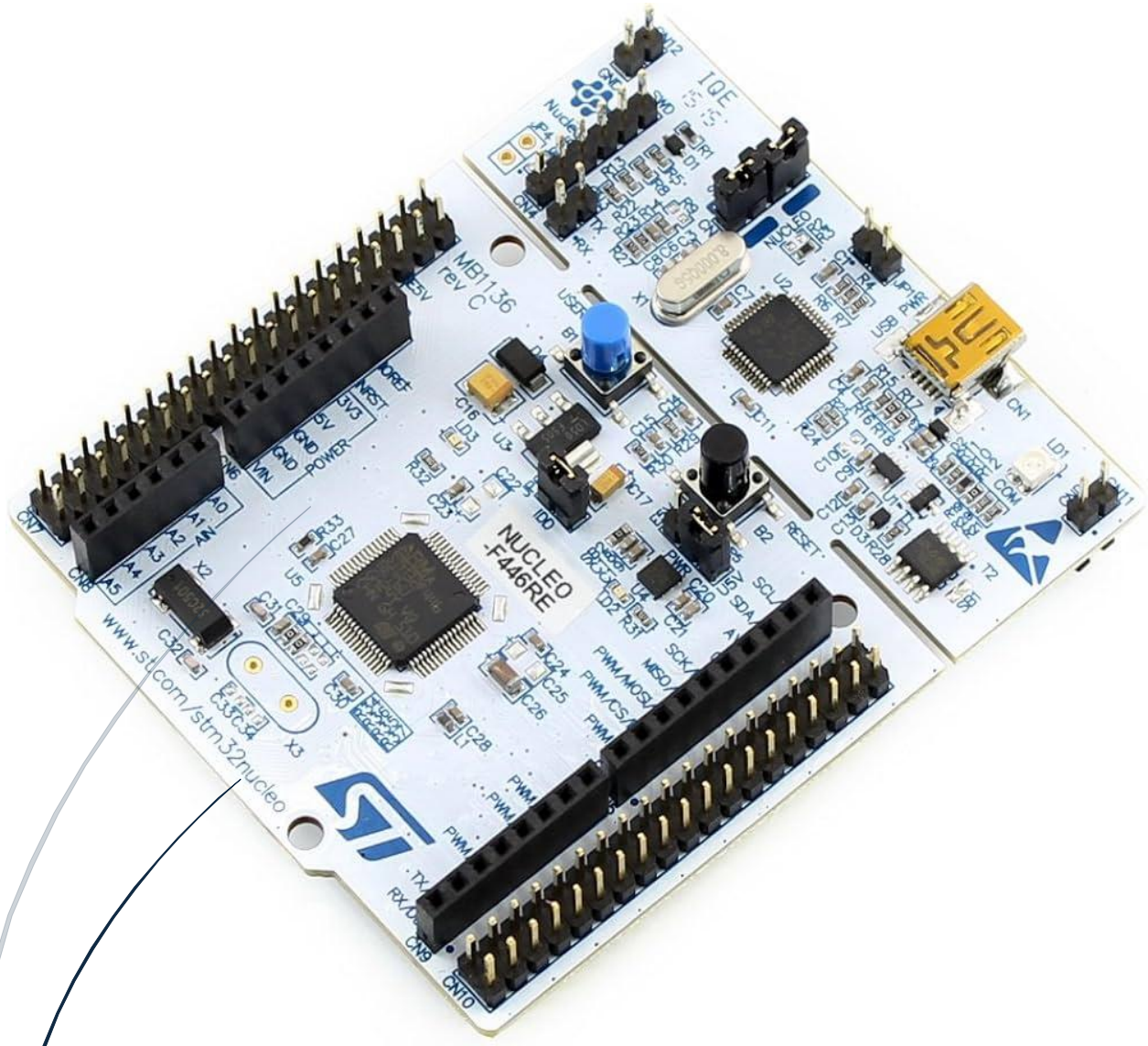
30/01/2026

Projet ES

FreeRTOS



NekoNoTsuki

Table des matières

Présentation du projet 4
1. Architecture matérielle et périphériques utilisés..... 5
1.1. BOM 5
2. Architecture logicielle et organisation des tâches 8
2.1. Gestion des taches 8
2.2. Conception du menu 9
2.3. Fonctions crée 10
3. Conclusion et retour d’expérience 14
3.1. Problèmes..... 14
3.2. Gestion du projet..... 14
3.3. Conclusion 15

Figure 1Bom partie 1 5
Figure 2 Bom partie 2 5
Figure 3 Diagramme fonctionnel 6
Figure 4 Distribution des clocks 7
Figure 5 Graphe d'état des taches 8
Figure 6 Schémas du menu 9

Présentation du projet

Ce projet consiste à concevoir et implémenter une chaîne de traitement audio temps réel sur un microcontrôleur STM32F446RE, reposant sur l'utilisation d'un système d'exploitation temps réel FreeRTOS. Le traitement audio est réalisé via une interface I2S, connectée à un module PMOD audio, permettant l'acquisition et la restitution du signal sonore.

Le cœur du projet repose sur l'implémentation d'un filtrage numérique appliqué au flux audio, avec une architecture logicielle structurée autour de tâches FreeRTOS dédiées au traitement du signal, à la gestion de l'interface utilisateur et à l'affichage. Les paramètres du filtrage, tels que le gain et la fréquence de coupure, peuvent être modifiés dynamiquement par l'utilisateur à l'aide d'un encodeur rotatif.

L'interface utilisateur s'appuie sur un écran OLED, offrant un menu de navigation permettant d'ajuster les paramètres en temps réel et de visualiser les valeurs sélectionnées. Cette interaction directe permet de valider immédiatement l'impact des réglages sur le signal audio traité.

Ce projet vise ainsi à mettre en œuvre une application complète de traitement du signal embarqué, intégrant à la fois des contraintes temps réel, une architecture multitâche et une interface homme-machine adaptée à un système embarqué.

1. Architecture matérielle et périphériques utilisés

1.1. BOM

Afin de réaliser ce projet, il a été nécessaire de disposer d'un ensemble de composants matériels adaptés aux contraintes du traitement audio temps réel sur système embarqué. La liste détaillée du matériel utilisé est présentée sous forme de nomenclature dans les figures **Figure 1** et **Figure 2**, qui regroupent l'ensemble des éléments nécessaires à la mise en œuvre du système, incluant le microcontrôleur, les périphériques audio et les dispositifs d'interaction utilisateur.

Part Number	Quantity	Manufacturer	Distributor	Distributor SKU
NUCLEO-F446RE	1	STMicroelectronics	Mouser Electronics	511-NUCLEO-F446RE
410-379	1	Digilent	RS FR	1840481
MD21605B6W-FPTLWI3	1	Midas	Farnell FR	3565078
PEC11R-4120K-S0018	1	Bourns Electronics	Farnell FR	2663522
PRT-12796	1	Sparkfun Electronics	Mouser Electronics	474-PRT-12796
PSG-JRBN40-MF	1	Pro Signal	Farnell FR	2452752

Figure 1Bom partie 1

Minimum Order	Stock	Unit Price in EUR	Buy from Distributor URL
1	3 332	12,75	https://www.mouser.fr/P
1		34,28	https://fr.rs-
1	76	11,43	https://fr.farnell.com/mid
1	888	2,28	https://fr.farnell.com/bou
1	299	2,36	https://www.mouser.fr/P
1	209	5,19	https://fr.farnell.com/pro-

Figure 2 Bom partie 2

Chaque composant est interconnecté conformément au diagramme fonctionnel présenté en **Figure 3**. Ce diagramme met en évidence le rôle central du microcontrôleur STM32F446RE ainsi que les différentes interfaces de communication mises en œuvre. La chaîne audio repose sur deux interfaces I2S distinctes : une interface dédiée à la réception des données audio et une interface dédiée à leur transmission. Cette séparation permet de gérer de manière claire et efficace le flux audio entrant et sortant. Les deux interfaces I2S sont reliées au module audio PMOD et fonctionnent de manière synchronisée afin d'assurer un traitement continu du signal.

L'encodeur rotatif est connecté à un timer configuré en mode encodeur, ce qui permet une navigation fluide dans les menus et l'ajustement précis des paramètres. L'écran OLED est piloté via un bus I2C, utilisé pour l'affichage des informations et des paramètres sélectionnés par l'utilisateur.

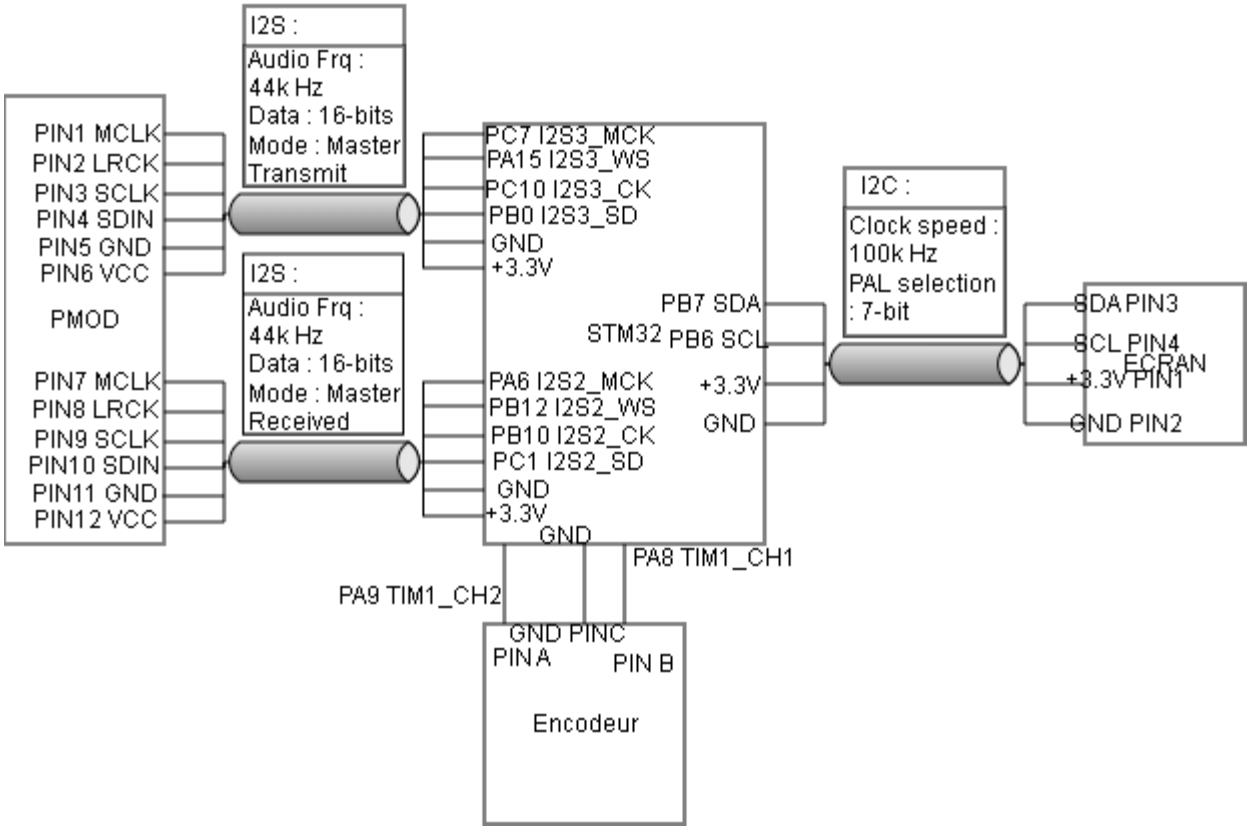


Figure 3 Diagramme fonctionnel

La configuration des périphériques a été définie afin de répondre aux exigences fonctionnelles du projet. Les interfaces I2S sont configurées en mode maître, avec une fréquence d'échantillonnage de 44 kHz et une résolution de 16 bits. L'une des interfaces est utilisée pour la réception des échantillons audio, tandis que l'autre assure la transmission du signal traité. Ce choix permet au microcontrôleur de générer directement les signaux d'horloge nécessaires aux échanges audio et de garantir une synchronisation correcte entre l'acquisition et la restitution. L'utilisation du DMA pour les transferts I2S limite l'intervention du processeur et assure un flux audio continu, compatible avec les contraintes temps réel.

La génération et à la distribution des horloges, présentées en **Figure 4** sont configurer ainsi. Le système utilise une horloge externe comme source principale, à partir de laquelle la PLL génère l'horloge système du microcontrôleur. Les différents bus internes, AHB, APB1 et APB2, sont cadencés à des fréquences adaptées aux périphériques qu'ils desservent. Pour la partie audio, une horloge dédiée est employée afin de fournir aux interfaces I2S une fréquence stable et précise, indispensable au respect de la fréquence d'échantillonnage et à la qualité du signal traité.

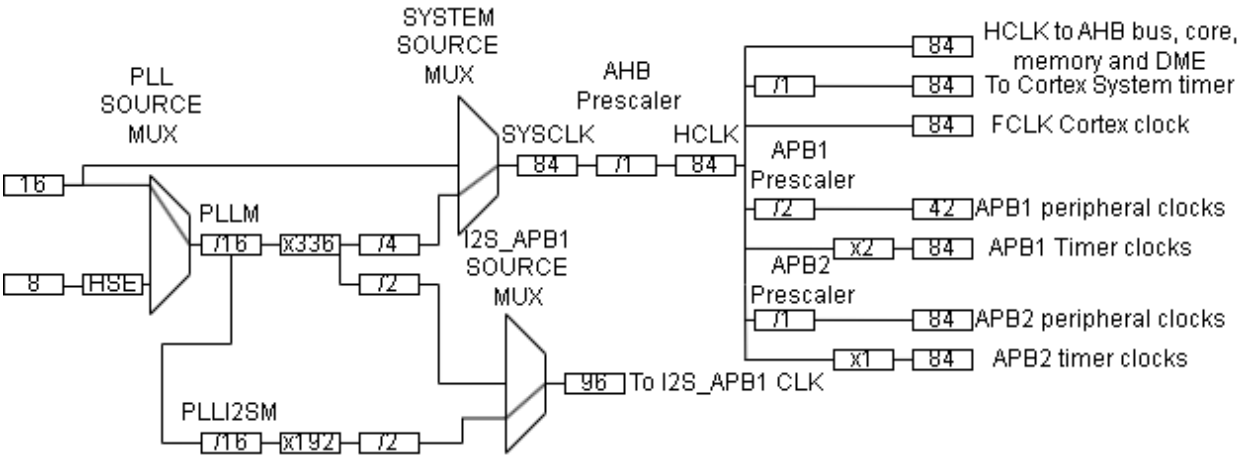


Figure 4 Distribution des clocks

2. Architecture logicielle et organisation des tâches

2.1. Gestion des taches

Dans ce projet, l'architecture logicielle repose sur plusieurs tâches FreeRTOS afin de gérer de manière structurée les différents événements du système, notamment la gestion de l'interface utilisateur, l'affichage et le traitement audio en temps réel. Cette organisation permet de séparer clairement les responsabilités et d'assurer une exécution fiable malgré les contraintes de synchronisation imposées par le flux audio. L'interaction entre les tâches, ainsi que les mécanismes de synchronisation associés, sont présentées en **Figure 5**.

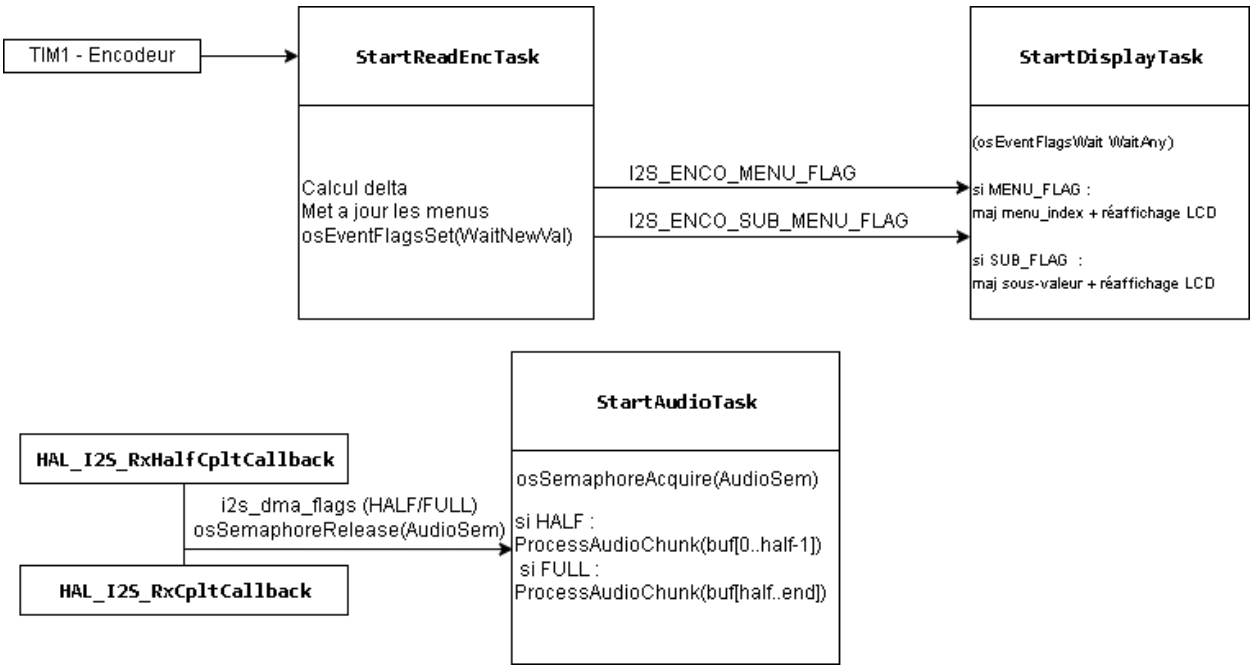


Figure 5 Graphe d'état des taches

Le fonctionnement est autour de trois tâches principales. La tâche StartReadEncTask est dédiée à la lecture de l'encodeur et au calcul du déplacement utilisateur sous forme d'un incrément delta. En fonction du mode de navigation, cette tâche met à jour soit la sélection du menu, soit la valeur associée à un paramètre (par exemple le gain ou la fréquence de coupure), puis en informe l'interface graphique via des flags.

La tâche StartDisplayTask reste bloquée en attente de ces flags et se réveille uniquement lorsqu'une mise à jour est nécessaire. Elle réalise alors l'actualisation de l'écran en distinguant deux cas : un changement de menu ou une modification de valeur.

Enfin, la tâche StartAudioTask assure le traitement du signal audio. Elle est déclenchée par un sémaphore libéré depuis les callbacks de fin de transfert DMA I2S. À chaque réveil, elle traite uniquement la portion de buffer concernée afin de garantir la continuité du flux audio.

2.2. Conception du menu

L'interface utilisateur a été conçue sous la forme d'un menu simple permettant de naviguer entre les paramètres et de modifier leurs valeurs via l'encodeur. La logique de navigation et le rendu attendu à l'écran sont présentés en **Figure 6**. Cette interface permet notamment d'afficher le paramètre sélectionné et d'observer instantanément la valeur appliquée, facilitant ainsi l'ajustement en conditions réelles d'utilisation.

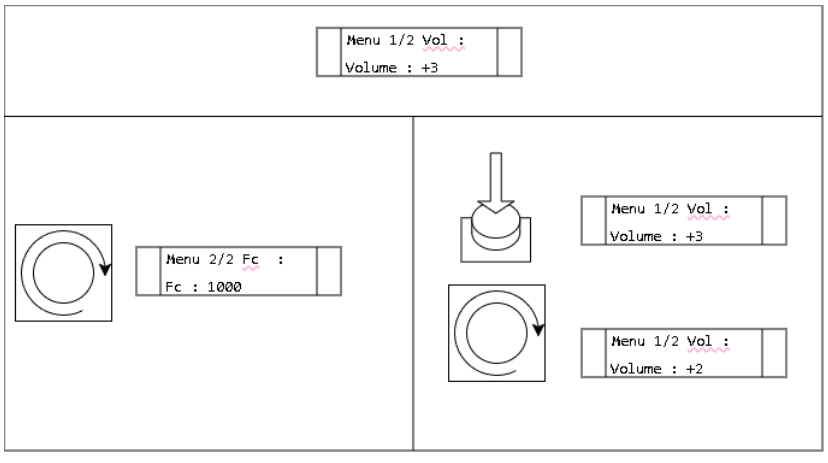


Figure 6 Schémas du menu

2.3. Fonctions crée

```
static inline int16_t EncoderSteps(void)
```

Cette fonction permet de lire la position de l'encodeur rotatif à partir du compteur du timer TIM1 et de la convertir en une valeur signée exploitable par l'application. Le décalage appliqué permet d'adapter la résolution de lecture aux besoins de la navigation dans les menus.

Le test unitaire a consisté à faire tourner l'encodeur dans les deux sens et à vérifier que la valeur retournée évolue de manière cohérente, avec une augmentation ou une diminution selon le sens de rotation, sans variations erratiques.

```
static inline int32_t clamp_int32(int32_t value, int32_t min, int32_t max)
```

Cette fonction permet de borner une valeur entière sur 32 bits entre une limite minimale et une limite maximale définies. Elle est utilisée afin de garantir que les paramètres manipulés par l'utilisateur restent dans des plages compatibles avec le fonctionnement du système.

Le test unitaire a consisté à appliquer la fonction à des valeurs volontairement inférieures à la borne minimale, supérieures à la borne maximale et comprises dans l'intervalle autorisé, puis à vérifier que la valeur retournée correspond bien au comportement attendu dans chaque cas.

```
static void UpdateBiquadIfFreqChanged(void)
```

Cette fonction assure la mise à jour des coefficients du filtre biquad uniquement lorsque la fréquence de coupure a été modifiée. Elle évite ainsi les recalculs inutiles et garantit la cohérence entre le paramètre utilisateur et le traitement audio appliqué.

Le test unitaire a consisté à modifier la fréquence de coupure via l'encodeur et à vérifier que les coefficients du filtre sont recalculés uniquement en cas de changement effectif, tout en confirmant la prise en compte immédiate du nouveau réglage sur le signal audio traité.

```
static void ProcessAudioChunk(int16_t *src, int16_t *dst)
```

Cette fonction assure la mise à jour des coefficients du filtre biquad uniquement lorsque la fréquence de coupure a été modifiée. Elle évite ainsi les recalculs inutiles et garantit la cohérence entre le paramètre utilisateur et le traitement audio appliqué.

Le test unitaire a consisté à modifier la fréquence de coupure via l'encodeur et à vérifier que les coefficients du filtre sont recalculés uniquement en cas de changement effectif, tout en confirmant la prise en compte immédiate du nouveau réglage sur le signal audio traité.

```
static inline int16_t clamp_int16(int32_t value, int16_t min, int16_t max)
```

Cette fonction permet de borner une valeur entière sur 32 bits dans un intervalle défini sur 16 bits, puis de la convertir en int16_t. Elle est utilisée pour sécuriser certaines valeurs avant leur utilisation ou leur affichage.

Le test unitaire a consisté à fournir des valeurs inférieures, supérieures et comprises dans l'intervalle autorisé, puis à vérifier que la valeur retournée respecte correctement les limites définies.

```
size_t MenuDisplay(uint8_t _Id, char* _Buf, size_t _lenghtBuf)
```

Cette fonction permet de générer le texte du menu principal à afficher sur l'écran OLED en fonction de l'identifiant du menu sélectionné. Elle réalise des vérifications élémentaires sur les paramètres d'entrée afin d'éviter tout accès invalide, puis copie le libellé correspondant dans le buffer fourni.

Le test unitaire a consisté à parcourir les différents menus à l'aide de l'encodeur et à vérifier que le texte affiché à l'écran correspond bien au menu sélectionné. Des tests supplémentaires avec des identifiants invalides ont permis de confirmer l'affichage du message d'erreur prévu.

```
size_t SubMenuDisplay(uint8_t _Id_ROW, int16_t _Value, char* _Buf, size_t _lenghtBuf)
```

Cette fonction permet de générer le texte du sous-menu affiché sur la seconde ligne de l'écran OLED en fonction du menu actif et de la valeur associée. Selon l'identifiant du menu, elle formate soit l'affichage du volume, soit celui de la fréquence de coupure. Une vérification des paramètres d'entrée est réalisée afin d'éviter toute utilisation invalide du buffer.

Le test unitaire a consisté à naviguer entre les menus et à modifier les valeurs via l'encodeur, puis à vérifier que le texte affiché sur l'écran correspond correctement au paramètre sélectionné et à sa valeur mise à jour. Des tests avec des identifiants non valides ont également permis de vérifier l'affichage du message d'erreur prévu.

```
int8_t ClampMenuIndex(int32_t _Value, int32_t _Max, int32_t _Min)
```

Cette fonction permet de borner un index de menu entre une valeur minimale et une valeur maximale définies. Elle garantit que l'index utilisé pour la navigation reste valide et évite tout dépassement lors du parcours des menus.

Le test unitaire a consisté à forcer des valeurs inférieures au minimum et supérieures au maximum, puis à vérifier que l'index retourné correspond systématiquement aux bornes définies. La navigation normale a également été testée afin de confirmer le bon comportement lors des déplacements successifs dans le menu.

```
void ApplyGain(const int16_t *_Src, int16_t *_Dis, uint32_t _Size, int32_t _Gain)
```

Cette fonction applique un gain numérique à un buffer audio en entrée afin d'ajuster l'amplitude du signal. Le gain est réalisé par décalage binaire des échantillons, ce qui permet une modification rapide du niveau sonore tout en limitant la charge de calcul. Des vérifications sont effectuées sur les paramètres d'entrée ainsi que sur la plage de gain autorisée, et une saturation est appliquée afin d'éviter tout dépassement de la dynamique des données sur 16 bits.

Le test unitaire a consisté à faire varier la valeur du gain via l'interface utilisateur et à observer l'évolution du niveau sonore en sortie. La continuité du signal ainsi que l'absence de saturation non contrôlée ont été vérifiées lors des changements de gain.

```
void Param_Biq_filter_2nd_Order_Low_pass(uint16_t _Fc, uint16_t _Fs, float32_t* _Buf)
```

Cette fonction calcule les coefficients d'un filtre passe-bas biquad du second ordre à partir d'une fréquence de coupure et d'une fréquence d'échantillonnage. Les coefficients sont ensuite normalisés et stockés dans un buffer afin d'être utilisés par l'algorithme de filtrage, avec une prise en compte du format attendu selon le mode de filtrage sélectionné.

Le test unitaire a consisté à faire varier la fréquence de coupure via l'interface utilisateur, puis à vérifier la cohérence des coefficients générés (affichage sur la liaison série) ainsi que l'effet audible du filtrage sur le signal, avec une atténuation progressive des hautes fréquences lorsque la fréquence de coupure diminue.

```
size_t Apply_Biquad_Filter_DF1(const q15_t* _Src, q15_t* _Dst, uint32_t _Size, const float32_t* _Coeffs, q15_t* _State)
```

Cette fonction applique un filtre biquad du second ordre sur un buffer audio en format Q15 selon la structure Direct Form I. Les coefficients calculés en flottant sont convertis en Q15, puis le filtrage est réalisé échantillon par échantillon en conservant l'état interne du filtre afin d'assurer la continuité entre deux appels. Une saturation est appliquée pour garantir que la sortie reste dans la dynamique 16 bits.

Le test unitaire a consisté à activer ce mode de filtrage, à faire varier la fréquence de coupure, puis à vérifier la stabilité du traitement (absence de décrochements) ainsi que l'effet audible du filtre sur le signal. L'état du filtre a également été validé en vérifiant que le comportement reste cohérent lors du traitement successif de plusieurs blocs audio.

```
size_t Apply_Biquad_Filter_DF2T(const int16_t* _Src, int16_t* _Dst, uint32_t _Size, const float32_t* _Coeffs, float32_t* _State)
```

Cette fonction applique un filtre biquad du second ordre selon la structure Direct Form II transposée sur un buffer audio. Le calcul est réalisé en virgule flottante, avec conservation de l'état interne du filtre entre deux appels afin d'assurer la continuité du traitement. La sortie est ensuite convertie en format entier 16 bits avec saturation pour éviter tout dépassement de plage.

Le test unitaire a consisté à activer ce mode de filtrage, à faire varier la fréquence de coupure et à vérifier le comportement du signal traité à l'écoute. La stabilité du filtre et l'absence d'artefacts audibles lors du traitement de blocs audio successifs ont permis de valider le bon fonctionnement de cette implémentation.

3. Conclusion et retour d'expérience

3.1. Problèmes

Un premier problème est apparu lors de l'utilisation de fonctions de type SetFlag depuis une interruption. Bien que la documentation indique la possibilité d'utiliser ce mécanisme dans un contexte d'IRQ, son fonctionnement n'a pas été validé dans le cadre de ce projet. Le problème a donc été contourné par la mise en place d'un sémaphore, solution qui s'est révélée fonctionnelle et plus adaptée à la synchronisation entre interruptions et tâches.

D'autres difficultés ont également été rencontrées lors de la mise en place des différentes fonctions de filtrage. En particulier, le réglage des paramètres du filtre a demandé de nombreux ajustements afin d'obtenir un comportement cohérent et stable. Ces phases de mise au point ont mis en évidence l'importance d'une bonne compréhension des paramètres théoriques du filtrage et de leur impact direct sur le signal traité.

3.2. Gestion du projet

Concernant la gestion du projet, celui-ci ayant été réalisé seul, l'organisation est restée volontairement simple. La majeure partie du développement a été effectuée durant le mois de décembre, période durant laquelle l'architecture matérielle et logicielle a été mise en place. Le mois de janvier a ensuite été principalement consacré aux phases de test, de correction et d'amélioration du fonctionnement global. En termes d'avancement, le projet peut être considéré comme abouti d'un point de vue fonctionnel. Néanmoins, des améliorations restent possibles, tant sur le plan théorique que pratique, notamment en termes d'optimisation et d'extension des fonctionnalités. Ces évolutions n'ont pas pu être explorées davantage par manque de temps et de connaissances plus approfondies.

3.3. Conclusion

Pour conclure, ce projet s'est révélé particulièrement enrichissant. Il a permis de consolider de manière concrète la prise en main de FreeRTOS, notamment à travers la mise en œuvre d'une architecture multitâche, la gestion des interruptions et l'utilisation de mécanismes de synchronisation adaptés aux contraintes temps réel. Le projet a également permis de mieux appréhender les problématiques liées au traitement du signal audio en temps réel, en mettant en évidence l'importance de la gestion des buffers, de la stabilité des horloges et du choix des paramètres de filtrage.

Certaines difficultés rencontrées au cours du développement ont mis en lumière le besoin d'une compréhension approfondie des fonctionnalités internes du microcontrôleur et de ses périphériques. À ce titre, un apport d'informations complémentaires sur certains mécanismes spécifiques du STM32 aurait permis de gagner en efficacité et d'explorer des solutions alternatives plus rapidement. Par ailleurs, la définition d'objectifs plus diversifiés que le simple filtrage audio aurait offert l'opportunité d'exploiter d'autres aspects de la plateforme STM32, tels que des traitements plus complexes ou des interactions supplémentaires avec le système.

Malgré ces limites, le projet a permis d'aboutir à une application fonctionnelle et cohérente, intégrant à la fois des aspects matériels, logiciels et temps réel. Il constitue ainsi une base solide pour de futures améliorations et a offert une vision globale et concrète des étapes nécessaires à la conception d'un système embarqué complet.

知識

知識（**chishiki**） désigne la connaissance acquise par l’apprentissage, l’expérience et la compréhension.

Ce terme exprime à la fois l’accumulation de savoirs et la capacité à les utiliser de manière réfléchie. Il rappelle que la connaissance n’est pas seulement théorique, mais qu’elle se construit et s’enrichit par la pratique et l’analyse.