Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Filière Systèmes industriels

Orientation Infotronics

Travail de bachelor
Diplôme 2016

William Espinosa

MIRES

Professeur
Alexandra Andersson

Expert
Hans-Peter Biner

Date de la remise du rapport
15.07.2016

Table of Contents

1 Introduction

Dans le monde de la recherche scientifique et médicale, il est aujourd'hui courant d'utiliser des rats et des souris pour mener diverses expériences.

Pour recueillir différentes données sur ces animaux, ceux-ci sont reliés en permanence à des câbles, ce qui diminue grandement le confort de l'animal, ainsi que celui du personnel lorsqu'ils doivent déplacer les sujets.

Ainsi, un premier projet a été réalisé. Celui-ci se penchait sur la réalisation d'une carte miniature ayant pour but d'être implantée dans les rats. Cette carte devait permettre l'envoi de données par Bluetooth® Low Energy et est alimentée par de petites batteries. Néanmoins, l'utilisation de batteries présente des désavantages majeurs : Le débit est grandement limité et la durée de vie totale du système est réduite.

De ce constat est né le projet actuel.

Celui-ci a pour but de supprimer totalement le système d'alimentation par batterie et de le remplacer par une alimentation sans fil.

La transmission d'énergie sans fil, réalisée grâce à un couplage inductif résonant, va permettre d'obtenir un débit de données plus important et sans limite d'utilisation dans le temps.

Ce type d'alimentation est très sensible à l'environnement et doit donc être régulé en permanence pour garantir que le couplage reste toujours optimal.

Le système complet sera donc composé d'une boucle d'alimentation régulée de manière motorisée et dimensionnée pour pouvoir contenir une cage de laboratoire standard, et d'un circuit de miniaturisé qui sera implanté dans l'animal.

2 Cahier des charges

Le système doit correspondre aux différents critères suivants :

- Fonctionnement dans la bande de fréquence ISM 26.957-27.283 Mhz
- Communication Bluetooth Low Energy entre la carte et le système de régulation
- Régulation motorisée du couplage inductif résonant afin d'avoir toujours la transmission maximale d'énergie
- Régulation de la fréquence de fonctionnement

3 Bref rappel du travail de semestre

Avant le travail de diplôme, ce projet a été l'objet d'un travail de semestre.

Celui-ci avait pour but de tester le fonctionnement du système d'alimentation afin d'en connaître les limites.

Le développement réalisé ne comprenait alors pas de module Bluetooth, mais simulait celui-ci à l'aide d'une charge statique et d'une charge pulsée afin de s'approcher de la consommation d'un module Bluetooth Low Energy.

L'objectif était ainsi de pouvoir consommer depuis la carte un courant statique de 300 µA et un courant pulsé de 8 mA par pulsation, avec une durée et un intervalle variable.

Les résultats obtenus montraient que l'on pouvait obtenir dans ces conditions un courant moyen de plus de 800 µA à 2.8 V, soit environ 2.2 mW. L'alimentation n'était alors poussée qu'à un peu plus de la moitié de ses capacités.

Ces résultats ont permis de savoir quelle consommation viser lors de la réalisation du travail de diplôme.

4 Principe de fonctionnement

4.1 Schéma de principe

Le système complet peut être décomposé en 3 blocs :

- Boucle de transmission d'énergie
- Carte de mesure
- Carte de régulation du couplage à l'aide d'un moteur pas à pas

Le schéma de la Figure 1 ci-dessous présente rapidement le système.

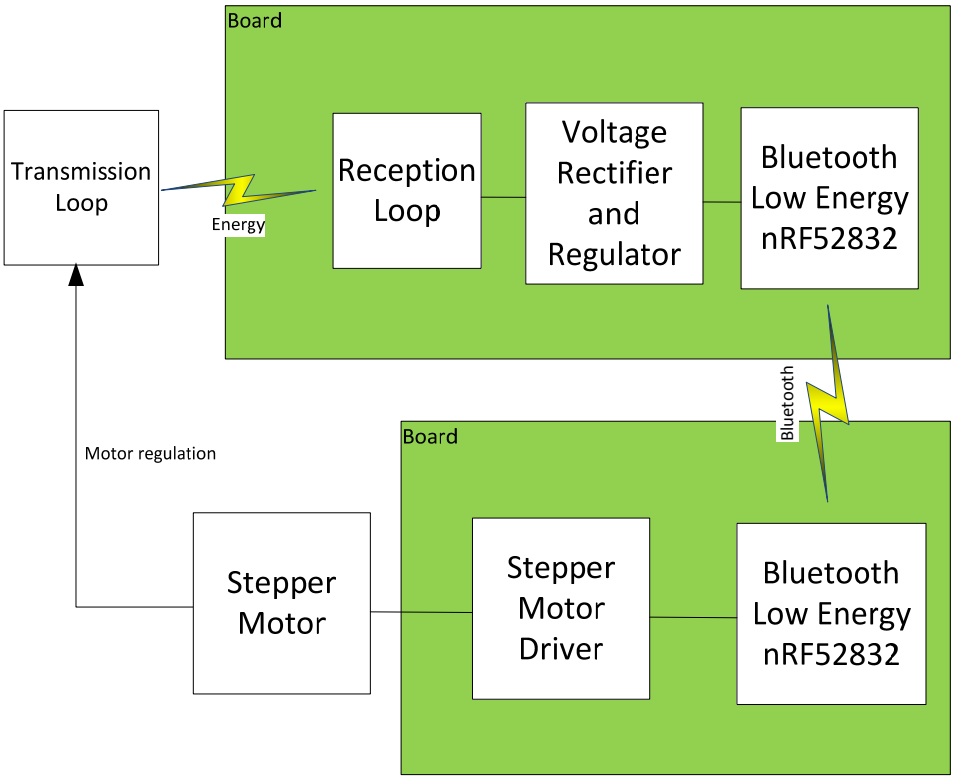


Figure 1 : Schéma de principe du système

Le système fonctionne par couplage inductif résonant : Les boucles d'alimentation et de réception résonnent à la même fréquence, ce qui permet d'obtenir un rendement sur la distance bien meilleur que celui d'un système non-résonant.

4.2 Transmission d'énergie

La partie d'alimentation est composée d'une boucle de cuivre de 39 cm de côté, terminée par un condensateur variable et munie d'une vis de réglage.

Cette boucle est conçue pour avoir une fréquence de résonnance proche de la fréquence centrale de 27.12 Mhz autorisée par les bandes ISM.

Le condensateur permet ainsi de modifier la fréquence de résonnance de la boucle afin de se coordonner avec la boucle de réception qui possède une fréquence de résonnance fixe. Celui-ci est un condensateur à air APC-100 de Hammarlund ayant une capacité de 5.5 à 100 pF.

Le but de la vis de réglage est le même que celui du condensateur, mais celle-ci permet des réglages plus fins et précis.

La valeur du condensateur peut être modifiée par le moteur pas à pas de régulation afin d'assurer le meilleur couplage possible.

4.3 Réception d'énergie

La partie de réception est composée d'une bobine réalisée à partir d'un fil de cuivre, d'une dimension proche de celle de la carte finale, 20 x 6 mm. Des condensateurs en série fixent la fréquence de résonance dans les valeurs voulues.

Pour les tests, cette partie est réalisée sur son propre PCB au lieu d'être inclue à la carte principale.

4.4 Carte de mesure

La carte de mesure a pour tâche de fournir par Bluetooth Low Energy les mesures nécessaires à la régulation du couplage résonant.

Pour cela, elle convertit d'abord le courant alternatif provenant de la bobine de réception en courant continu par un pont de diodes Schottky.

Ce courant continu est ensuite régulé par un régulateur de type LDO qui va pouvoir fournir les 3 volts voulus avec une très faible chute de tension.

Enfin, un System On Chip muni d'un module Bluetooth Low Energy permettra d'effectuer les mesures et de les envoyer à la carte de régulation.

4.5 Carte de régulation

La carte de régulation a pour but de contrôler la valeur du condensateur de réglage pour garantir un couplage optimal.

Le condensateur est contrôlé à l'aide d'un moteur pas-à-pas dirigé par la carte de régulation. Pour cela, un driver supplémentaire est nécessaire afin d'alimenter correctement les bobines du moteur pour les déplacements voulus.

Le tout est dirigé grâce aux valeurs reçues par Bluetooth depuis la carte de mesure.

4.6 Régulation de fréquence

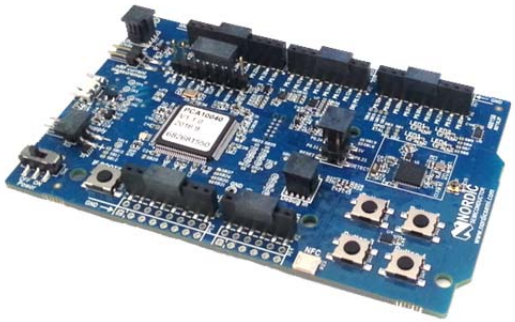
Bien que les valeurs de la bobine de réception et des condensateurs soient connues, on ne peut pas connaître avec précision la fréquence de résonance. En effet, la carte a pour but d'être implantée dans une souris de laboratoire, la fréquence de résonnance va varier légèrement dans cet environnement.

Ainsi, il est nécessaire de pouvoir varier la fréquence d'entrée afin de s'en approcher au maximum. Pour cela, le générateur de fréquence doit être contrôlé par son port GPIB.

Cette partie du cahier des charges n'a pas pu être réalisée par faute de temps.

5 Implémentation

5.1 nRF52 Development Kit



Pour ce projet, le développement a été réalisé à l'aide de cartes nRF52 Development Kit, développée par Nordic Semiconductor.

Ce kit de développement permet la réalisation rapide d'applications spécialement conçue pour le BLE, à l'aide du SoC nRF52832 muni d'un ARM Cortex-M4F. Il met notamment à disposition toutes les entrées/sorties et interfaces, ainsi que 4 LEDs et 4 boutons.

Deux de ces cartes sont utilisées pour le projet.

La première est alimentée à l'intérieur de la boucle et mesure les valeurs de la tension avant régulation avant de l'envoyer par Bluetooth.

La deuxième, située à l'extérieur de la boucle, va se connecter à la première et récupérer les mesures effectuées. Grâce à ces mesures, elle va ensuite contrôler le moteur de régulation du couplage.

5.2 Bluetooth Low Energy (BLE)

5.2.1 Principe

Le BLE est une évolution du Bluetooth classique introduite dans la spécification Bluetooth 4.0.

Cette évolution de la norme propose une réduction considérable de la consommation d'énergie avec une puissance maximale de 10 mW, ce qui en fait un candidat idéal pour toutes les applications embarquées.

L'architecture du protocole BLE se divise en 7 couches comme présenté dans la Figure 2.

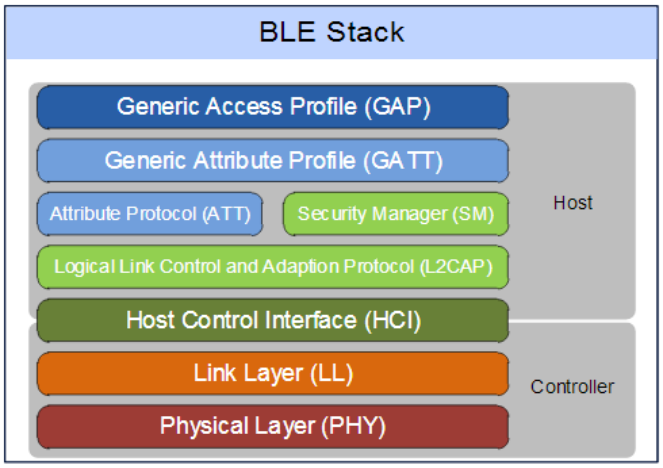


Figure 2 : Architecture du stack BLE [https://www.bluetooth.org/tpg/RefNotes/BLE_Stack_RIN.pdf]

Les couches les plus importantes à configurer lors du développement sont celles situées le plus haut : GAP et GATT.

5.2.1.1 GAP

La couche GAP définit toutes les procédures par lesquelles un appareil peut se rendre visible aux autres et comment ils peuvent communiquer entre eux.

Les options de visibilité comprennent notamment :

- Nom de l'appareil
- Type d'appareil (*Appearance*)
- Paramètres de connexion : Intervalles de connexion, latence, etc.

Au niveau de la communication, l'appareil peut choisir de fonctionner selon deux modes :

- Broadcaster - Observer : Communication sans connexion
- Peripheral - Central : Communication nécessitant une connexion entre les deux appareils pour transférer des données

5.2.1.2 GATT

La couche GATT défini les méthodes d'échange de données une fois que la connexion est établie par GAP.

Celle-ci se décompose en une hiérarchie en 4 niveaux. Le plus haut niveau est le *Profile* qui comprend un ou plusieurs *Services* nécessaires pour un cas d'utilisation donné. Chaque *Service* regroupe des *Characteristics.* A son tour, une *Characteristic* contient des *Descriptors* et une *Value.*

Cette hiérarchie est illustrée à la Figure 3.

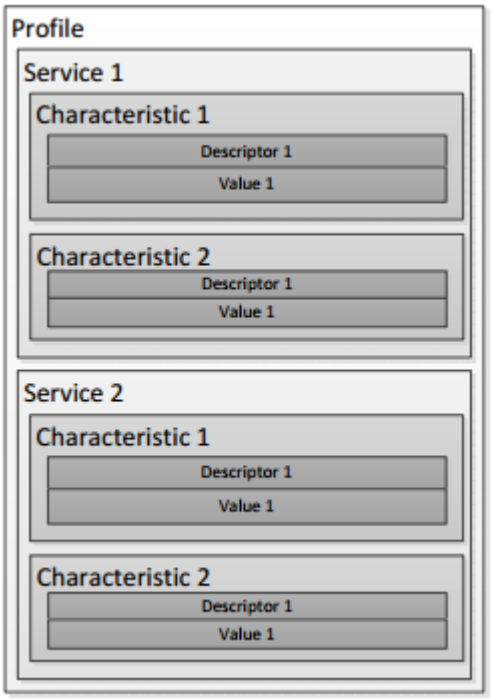


Figure 3 : Structure hiérarchique du BLE

Une liste de services officiels a été définie dans la spécification du Bluetooth 4.0, mais il reste possible de développer ses propres services si ceux existants ne correspondent pas à l'utilisation voulue. Un exemple de *Service* est le « Battery Service » qui permet de donner l'état d'une batterie et qui est composé d'une seule *Characteristic « Battery Level »* qui contient la valeur de la batterie, accessible en lecture.

GATT accepte des requêtes de lecture, d'écriture ou encore de notification et y répond en retournant la valeur voulue ou en modifiant celle-ci dans le cas d'une écriture.

Chaque *Service* est défini par un UUID unique de 128-bit composé d'un UUID de base (14 bytes) et d'un UUID de service (2 bytes), comme indiqué dans la Figure 4.

Figure 4 : BLE 128-bit UUID

L'UUID de base indiqué dans cette figure n'est utilisable que pour les services officiels et doit être changé dans le cas d'un service personnalisé.

5.2.2 Service VMSR

Comme signalé au point 5.2.1.2, différents services ont été définis dans le Bluetooth 4.0, néanmoins, aucun d'entre eux ne correspond réellement à une mesure de tension. Un service personnalisé doit donc être créé, sous le nom de Voltage Measurement Service (VMSR).

Ce service est composé d'une seule *Characteristic*, « Voltage Value » qui donne la valeur actuelle de la tension mesurée. Il est possible de lire cette *Characteristic* ou de recevoir les notifications directement.

Comme c'est un service personnalisé, il est nécessaire de lui attribuer un UUID de base. Celui-ci est généré par l'outil « NRFgo Studio » de Nordic Semiconductor : C96Cxxxx-E04D-43A7-AF24-0310CB6CB855.

L'UUID de service peut être défini de façon arbitraire, la valeur 0x2000 a ainsi été choisie pour le *Service* et la valeur 0x2001 pour la *Characteristic*.

Type	UUID
Base UUID	C96Cxxxx-E04D-43A7-AF24-0310CB6CB855
VMSR Service UUID	2000
Voltage Value Characteristic UUID	2001

Une explication de la programmation de ce service est disponible au point 5.3.1.1.

5.3 Programmation

Les deux cartes utilisées dans ce projet contiennent chacun un code qui lui est propre et qui sera expliqué dans les points suivants.

Le code est disponible en **Error! Reference source not found.** à **Error! Reference source not found.**.

5.3.1 Carte de mesure

La carte de mesure est celle qui est alimentée par le système d'alimentation sans fil.

Cette carte va être celle qui lance l'advertising BLE et qui va agir en tant que serveur (ou « Peripheral »). Elle va mesurer la valeur de la tension avant régulation et la transmettre à l'autre carte par Bluetooth Low Energy.

Le code est réparti en 2 fichiers : main.c et vmsr.c

5.3.1.1 vmsr.c

Ce fichier contient les fonctions relatives au service BLE personnalisé « Voltage Measurement Service » créé pour les besoin du projet.

Il est composé de trois fonctions principales :

- ble_vmsr_init :
  Fonction d'initialisation qui met en place la structure du service.
  Elle ajoute l'UUID du *Service* au SoftDevice (protocole sans fil de Nordic Semiconductor)

- voltage_value_char_add :
  Configure la *Characteristic* « Voltage Value » et l'ajoute au SoftDevice en le liant au service VMSR
- ble_vmsr_voltage_update :
  Cette fonction est appelée par le programme principal lorsqu'une nouvelle mesure de tension (float) est disponible.
  Si l'appareil est connecté par Bluetooth, elle envoie la mesure à l'autre appareil à l'aide de la fonction sd_ble_gatts_hvx

5.3.1.2 main.c

Programme principal.

Fonctions principales :

- main :
  Initialise tous les modules nécessaires au fonctionnement du programme, comme les timers ou le BLE, et démarre l'advertising
- adc_configure :
  Configure le convertisseur AD pour réaliser les conversions de la tension non-régulée.
- Saadc_event_handler :
  Fonction de callback du convertisseur AD qui met en forme le résultat de la conversion et démarre l'envoi par Bluetooth.

5.3.1.2.1 Conversion AD

Pour les mesures de tension, il convient de réaliser une conversion AD sur la valeur de tension non-régulée.

Pour cela, le NRF52832 est muni d'un ADC à approximation successive (*Successive Approximation ADC – SAADC).*

Le temps total d'une conversion AD dépend du temps de conversion et du temps d'acquisition :

- Le temps de conversion est toujours en dessous de 2 µs
- Le temps d'acquisition augmente avec la résistance d'entrée selon le Tableau 1 suivant :

Résistance [kΩ]	Acquisition time [µs]
<= 10	3
<= 40	5
<= 100	10
<= 200	15
<= 400	20
<= 800	40

Tableau 1 : Temps d'acquisition selon la résistance de source

Ce SAADC dispose de base d'une résolution de 12 bits (9 bits effectifs) qu'il est possible de monter à 14 bits (10.5 bits effectifs) en utilisant de l'oversampling. L'oversampling va également permettre de

réduire au maximum le bruit lors de la conversion en réalisant une moyenne sur plusieurs valeurs, pour un résultat plus précis.

5.3.1.2.1.1 Configuration

Le driver SAADC est initialisé avec la configuration suivante :

- Utilisation de l'input analogique AIN0 (Pin P0.02)
- Gain d'entrée de ¼ pour correspondre à la référence interne de 0.6 V
- Channel 0

Comme indiqué au point 5.3.1.2.1, le temps d'acquisition d'un *sample* dépend de la résistance d'entrée.

Pour minimiser le courant consommé par la conversion, une grande résistance doit être choisie. Néanmoins, plus la résistance est grande, plus le temps d'acquisition sera élevé et consommera du courant.

Le bon compromis doit donc être choisi. Des mesures ont été réalisées pour chaque temps d'acquisition et sont présentées dans la Figure 5 :

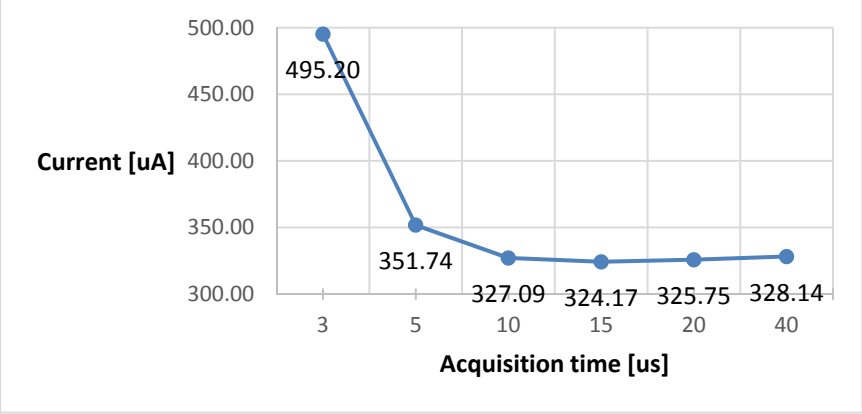


Figure 5 : Courant en fonction du temps d'acquisition

On constate un minimum de courant consommé pour 15 µs, mais les valeurs restent très proches à partir de 10 µs.

Un temps de 40 µs a finalement été choisi car il permet une utilisation avec n'importe quelle résistance de moins de 800 kΩ.

5.3.1.2.1.2 Erreur driver SAADC

Dans le SDK 11.0.0 de Nordic Semiconductor, une erreur existe dans le driver SAADC. Lorsqu'il est initialisé, il initialise également le module EasyDMA qui gère l'accès direct à la mémoire, mais il ne le désactive pas lorsqu'aucune conversion n'est en cours. Ainsi, on constate une consommation constante de près de 2 mA, ce qui est au-delà des capacités de notre système d'alimentation.

Pour corriger ce problème, le driver SAADC doit être initialisé avant chaque conversion, et désactivé directement après.

5.3.1.2.2 Bluetooth

5.3.1.2.2.1 Advertising

Pour être visible aux autres appareils, la carte de mesure va envoyer des messages d'*advertising* de manière répétée.

Ces messages contiennent entre autres le nom de l'appareil et l'UUID de chaque service disponible.

Lors d'un envoi de paquets Bluetooth, des pics de courants jusqu'à 12 mA ont lieu. La Figure 6 ci-dessous montre ce phénomène.

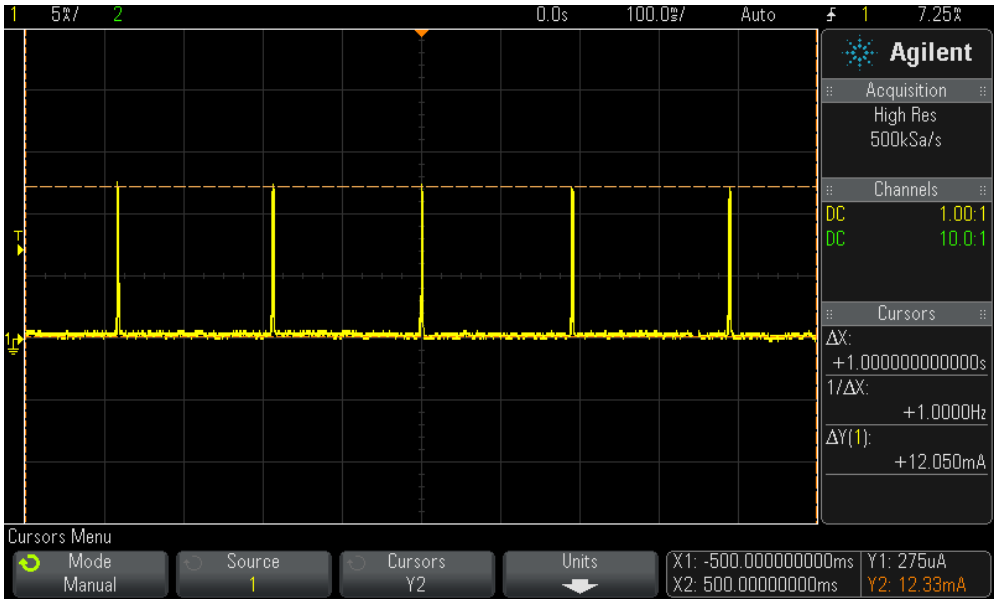


Figure 6 : Fonctionnement de l'advertising BLE dans le temps

Ces pics durent environ 3 ms ce qui donne un courant moyen de 7 mA.

Selon la norme Bluetooth 4.0, l'intervalle entre deux de ces pics est défini entre 20 ms et 10 s. Pour limiter le courant consommé, un intervalle de 500 ms a été choisi. Celui-ci permet une connexion rapide à l'appareil, tout en ne consommant que 42 µA en moyenne.

Une fois une connexion établie, l'advertising s'arrête et ne reprendra qu'à la déconnexion.

5.3.1.2.2.2 Transmission

Une fois que deux appareils sont connectés ensemble, la transmission de paquets de données peut s'effectuer.

Chaque paquet transmis contient la valeur de tension récupérée du convertisseur AD.

Les données sont envoyées à chaque fois qu'une valeur est récupérée de la conversion AD, mais uniquement si la nouvelle valeur est différente de la précédente.

5.3.2 Carte de régulation moteur

Cette deuxième carte va agir en tant que client (ou « Central »). Elle va recevoir les valeurs de tension par Bluetooth et fera tourner le moteur en conséquence pour réguler le couplage.

Tout comme la carte de mesure, deux fichiers contiennent le code utilisé : main.c et vmsr_c.c

5.3.2.1 vmsr_c.c

Comme au point 5.3.1.1, ce fichier contient toutes les fonctions relatives au service « Voltage Measurement Service ». La différence principale est que vmsr.c gère l'envoi de paquet alors que vmsr_c.c gère la réception.

Les fonctions principales sont :

- ble_vmsr_c_init :
  Initialise la structure du service et ajoute son UUID au SoftDevice.
- ble_vmsr_on_db_disc_evt :
  Lorsque le service VMSR est découvert sur le serveur, sauve les informations concernant le service.
- on_hvx :
  Chaque fois qu'une notification est reçue par Bluetooth, cette fonction vérifie si c'est une tension qui est reçue et la sauvegarde.

5.3.2.2 main.c

Programme principal.

Fonctions principales :

- main :
  Initialise tous les modules nécessaires comme les timers ou la régulation, et démarre le scan Bluetooth pour découvrir la carte de mesure
- regulation_init :
  Initialise les entrées/sorties pour le contrôle du moteur
- motor_regulation :
  Régulation de la boucle d'alimentation en contrôlant le moteur
- scan_start :
  Configure et démarre le scan Bluetooth afin de trouver un appareil qui a un advertising en cours.

5.4 Circuit imprimé

La carte réalisée reçoit le courant alternatif à haute fréquence de la boucle de réception.

Son but est de redresser ce courant et de le réguler à 3 volts, ce qui permettra d'alimenter le nRF52832.

Par manque de temps, le nRF52832 n'a pas été implémenté sur le circuit imprimé. A la place, le Development Kit nRF52 a été utilisé directement.

La Figure 7 suivante présente le circuit réalisé :

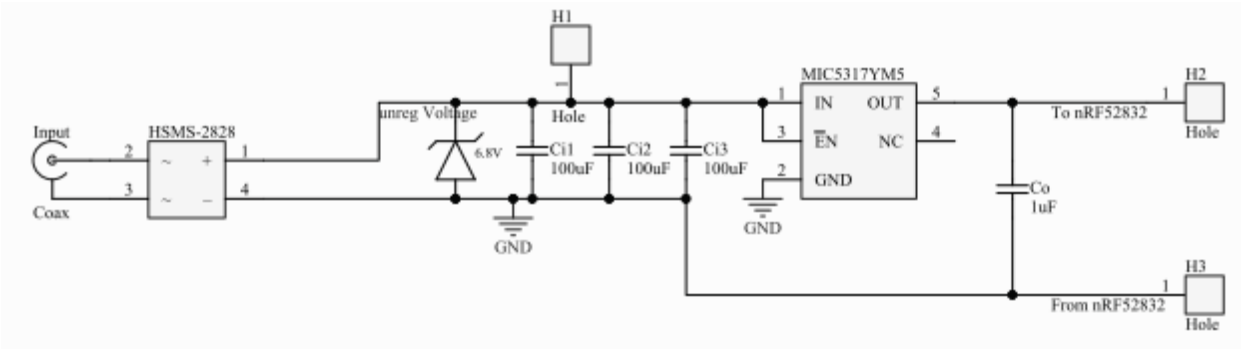


Figure 7 : Circuit redresseur et d'alimentation

Chaque partie va être expliquée plus en détail dans les points qui suivent.

La bill of materials du circuit est disponible en **Error! Reference source not found.** et le PCB en **Error! Reference source not found.**.

Chaque partie est détaillée dans les points suivants.

5.4.1 Redresseur

Le circuit reçoit un courant alternatif depuis la bobine de réception au travers d'un connecteur SMA.

Afin de convertir le courant alternatif en courant continu, le pont de diode HSMS-2828 de chez Avago Technologies est utilisé.

Comme le système fonctionne dans des hautes fréquences, il est à noter que le HSMS-2828 utilise des diodes Schottky. Celles-ci ont des avantages sur les diodes à jonction p-n classiques :

- Bien plus rapides et donc plus adaptées pour une utilisation HF
- *Forward voltage* de seulement 0.34 V, ce qui donne une perte totale de seulement 0.7 V, donc très adapté à l'utilisation à basse tension

5.4.2 Régulateur

L'objectif étant d'obtenir une tension de 3 V sur les charges, un régulateur de type LDO est utilisé pour garantir cette valeur.

Ce régulateur est un MIC5317-3.0, conçu pour fonctionner jusqu'à 6 V. Pour le circuit de test, le format SOT23 est utilisé, mais une version 1mm x 1mm TDFN existe pour la version finale du circuit.

Une diode Zener de 6.8 V protège le pont de diode et l'entrée du régulateur qui ne doit en aucun cas dépasser 7 V sous peine de destruction.

Les trois condensateurs au tantale de 100 µF, pour un total de 300 µF, permettent de stocker l'énergie nécessaire pour alimenter le circuit lorsque le module Bluetooth tire des pics de courant lors de chaque envoi de données.

Un condensateur d'au moins 1 µF doit relier l'entrée et la sortie du régulateur au GND pour garantir sa stabilité. L'entrée est déjà reliée aux condensateurs de 100 µF, mais il est conseillé par le constructeur d'utiliser des condensateurs céramique X7R pour filtrer le bruit à haute fréquence, il est donc nécessaire d'en ajouter un en entrée et en sortie.

Malgré l'ajout de ces condensateurs, la tension en sortie du régulateur a tendance à diminuer lorsque l'amplitude du signal HF augmente, jusqu'à un peu plus de 2.8 V.

5.4.3 nRF52832

Pour ce projet, le nRF52832 n'a pas été implémenté directement sur la carte réalisée par faute de temps.

A la place, le *Development Kit nRF52* a été utilisé directement. Ce kit de développement comprend beaucoup de composants supplémentaires, notamment une PCA10040. Il n'est donc pas possible d'alimenter le tout uniquement avec l'alimentation sans fil.

Une modification a donc dû être réalisée afin d'alimenter uniquement le nRF52832 par l'alimentation sans fil, tout en alimentant le reste par pile ou par USB. Pour cela, le bridge SB9 doit être coupé afin de séparer VDD et VDD_nRF. La puce peut alors être alimentée directement depuis les pins P22.

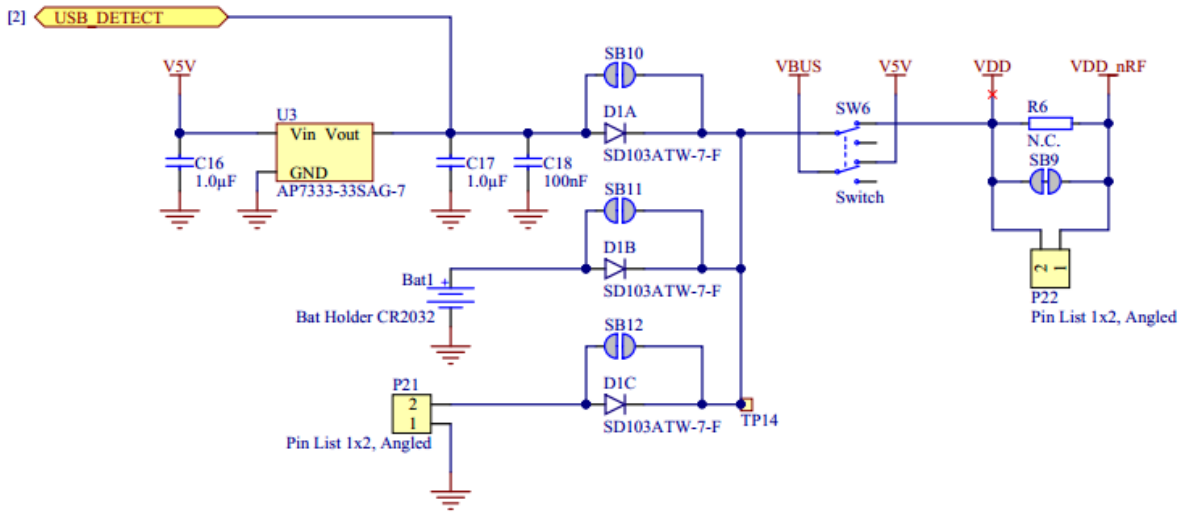


Figure 8 : Schéma d'alimentation du nRF52 Development Kit [https://www.nordicsemi.com/eng/nordic/download_resource/50980/3/83440412]

5.4.4 Problèmes rencontrés

Au départ, contrairement à la modification présentée au point 5.4.3, le Development Kit devait être alimenté entièrement par l'alimentation sans fil.

Il a été remarqué qu'entre 1.5 et 2.6 volts, cette carte consomme énormément de courant, un autre circuit a ainsi été mis en place pour essayer de palier à cette problématique, comme on peut le voir à la Figure 9.

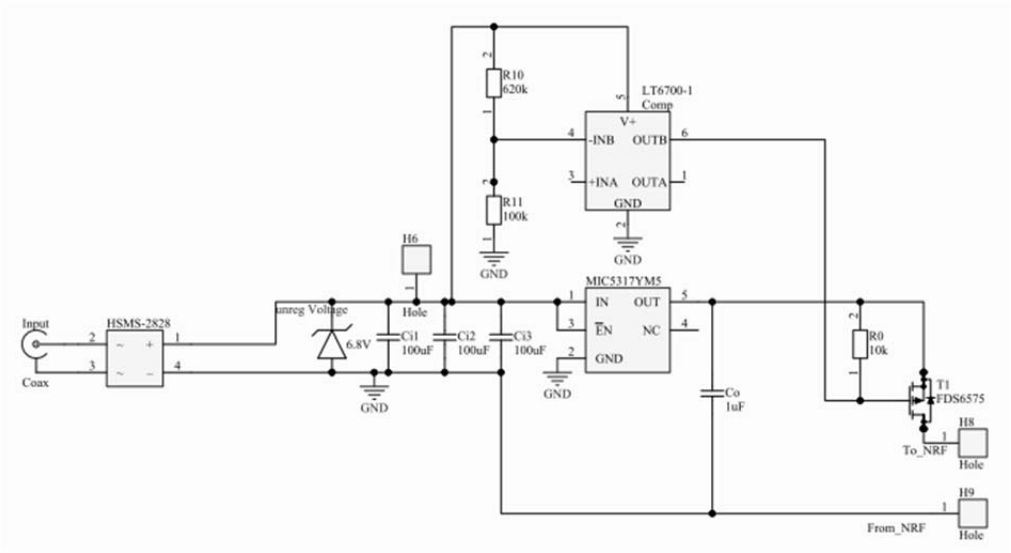


Figure 9 : Schéma alternatif avec comparateur

Dans cette version un comparateur LT6700-1 est présent afin de vérifier que la tension non-régulée est supérieure à 2.6 V. Si c'est le cas, le MOSFET-P en sortie est fermé et la carte est alimentée.

Malgré cela, impossible d'alimenter la carte complète de cette manière. En effet, il a été mesuré, lors du démarrage de la carte, deux pics de courant de 150 mA, suivis d'un pic de courant de 100 mA durant 40 ms ont lieu, ce qui est largement au-delà de ce que peut fournir notre système comme on le voit à la Figure 10.

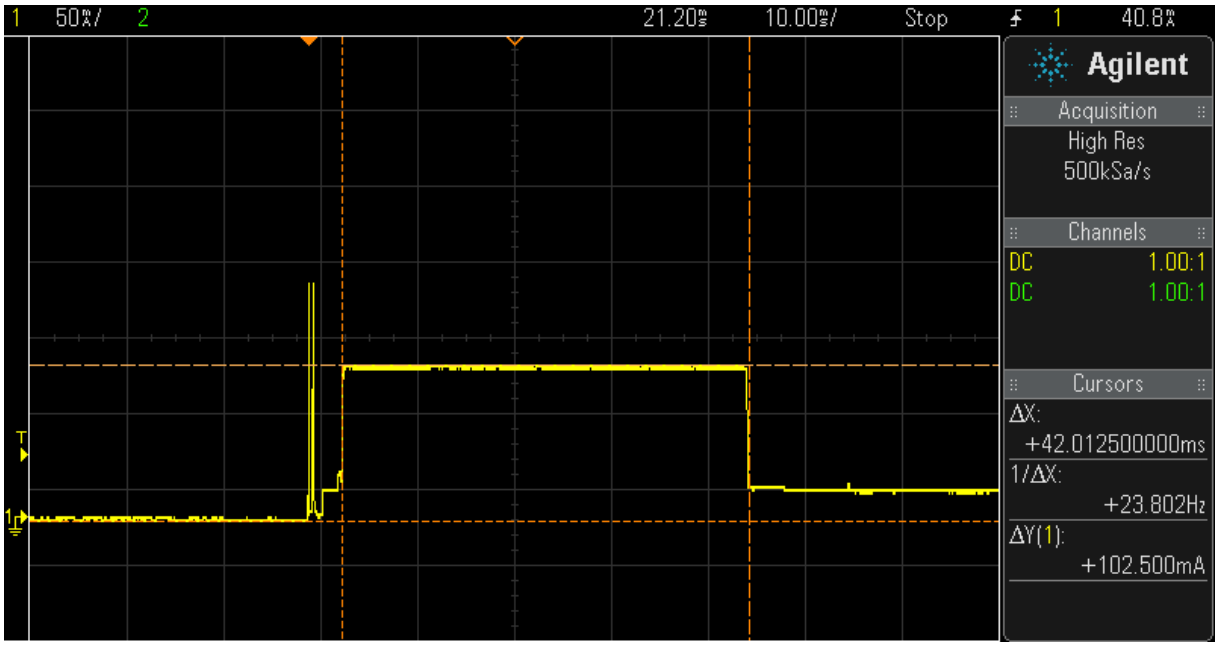


Figure 10 : Courant de démarrage nRF52 Development Kit

Ce circuit a donc dû être abandonné au profit de celui présenté au point 5.4.3.

5.5 Motorisation

5.5.1 Choix de matériel

5.5.1.1 Moteur

Pour la régulation du couplage entre les deux boucles, il a été choisi d'utiliser un moteur pas-à-pas pour contrôler la valeur du condensateur.

Ce choix s'est fait pour la facilité d'utilisation d'un tel moteur, et pour la nécessité d'avoir des positions précises et de les maintenir.

Un moteur bipolaire de type NEMA17 a été choisi, le modèle 17HS5413 de Act-motor qui permet des pas de 1.8° (200 pas par tour), qu'il est possible de multiplier jusqu'à 16 fois grâce aux microsteps, jusqu'à obtenir 3200 positions par tour.

Le courant nominal de ce moteur est de 1.3 A par phase.

5.5.1.2 Driver

Afin de contrôler le moteur, il est nécessaire d'utiliser un driver qui va fournir le courant nécessaire à chaque coil et les alimenter correctement pour avoir le mouvement voulu (direction et vitesse).

Le choix s'est ici porté sur le Big Easy Driver v1.2 proposé sous licence Creative Commons par Brian Schmalz..

Ce driver permet le contrôle d'un moteur bipolaire, avec jusqu'à 2 A par phase, ce qui convient aux 1.3 A du moteur choisi.

Il permet également de gérer la direction et propose un microstepping de 1 à 16 microsteps pour augmenter le nombre de positions possibles.

5.5.2 Implémentation

5.5.2.1 Moteur

Le moteur était d'abord relié au condensateur variable directement à l'aide d'un accouplement élastique. Ce type d'accouplement permet notamment d'atténuer les chocs lors des démarrages et des arrêts du moteur, mais il permet également d'accoupler deux arbres qui ne sont pas tout à fait alignés.

Un problème se posait avec cette installation : l'accouplement métallique étant un conducteur électrique, une grande quantité d'énergie était perdue en s'échappant par le moteur.

Il a ainsi été nécessaire de rajouter un isolant entre deux, c'est pour cela qu'une baguette de bois a été mise en place, reliée à l'arbre du moteur par un tube de plastique, comme on le voit dans la Figure 11.

Figure 11 : Accouplement du condensateur et du moteur

5.5.2.2 Driver

Comme expliqué au point 5.5.1.2, le driver permet d'utiliser des microsteps qui augmentent le nombre de positions possibles. Ce paramètre sera ici laissé à 16 microsteps, afin d'avoir les modifications de capacité les plus petites possibles sur le condensateur.

On atteint les 3200 positions par tour pour le moteur choisi, ce qui donne sur le condensateur une variation théorique de 29.53 fF par position.

Pour contrôler ce driver, il est nécessaire d'utiliser au minimum deux sorties du nRF52 pour contrôler deux pins de la carte :

- DIR : lorsque la pin DIR est à 0, le moteur tourne dans une direction, sinon il tourne dans la direction opposée
- STEP : la pin step est maintenue à 1 par défaut. Chaque fois qu'elle passe à 0, le moteur effectue un pas.

Finalement, le courant doit être limité au courant nominal du moteur de 1.3 A. Pour cela, il suffit de tourner le potentiomètre CUR_ADJ et de mesurer la tension sur la pin TP1. La résistance de mesure vaut 0.11 Ω, la formule suivante donne la tension voulue :

$$Vref = I_{max} * 8 * R_s = 1.3 * 8 * 0.11 = 1.144\ V$$

5.5.2.3 Programmation

Pour le contrôle du driver, le module PWM du nRF52 est utilisé. Celui-ci est configuré avec un clock à 125 khz.

Ainsi, pour contrôler la vitesse de rotation du moteur, il faut définir la valeur maximale (top_value) jusqu'à laquelle va compter le PWM avant de faire basculer la sortie.

La vitesse du moteur va dépendre de la vitesse à laquelle les valeurs de tension sont reçues depuis le Bluetooth. En effet, pour ne manquer aucune variation de tension, il ne faut pas que le moteur tourne plus rapidement que les valeurs ne sont mesurées.

Avec les 20 valeurs de tension reçues chaque secondes, le moteur peut donc faire 20 microsteps par seconde.

Plusieurs vitesses sont définies dans le code : MIN_SPEED, DOUBLE_SPEED et QUADRUPLE_SPEED

MIN_SPEED est utilisé pour une régulation précise.

QUADRUPLE_SPEED est lui utilisé lorsque la tension est trop faible pour alimenter le nRF52 et que l'on ne reçoit donc pas de mesures par Bluetooth. Cette vitesse permet de rapidement faire le tour de toutes les valeurs du condensateur jusqu'à en atteindre une suffisante pour effectuer une connexion.

5.5.2.4 Régulation

La fonction de régulation est appelée à chaque fois qu'une nouvelle valeur de tension est reçue.

Si aucune connexion Bluetooth n'est détectée, cela signifie que la tension d'alimentation est trop faible pour faire fonctionner le nRF52832. Le moteur va alors tourner à QUADRUPLE_SPEED jusqu'à atteindre une valeur suffisante pour que la carte commence l'advertising.

A ce moment, la vitesse va diminuer jusqu'à MIN_SPEED pour démarrer la partie de régulation.

Le moteur va alors tourner dans une direction. Si la valeur mesurée est en augmentation, le moteur continue. Si elle diminue, c'est que le moteur tourne dans la mauvaise direction et s'éloigne de l'objectif, il faut donc inverser la direction du moteur.

La tension non-régulée est limitée par une diode Zener et ne peut pas dépasser les 6.8 V, il n'est donc pas possible de connaître l'état du couplage au-delà de cette valeur, le moteur peut donc s'arrêter lorsque cette valeur est atteinte et il ne recommencera à tourner que lorsque la tension recommence à baisser.

Toutes ces étapes sont résumées sur le flowchart de la Figure 12.

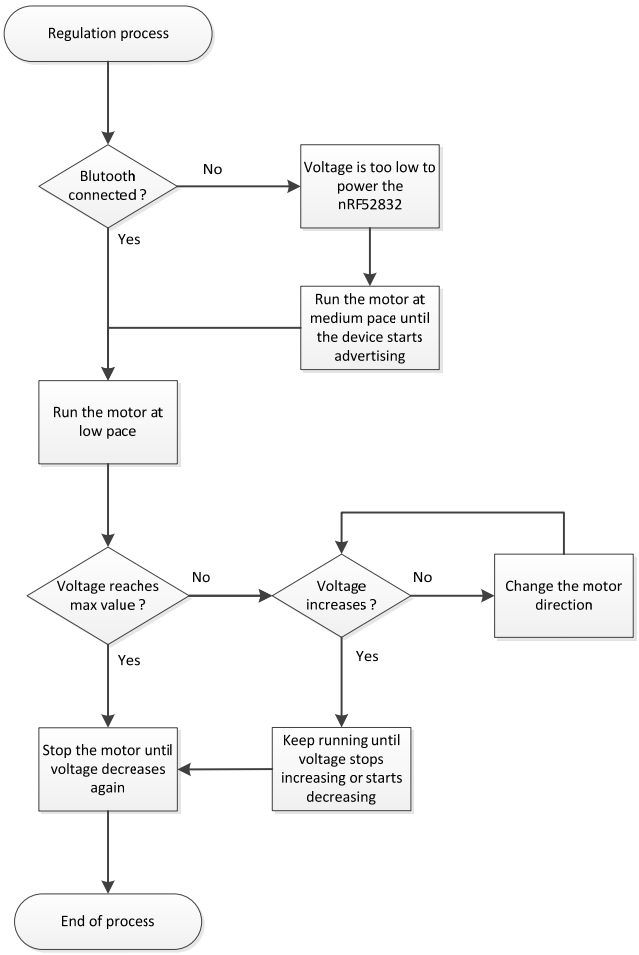


Figure 12 : Flowchart de la régulation motorisée

La difficulté de cette régulation réside dans le fait que la carte de mesure dispose d'une capacité d'entrée de 300 µF, ce qui induit une grande inertie dans la valeur de la tension.

En effet, comme la tension n'augmente ou ne diminue pas instantanément, lorsqu'un changement est détecté sur une mesure du convertisseur AD, le moteur, qui a continué à tourner, se retrouve déjà trop loin.

Pour minimiser ce problème, deux solutions ont été retenues :

- Diminution de la vitesse de rotation du moteur
- Ajout d'un délai au changement de direction du moteur pour éviter d'osciller

6 Mesures et tests

6.1 Matériel

Le matériel utilisé pour les différentes mesures est le suivant :

Type	Marque	Modèle	Serial Number
Network Analyzer	Agilent	E5071C	MY46104870
Oscilloscope	Agilent	MSO-X 3012A	MY51350230
Function Generator	Agilent	33220A	MY44039498
Signal Generator	HP	8647A	3349A01199
Multimeter	Agilent	U1252A	TW46410030
Multimeter	Agilent	U1252A	TW46410014

6.2 Transmission Bluetooth

Le Bluetooth peut être testé directement depuis un smartphone, grâce à l'application nRF Master Control Panel, développée par Nordic Semiconductor.

La Figure 13 montre que la carte de mesure a bien démarré l'advertising, avec un intervalle de 500 ms comme prévu dans la configuration.

On voit que le nom ne peut pas être envoyé en entier, seuls les 4 premiers caractères sont affichés à cause de l'envoi du UUID 128-bit qui utilise toute la place à disposition dans le paquet.

Une fois connecté, on voit sur la Figure 14 le service personnalisé VMSR avec son UUID unique et sa *Characteristic* « Voltage value ». La valeur reçue « 6A-A8-A1-3F » correspond à la valeur de la tension mesurée par le convertisseur AD, valeur qui correspond ici à 1.317 V.

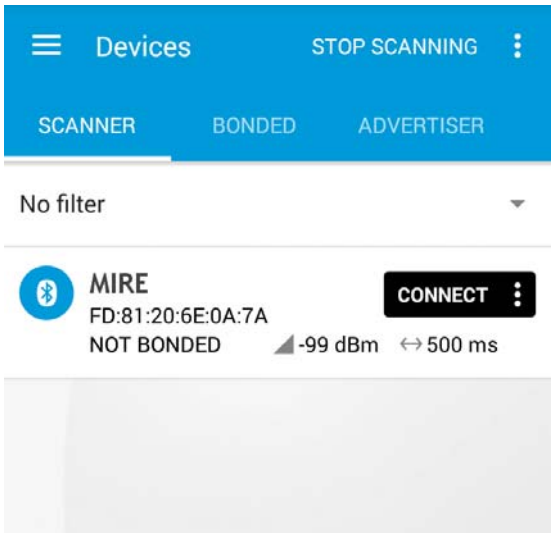


Figure 13 : Appareil visible après un scan

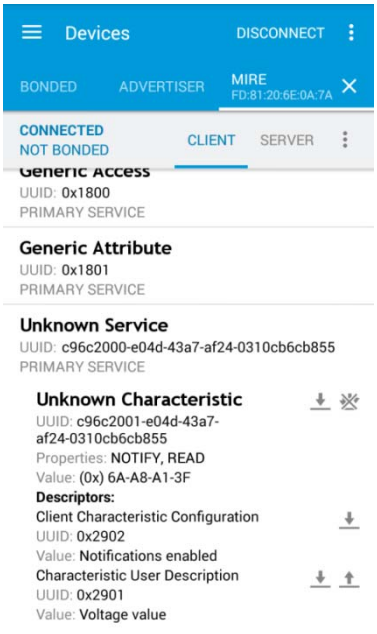


Figure 14 : Connexion et récupération de la tension

6.3 Puissance de sortie

L'alimentation sans fil ne permettant pas un transfert d'énergie conséquent, notamment à cause de la taille réduite de la bobine de réception, il est essentiel que la consommation totale de la carte soit la plus faible possible. Afin de connaître la limite, la puissance maximale est mesurée grâce à une charge résistive.

La tension reste relativement stable jusqu'à descendre à 2.8 V. Après cela, elle va chuter brusquement, c'est la valeur limite à laquelle on s'arrêtera.

Avec l'installation utilisée lors de la réalisation de ce travail, la carte de mesure réalisée parvient au maximum à fournir 1.44 mA à 2.8 V, soit environ 4 mW.

6.4 Couplage

La Figure 15 ci-dessous présente l'impédance en fonction de la fréquence, avec en rouge la bobine de réception et en bleu celle d'alimentation.

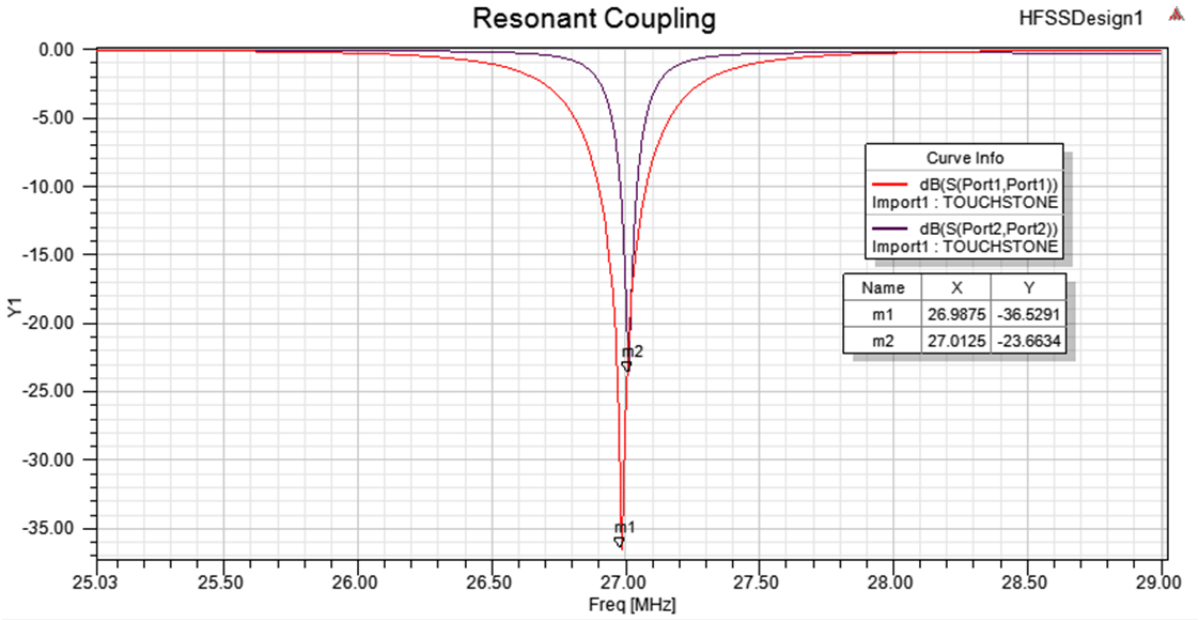


Figure 15 : Couplage résonant des boucles d'alimentation et de réception

On voit que chaque bobine est en condition de résonance, avec une impédance minimale.

La bobine de réception, en rouge, a une fréquence de résonance fixe de 26.9875 MHz comme montré par le marqueur m1.

La courbe de la boucle d'alimentation, en bleu, est obtenue en modifiant la valeur du condensateur variable afin de se rapprocher au mieux des 26.9875 MHz.

6.5 Régulation motorisée

Lorsque la tension est trop basse pour alimenter la carte de mesure, le moteur tourne rapidement jusqu'à obtenir une connexion. Selon la position de base du condensateur, le temps nécessaire peut atteindre jusqu'à une dizaine de secondes.

A partir de la connexion, la valeur de tension va beaucoup osciller, à cause du délai induit par les condensateurs, jusqu'à ce que le moteur parvienne enfin à la stabiliser.

Comme prévu dans le cahier des charges, on finit par obtenir une valeur optimale. Le temps nécessaire peut atteindre, dans les pires cas, près d'une minute.

Si on démarre la régulation alors que la tension est déjà stable, mais pas à une valeur optimale, le résultat est beaucoup plus rapide.

Une fois à la valeur jugée maximale, la régulation continue afin d'assurer que le couplage reste idéal, malgré d'éventuelles modifications environnementales (Le passage proche d'une personne par exemple).

Il est à noter que si l'amplitude fournie à la boucle d'alimentation est trop basse, la régulation ne fonctionne pas. Dans les tests réalisés, le système fonctionne bien à partir de -4.5 dBm mais pas plus bas.

Il est certainement possible d'améliorer à la fois la vitesse de régulation et de diminuer l'amplitude d'alimentation nécessaire en faisant des ajustements dans le code, mais davantage de temps de développement sera nécessaire.

7 Conclusion

Le domaine des hautes fréquences est un domaine complexe qui demande une grande précision et nécessite d'être très méthodique. En effet, le moindre changement dans l'environnement peut entraîner des grandes modifications dans le comportement du système.

Ce problème est apparu notamment avec la boucle de réception, fabriquée à la main et donc très fragile. Ainsi, il est nécessaire de faire très attention en la déplaçant, car la moindre modification dans sa structure va totalement changer les conditions de résonance.

Néanmoins, un éventuel produit final ne présenterait pas ces problèmes, ou très peu. Beaucoup de temps a été ainsi perdu sur ce genre d'éléments, ce qui a ralenti l'avancée du projet, notamment la partie de régulation.

Il a été intéressant de découvrir le Bluetooth Low Energy qui est une technologie toujours plus présente, notamment avec l'avancée du nombre d'objets connectés et du domaine de l'Internet of Things qui prend de plus en plus d'importance.

Après avoir réalisé l'alimentation sans fil, la communication Bluetooth et la régulation motorisée, il m'apparaît aujourd'hui qu'une implémentation qui répond au cahier des charges peut être atteinte rapidement et que le produit final voulu semble réalisable en pratique.

Sion, le 14 juillet 2016

William Espinosa

8 Références

NORDIC SEMICONDUCTOR Nordic Semiconductor Infocenter.
http://infocenter.nordicsemi.com

NORDIC SEMICONDUCTOR nRF52 Development Kit Hardware Files
https://www.nordicsemi.com/eng/nordic/download_resource/50980/3/83440412

BLUETOOTH SIG *Bluetooth®* Core Specification 4.2.
https://www.bluetooth.com/specifications/adopted-specifications

AVAGO TECHNOLOGIES HSMS-282x Datasheet.
http://docs.avagotech.com/docs/AV02-1320EN

MICREL SEMICONDUCTOR MIC5317 Datasheet.
http://ww1.microchip.com/downloads/en/DeviceDoc/MIC5317.pdf

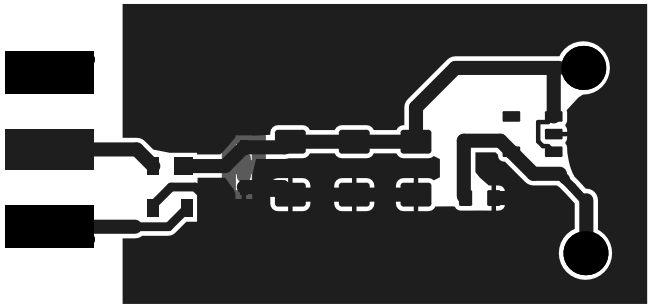
ACT-MOTOR 17HS5413 Stepper motor specifications
http://www.act-motor.com/product/17hs_en.html

ZABER TECHNOLOGIES Microstepping tutorial
http://www.zaber.com/microstepping-tutorial

BRIAN SCHMALZ Big Easy Driver Stepper Motor Driver
http://www.schmalzhaus.com/BigEasyDriver

9 Annexes

Comment	Description	Designator	Manufacturer	Model	Footprint	Quantity
100 uF	Tantalum Capacitor	Ci1, Ci2, Ci3	VISHAY	298W107X0010Q2T	3216	3
1 uF	Tantalum Capacitor	Co	VISHAY	TP8M105M010C	1608	1
	Bridge Quad Schottky Redresser	HSMS-2828	AVAGO	HSMS-2828	SOT143	1
	SMA	Input			SMA-EDGE	1
	High Performance Single 150mA LDO	MIC5317YM5	MICREL	MIC5317-3.0YM5	SOT23-5	1
6.8 V	Zener Diode	Zener	PANASONIC	DZ2J068M0L	SOD323	1



Altium Limited 12a Rodborough Rd Frenchs Forest NSW 2086	ENGINEER:	TITLE: Carte de mesure	
	PCB DESIGNER: William Espinosa		
	DATE: 14.07.2016	PART NO.:	REV: 1
	FILE NAME: Regulator board.PcbDoc	DWG NO:	SCALE: 2:1

```
/* Copyright (c) 2014 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <math.h>


#include "nordic_common.h"
#include "nrf.h"
#include "nrf_drv_saadc.h"
#include "nrf_drv_ppi.h"
#include "nrf_drv_timer.h"
#include "app_error.h"
#include "ble.h"
#include "ble_hci.h"
#include "ble_srv_common.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "boards.h"
#include "softdevice_handler.h"
#include "app_timer.h"
#include "device_manager.h"
#include "pstorage.h"
#include "app_trace.h"
#include "bsp.h"
#include "bsp_btn_ble.h"
#include "sensorsim.h"
#include "nrf_gpio.h"
#include "ble_hci.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_vmsr.h"
#include "SEGGER_RTT.h"
#include "nrf_soc.h"

#define IS_SRVC_CHANGED_CHARACT_PRESENT  1                                    /**<
Include or not the service_changed characteristic. if not enabled, the server's database
cannot be changed for the lifetime of the device*/

#define CENTRAL_LINK_COUNT               0                                    /**<
Number of central links used by the application. When changing this number remember to
adjust the RAM settings*/
#define PERIPHERAL_LINK_COUNT            1                                    /**<
Number of peripheral links used by the application. When changing this number remember to
adjust the RAM settings*/

#define DEVICE_NAME                      "MIRES"
/**< Name of device. Will be included in the advertising data. */
#define MANUFACTURER_NAME                "NordicSemiconductor"                /**<
Manufacturer. Will be passed to Device Information Service. */
#define APP_ADV_INTERVAL                 800                                  /**< The
advertising interval (in units of 0.625 ms). */
#define APP_ADV_TIMEOUT_IN_SECONDS       180                                  /**< The
advertising timeout in units of seconds. */

#define APP_TIMER_PRESCALER              0                                    /**<
Value of the RTC1 PRESCALER register. */
#define APP_TIMER_OP_QUEUE_SIZE          1                                    /**<
Size of timer operation queues. */
```



```
#define MIN_CONN_INTERVAL                MSEC_TO_UNITS(100, UNIT_1_25_MS)           /**<
Minimum acceptable connection interval (0.1 seconds). */
#define MAX_CONN_INTERVAL                MSEC_TO_UNITS(200, UNIT_1_25_MS)           /**<
Maximum acceptable connection interval (0.2 second). */
#define SLAVE_LATENCY                    0                                          /**<
Slave latency. */
#define CONN_SUP_TIMEOUT                 MSEC_TO_UNITS(4000, UNIT_10_MS)            /**<
Connection supervisory timeout (4 seconds). */

#define FIRST_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(5000, APP_TIMER_PRESCALER) /**<
Time from initiating event (connect or start of notification) to first time
sd_ble_gap_conn_param_update is called (5 seconds). */
#define NEXT_CONN_PARAMS_UPDATE_DELAY    APP_TIMER_TICKS(30000, APP_TIMER_PRESCALER)/**<
Time between each call to sd_ble_gap_conn_param_update after the first call (30 seconds). */
#define MAX_CONN_PARAMS_UPDATE_COUNT     3                                          /**<
Number of attempts before giving up the connection parameter negotiation. */

#define SEC_PARAM_BOND                   1                                          /**<
Perform bonding. */
#define SEC_PARAM_MITM                   0                                          /**< Man
In The Middle protection not required. */
#define SEC_PARAM_LESC                   0                                          /**< LE
Secure Connections not enabled. */
#define SEC_PARAM_KEYPRESS               0                                          /**<
Keypress notifications not enabled. */
#define SEC_PARAM_IO_CAPABILITIES        BLE_GAP_IO_CAPS_NONE                       /**< No
I/O capabilities. */
#define SEC_PARAM_OOB                    0                                          /**< Out
Of Band data not available. */
#define SEC_PARAM_MIN_KEY_SIZE           7                                          /**<
Minimum encryption key size. */
#define SEC_PARAM_MAX_KEY_SIZE           16                                         /**<
Maximum encryption key size. */

#define DEAD_BEEF                        0xDEADBEEF                                 /**<
Value used as error code on stack dump, can be used to identify stack location on stack
unwind. */

#define SAADC_TIMER_INTERVAL
50                                                                                  /**<
SAADC conversion interval. */

static dm_application_instance_t         m_app_handle;                              /**<
Application identifier allocated by device manager */

static uint16_t                          m_conn_handle  = BLE_CONN_HANDLE_INVALID;  /**<
Handle of the current connection. */



static ble_vmsr_t
m_vmsr;                                                                    /**< Voltage
Measurement Service structure. */

static                                                                  nrf_saadc_value_t
adc_buf;                                  /**< SAADC buffer. */

static const nrf_drv_timer_t                    interval_timer  =
NRF_DRV_TIMER_INSTANCE(1);          /**< Using Timer1 for the interval between two AD
conversions. */



/**@brief Callback function for asserts in the SoftDevice.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to
 analyze
```

```
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num   Line number of the failing ASSERT call.
 * @param[in] file_name  File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}

/**@brief Function for handling the ADC interrupt.
 *
 * @param[in] p_event       Saadc event
 */
void saadc_event_handler(nrf_drv_saadc_evt_t const * p_event)
{
        // SAADC Conversion done
    if (p_event->type == NRF_DRV_SAADC_EVT_DONE)
    {
                char                                str[10];

        nrf_saadc_value_t   adc_result;
        uint32_t            err_code;
                float                               voltage_value;
                uint16_t                        adc_max_val             =
                1023;           // 14 bits ADC (10 bits effective)
                double                          max_voltage             =
                6.8;            // max unregulated voltage value, limited by the Zener Diode

        adc_result = p_event->data.done.p_buffer[0];

        err_code = nrf_drv_saadc_buffer_convert(p_event->data.done.p_buffer, 1);
        APP_ERROR_CHECK(err_code);

                // ADC is 14-bit but only 10 bits are really effective so the 4 LSB are
                removed
                adc_result = adc_result >> 4;

                voltage_value = adc_result;

                // Calculate the real voltage from the adc value
                voltage_value    = voltage_value*max_voltage/adc_max_val;

                // Print voltage value
                sprintf(str, "%f V\n", voltage_value);

                SEGGER_RTT_WriteString(0, str);

                // Uninitialize the SAADC driver and clear all the SAADC interrupt to reduce
                current consumption
                nrf_drv_saadc_uninit();
                NVIC_ClearPendingIRQ(SAADC_IRQn);


        err_code = ble_vmsr_voltage_update(&m_vmsr, voltage_value);

                if (
            (err_code != NRF_SUCCESS)
            &&
            (err_code != NRF_ERROR_INVALID_STATE)
            &&
            (err_code != BLE_ERROR_NO_TX_PACKETS)
            &&
            (err_code != BLE_ERROR_GATTS_SYS_ATTR_MISSING)
          )
        {
            APP_ERROR_HANDLER(err_code);
        }
    }
}
```



```
/**@brief Configure ADC to do voltage measurement.
 */
static void adc_configure(void)
{
    ret_code_t err_code = nrf_drv_saadc_init(NULL, saadc_event_handler);
    APP_ERROR_CHECK(err_code);

        // Use the analog input AIN0 (P0.02)
    nrf_saadc_channel_config_t config =
    NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN0);


        // The NRF52832 uses a 0.6 V internal reference.
        // To match this reference, a gain of 1/4 is used.
        // Therefore, the max input value is 0.6 * 4 = 2.4 V.
        config.gain = NRF_SAADC_GAIN1_4;


        // The ADC acquisition time depends on the input resistance
        // Using a 800kOhm to limit the current -> 40us Acquisition time required
        config.acq_time = NRF_SAADC_ACQTIME_40US;

        // Enable SAADC channel 0 on input AIN0
        err_code = nrf_drv_saadc_channel_init(0,&config);
    APP_ERROR_CHECK(err_code);

        NRF_SAADC->CH[0].CONFIG |= 0x01000000;      //configure burst mode for channel 0
        (set bit 24)


        // Fill the buffer with the result of the conversion
    err_code = nrf_drv_saadc_buffer_convert(&adc_buf, 1);

        APP_ERROR_CHECK(err_code);
}

/**@brief Handler function for interval timer.
 *
 * @details This function will be called when the interval timer target value is reached.
 *                  It starts the sample timer that will get the ADC samples
 */
void interval_timer_handler(nrf_timer_event_t event_type, void* p_context)
{
        // Configure the ADC each time
        adc_configure();

        // Request a new sample
        nrf_drv_saadc_sample();
}

/**@brief Function for the Timer initialization.
 *
 * @details Initializes the timer module. This creates and starts application timers.
 */
static void timers_init(void)
{
        uint32_t err_code;

    // Initialize application timer module.
    APP_TIMER_INIT(APP_TIMER_PRESCALER, APP_TIMER_OP_QUEUE_SIZE, false);

        // Initialize interval timer
        err_code = nrf_drv_timer_init(&interval_timer, NULL, interval_timer_handler);
    APP_ERROR_CHECK(err_code);

        // 50 ms timer interval
        uint32_t ticks = nrf_drv_timer_ms_to_ticks(&interval_timer, SAADC_TIMER_INTERVAL);


        nrf_drv_timer_extended_compare(&interval_timer, NRF_TIMER_CC_CHANNEL0, ticks,
```

```
        NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);

        //nrf_drv_timer_enable(&interval_timer); // Timer disabled at startup, wait for a
        BLE connection to start
}


/**@brief Function for the GAP initialization.
 *
 * @details This function sets up all the necessary GAP (Generic Access Profile) parameters
 of the
 *          device including the device name, appearance, and the preferred connection
 parameters.
 */
static void gap_params_init(void)
{
    uint32_t                err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *)DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    // Choose an appearance
    //err_code = sd_ble_gap_appearance_set(BLE_APPEARANCE_);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

        // Connection parameters
    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    uint32_t            err_code;
    ble_vmsr_init_t     vmsr_init;

    // Initialize Voltage Measurement Service.
    memset(&vmsr_init, 0, sizeof(vmsr_init));

    vmsr_init.voltage_value_write_handler = NULL;

    err_code = ble_vmsr_init(&m_vmsr, &vmsr_init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling the Connection Parameters Module.
 *
 * @details This function will be called for all events in the Connection Parameters Module
 which
 *          are passed to the application.
 *          @note All this function does is to disconnect. This could have been done by simply
 *                setting the disconnect_on_fail config parameter, but instead we use the
 event
 *                handler mechanism to demonstrate its use.
 *
 * @param[in] p_evt  Event received from the Connection Parameters Module.
```



```
 */
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    uint32_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}


/**@brief Function for handling a Connection Parameters error.
 *
 * @param[in] nrf_error  Error code containing information about what went wrong.
 */
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for initializing the Connection Parameters module.
 */
static void conn_params_init(void)
{
    uint32_t               err_code;
    ble_conn_params_init_t cp_init;

    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = false;
    cp_init.evt_handler                    = on_conn_params_evt;
    cp_init.error_handler                  = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for putting the chip into sleep mode.
 *
 * @note This function will not return.
 */
static void sleep_mode_enter(void)
{
    uint32_t err_code;

    // Prepare wakeup buttons.
    err_code = bsp_btn_ble_sleep_mode_prepare();
    APP_ERROR_CHECK(err_code);

    // Go to system-off mode (this function will not return; wakeup will cause a reset).
    err_code = sd_power_system_off();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling advertising events.
 *
 * @details This function will be called for advertising events which are passed to the
 application.
 *
 * @param[in] ble_adv_evt  Advertising event.
 */
static void on_adv_evt(ble_adv_evt_t ble_adv_evt)
```

```
{
    switch (ble_adv_evt)
    {
        case BLE_ADV_EVT_FAST:
            break;
        case BLE_ADV_EVT_IDLE:
                        // Stops advertising after some time


                        // Restart
                        ble_advertising_start(BLE_ADV_MODE_FAST);

            // Go to sleep mode
                        //sleep_mode_enter();
            break;
        default:
            break;
    }
}


/**@brief Function for handling the Application's BLE Stack events.
 *
 * @param[in] p_ble_evt  Bluetooth stack event.
 */
static void on_ble_evt(ble_evt_t * p_ble_evt)
{
    switch (p_ble_evt->header.evt_id)
            {
        case BLE_GAP_EVT_CONNECTED:
                        // Start the ADC timer on connect
                        nrf_drv_timer_enable(&interval_timer);

            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            break;

        case BLE_GAP_EVT_DISCONNECTED:
                        // Stop the ADC timer
                        nrf_drv_timer_disable(&interval_timer);

            m_conn_handle = BLE_CONN_HANDLE_INVALID;
            break;

        default:
            break;
    }
}


/**@brief Function for dispatching a BLE stack event to all modules with a BLE stack event
handler.
 *
 * @details This function is called from the BLE Stack event interrupt handler after a BLE
 stack
 *          event has been received.
 *
 * @param[in] p_ble_evt  Bluetooth stack event.
 */
static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
{
    dm_ble_evt_handler(p_ble_evt);
    ble_conn_params_on_ble_evt(p_ble_evt);
    bsp_btn_ble_on_ble_evt(p_ble_evt);
    on_ble_evt(p_ble_evt);
    ble_advertising_on_ble_evt(p_ble_evt);
        ble_vmsr_on_ble_evt(&m_vmsr, p_ble_evt);
}


/**@brief Function for dispatching a system event to interested modules.
 *
```



```
 * @details This function is called from the System event interrupt handler after a system
 *          event has been received.
 *
 * @param[in] sys_evt  System stack event.
 */
static void sys_evt_dispatch(uint32_t sys_evt)
{
    pstorage_sys_event_handler(sys_evt);
    ble_advertising_on_sys_evt(sys_evt);
}


/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    uint32_t err_code;

    nrf_clock_lf_cfg_t clock_lf_cfg = NRF_CLOCK_LFCLKSRC;

        // Initialize the SoftDevice handler module.
    SOFTDEVICE_HANDLER_INIT(&clock_lf_cfg, NULL);

        ble_enable_params_t ble_enable_params;
    err_code = softdevice_enable_get_default_config(CENTRAL_LINK_COUNT,
                                                    PERIPHERAL_LINK_COUNT,
                                                    &ble_enable_params);
    APP_ERROR_CHECK(err_code);

    //Check the ram settings against the used number of links
    CHECK_RAM_START_ADDR(CENTRAL_LINK_COUNT,PERIPHERAL_LINK_COUNT);

    // Enable BLE stack.
    err_code = softdevice_enable(&ble_enable_params);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.
    err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.
    err_code = softdevice_sys_evt_handler_set(sys_evt_dispatch);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling events from the BSP module.
 *
 * @param[in]   event   Event generated by button press.
 */
void bsp_event_handler(bsp_event_t event)
{
    uint32_t err_code;
    switch (event)
    {
        case BSP_EVENT_SLEEP:
            sleep_mode_enter();
            break;

        case BSP_EVENT_DISCONNECT:
            err_code = sd_ble_gap_disconnect(m_conn_handle,
            BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            if (err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;

        case BSP_EVENT_WHITELIST_OFF:
```

```
            err_code = ble_advertising_restart_without_whitelist();
            if (err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;

        default:
            break;
    }
}


/**@brief Function for handling the Device Manager events.
 *
 * @param[in] p_evt  Data associated to the device manager event.
 */
static uint32_t device_manager_evt_handler(dm_handle_t const * p_handle,
                                           dm_event_t const  * p_event,
                                           ret_code_t        event_result)
{
    APP_ERROR_CHECK(event_result);

#ifdef BLE_DFU_APP_SUPPORT
    if (p_event->event_id == DM_EVT_LINK_SECURED)
    {
        app_context_load(p_handle);
    }
#endif // BLE_DFU_APP_SUPPORT

    return NRF_SUCCESS;
}


/**@brief Function for the Device Manager initialization.
 *
 * @param[in] erase_bonds  Indicates whether bonding information should be cleared from
 *                         persistent storage during initialization of the Device Manager.
 */
static void device_manager_init(bool erase_bonds)
{
    uint32_t               err_code;
    dm_init_param_t        init_param = {.clear_persistent_data = erase_bonds};
    dm_application_param_t register_param;

    // Initialize persistent storage module.
    err_code = pstorage_init();
    APP_ERROR_CHECK(err_code);

    err_code = dm_init(&init_param);
    APP_ERROR_CHECK(err_code);

    memset(&register_param.sec_param, 0, sizeof(ble_gap_sec_params_t));

    register_param.sec_param.bond         = SEC_PARAM_BOND;
    register_param.sec_param.mitm         = SEC_PARAM_MITM;
    register_param.sec_param.lesc         = SEC_PARAM_LESC;
    register_param.sec_param.keypress     = SEC_PARAM_KEYPRESS;
    register_param.sec_param.io_caps      = SEC_PARAM_IO_CAPABILITIES;
    register_param.sec_param.oob          = SEC_PARAM_OOB;
    register_param.sec_param.min_key_size = SEC_PARAM_MIN_KEY_SIZE;
    register_param.sec_param.max_key_size = SEC_PARAM_MAX_KEY_SIZE;
    register_param.evt_handler            = device_manager_evt_handler;
    register_param.service_type           = DM_PROTOCOL_CNTXT_GATT_SRVR_ID;

    err_code = dm_register(&m_app_handle, &register_param);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the Advertising functionality.
```



```
 */
static void advertising_init(void)
{
    uint32_t      err_code;
    ble_advdata_t advdata;


        ble_uuid_t m_adv_uuids = {VMSR_UUID_SERVICE, m_vmsr.uuid_type}; /**< Universally
        unique service identifiers. */


    // Build advertising data struct to pass into @ref ble_advertising_init.
    memset(&advdata, 0, sizeof(advdata));

    advdata.name_type               = BLE_ADVDATA_FULL_NAME;
    advdata.include_appearance      = true;
    advdata.flags                   = BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE;
    advdata.uuids_complete.uuid_cnt = 1;
    advdata.uuids_complete.p_uuids  = &m_adv_uuids;

    ble_adv_modes_config_t options = {0};
    options.ble_adv_fast_enabled  = BLE_ADV_FAST_ENABLED;
    options.ble_adv_fast_interval = APP_ADV_INTERVAL;
    options.ble_adv_fast_timeout  = APP_ADV_TIMEOUT_IN_SECONDS;

    err_code = ble_advertising_init(&advdata, NULL, &options, on_adv_evt, NULL);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds  Will be true if the clear bonding button was pressed to wake
 the application up.
 */
static void buttons_leds_init(bool * p_erase_bonds)
{
    bsp_event_t startup_event;

    uint32_t err_code = bsp_init(BSP_INIT_LED | BSP_INIT_BUTTONS,
                                 APP_TIMER_TICKS(100, APP_TIMER_PRESCALER),
                                 bsp_event_handler);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_btn_ble_init(NULL, &startup_event);
    APP_ERROR_CHECK(err_code);

    *p_erase_bonds = (startup_event == BSP_EVENT_CLEAR_BONDING_DATA);
}


/**@brief Function for the Power manager.
 */
static void power_manage(void)
{
    uint32_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for application main entry.
 */
int main(void)
{
    uint32_t err_code;
    bool erase_bonds;

    // Initialize.

        // Use DCDC converter instead of LDO to reduce power consumption
        sd_power_dcdc_mode_set(NRF_POWER_DCDC_ENABLE);
```

```
        timers_init();
    buttons_leds_init(&erase_bonds);
    ble_stack_init();
    device_manager_init(erase_bonds);

        gap_params_init();
    services_init();
    advertising_init();
    conn_params_init();

    // Start BLE advertising.
    err_code = ble_advertising_start(BLE_ADV_MODE_FAST);
    APP_ERROR_CHECK(err_code);

    // Enter main loop.
    for (;;)
    {
        power_manage();
    }
}

/**
 * @}
 */
```

```c
/* Copyright (c) 2012 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

#include "ble_vmsr.h"
#include <string.h>
#include "nordic_common.h"
#include "ble_srv_common.h"
#include "app_util.h"


/**@brief Connect event handler.
 *
 * @param[in]   p_vmsr      Voltage Measurement Service structure.
 * @param[in]   p_ble_evt   Event received from the BLE stack.
 */
static void on_connect(ble_vmsr_t * p_vmsr, ble_evt_t * p_ble_evt)
{
    p_vmsr->conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
}


/**@brief Disconnect event handler.
 *
 * @param[in]   p_vmsr      Voltage Measurement Service structure.
 * @param[in]   p_ble_evt   Event received from the BLE stack.
 */
static void on_disconnect(ble_vmsr_t * p_vmsr, ble_evt_t * p_ble_evt)
{
    UNUSED_PARAMETER(p_ble_evt);
    p_vmsr->conn_handle = BLE_CONN_HANDLE_INVALID;
}


void ble_vmsr_on_ble_evt(ble_vmsr_t * p_vmsr, ble_evt_t * p_ble_evt)
{
    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
            on_connect(p_vmsr, p_ble_evt);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnect(p_vmsr, p_ble_evt);
            break;
        default:
            break;
    }
}

static uint32_t voltage_value_char_add(ble_vmsr_t * p_vmsr, const ble_vmsr_init_t *
p_vmsr_init)
{
    ble_gatts_char_md_t char_md;
    ble_gatts_attr_md_t cccd_md;
    ble_gatts_attr_t    attr_char_value;
    ble_uuid_t          ble_uuid;
    ble_gatts_attr_md_t attr_md;


    memset(&cccd_md, 0, sizeof(cccd_md));

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.read_perm);
```



```c
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&cccd_md.write_perm);

    cccd_md.vloc = BLE_GATTS_VLOC_STACK;

    memset(&char_md, 0, sizeof(char_md));

        // description that the client can retrieve
        static char user_desc[] = "Voltage value";

    char_md.char_props.read                 = 1;
    char_md.char_props.notify               = 1;
    char_md.p_char_user_desc                = (uint8_t *) user_desc;
        char_md.char_user_desc_size             = strlen(user_desc);
        char_md.char_user_desc_max_size     = strlen(user_desc);
    char_md.p_char_pf                       = NULL;
    char_md.p_user_desc_md                  = NULL;
    char_md.p_cccd_md                       = &cccd_md;
    char_md.p_sccd_md                       = NULL;

    ble_uuid.type = p_vmsr->uuid_type;
    ble_uuid.uuid = VMSR_UUID_VOLTAGE_VALUE_CHAR;

    memset(&attr_md, 0, sizeof(attr_md));

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.read_perm);
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&attr_md.write_perm);

    attr_md.vloc        = BLE_GATTS_VLOC_STACK;
    attr_md.rd_auth     = 0;
    attr_md.wr_auth     = 0;
    attr_md.vlen        = 0;

    memset(&attr_char_value, 0, sizeof(attr_char_value));

    attr_char_value.p_uuid      = &ble_uuid;
    attr_char_value.p_attr_md   = &attr_md;
    attr_char_value.init_len    = sizeof(uint8_t);
    attr_char_value.init_offs   = 0;
    attr_char_value.max_len     = sizeof(float);
    attr_char_value.p_value     = NULL;

    return sd_ble_gatts_characteristic_add(p_vmsr->service_handle, &char_md,
                                           &attr_char_value,
                                           &p_vmsr->voltage_value_char_handles);
}

uint32_t ble_vmsr_init(ble_vmsr_t * p_vmsr, const ble_vmsr_init_t * p_vmsr_init)
{
    uint32_t   err_code;
    ble_uuid_t ble_uuid;

    // Initialize service structure
    p_vmsr->conn_handle                         = BLE_CONN_HANDLE_INVALID;
    p_vmsr->voltage_value_write_handler     = p_vmsr_init->voltage_value_write_handler;
        p_vmsr->voltage                                 = 0;

    // Add base UUID to softdevice's internal list.
    ble_uuid128_t base_uuid = VMSR_UUID_BASE;
    err_code = sd_ble_uuid_vs_add(&base_uuid, &p_vmsr->uuid_type);
    if (err_code != NRF_SUCCESS)
    {
        return err_code;
    }

    ble_uuid.type = p_vmsr->uuid_type;
    ble_uuid.uuid = VMSR_UUID_SERVICE;

    err_code = sd_ble_gatts_service_add(BLE_GATTS_SRVC_TYPE_PRIMARY, &ble_uuid,
    &p_vmsr->service_handle);
    if (err_code != NRF_SUCCESS)
    {
```

```
        return err_code;
    }

    err_code = voltage_value_char_add(p_vmsr, p_vmsr_init);
    if (err_code != NRF_SUCCESS)
    {
        return err_code;
    }

    return NRF_SUCCESS;
}

uint32_t ble_vmsr_voltage_update(ble_vmsr_t * p_vmsr, float voltage_value)
{
    if (p_vmsr == NULL)
    {
        return NRF_ERROR_NULL;
    }

             uint32_t err_code = NRF_SUCCESS;

        // Send value if connected
        if ((p_vmsr->conn_handle != BLE_CONN_HANDLE_INVALID))
        {
            // Only send the voltage over BLE if the value is different from the last one
            if (voltage_value != p_vmsr->voltage)
            {
                    ble_gatts_hvx_params_t params;
                    uint16_t len = sizeof(voltage_value);       // Float size = 4

                    memset(&params, 0, sizeof(params));

                    params.type         = BLE_GATT_HVX_NOTIFICATION;
                    params.handle   = p_vmsr->voltage_value_char_handles.value_handle;
                    params.offset   = 0;
                    params.p_data   = (uint8_t *)&voltage_value;
                    params.p_len        = &len;

                    // Update the value in the struct
                    p_vmsr->voltage = voltage_value;

                    //
                    err_code = sd_ble_gatts_hvx(p_vmsr->conn_handle, &params);
            }
        }

        return err_code;
}
```

```c
/* Copyright (c) 2012 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

/** @file
 *
 * @brief Voltage Measurement Service module.
 *
 * @details This module implements the Voltage Measurement Service with the Voltage
 characteristic.
 *          During initialization it adds the Voltage Measurement Service and Voltage
 characteristic
 *          to the BLE stack database.
 *
 *
 * @note Attention!
 *  To maintain compliance with Nordic Semiconductor ASA Bluetooth profile
 *  qualification listings, this section of source code must not be modified.
 */

#ifndef BLE_VMSR_H__
#define BLE_VMSR_H__

#include <stdint.h>
#include "ble.h"
#include "ble_srv_common.h"

// Custom UUID for the Voltage Measurement Service
#define VMSR_UUID_BASE                      {0x55, 0xB8, 0x6C, 0xCB, 0x10, 0x03, 0x24, 0xAF,
0xA7, 0x43, 0x4D, 0xE0, 0x00, 0x00, 0x6C, 0xC9}
#define VMSR_UUID_SERVICE                   0x2000
#define VMSR_UUID_VOLTAGE_VALUE_CHAR    0x2001

// Forward declaration of the ble_vmsr_t type.
typedef struct ble_vmsr_s ble_vmsr_t;

/**@brief Voltage Measurement Service event handler type. */
typedef void (*ble_vmsr_voltage_update_handler_t) (ble_vmsr_t * p_vmsr, uint8_t new_value);

/**@brief Voltage Measurement Service init structure. */
typedef struct
{
    ble_vmsr_voltage_update_handler_t voltage_value_write_handler; /**< Called when the
    voltage value is updated. */
} ble_vmsr_init_t;

/**@brief Voltage Measurement Service structure. This contains various status information
for the service. */
struct ble_vmsr_s
{
  uint16_t                                service_handle;
  ble_gatts_char_handles_t                voltage_value_char_handles;
  uint8_t                                 uuid_type;
  double                                  voltage;
  uint16_t                                conn_handle;
  bool                                    is_notifying;
  ble_vmsr_voltage_update_handler_t     voltage_value_write_handler;
};

/**@brief Function for initializing the Voltage Measurement Service.
 *
 * @param[out]  p_vmsr      Voltage Measurement Service structure. This structure will have
 to be supplied by
```



```c
 *                          the application. It will be initialized by this function, and
 will later
 *                          be used to identify this particular service instance.
 * @param[in]   p_vmsr_init  Information needed to initialize the service.
 *
 * @return      NRF_SUCCESS on successful initialization of service, otherwise an error code.
 */
uint32_t ble_vmsr_init(ble_vmsr_t * p_vmsr, const ble_vmsr_init_t * p_vmsr_init);


/**@brief Add Voltage Value characteristic.
 *
 * @param[in]   p_vmsr        Voltage Measurement Service structure.
 * @param[in]   p_vmsr_init   Information needed to initialize the service.
 *
 * @return      NRF_SUCCESS on success, otherwise an error code.
 */
static uint32_t voltage_value_char_add(ble_vmsr_t * p_vmsr, const ble_vmsr_init_t *
p_vmsr_init);


/**@brief Function for handling the Application's BLE Stack events.
 *
 * @details Handles all events from the BLE stack of interest to the Voltage Measurement
 Service.
 *
 * @param[in]   p_vmsr      Voltage Measurement Service structure.
 * @param[in]   p_ble_evt   Event received from the BLE stack.
 */
void ble_vmsr_on_ble_evt(ble_vmsr_t * p_vmsr, ble_evt_t * p_ble_evt);


/**@brief Send the voltage when an updated voltage value is received from the ADC.
 *
 * @param[in]   p_vmsr          Voltage Measurement Service structure.
 * @param[in]   voltage_value       Current voltage value
 *
 * @return      NRF_SUCCESS on success, otherwise an error code.
 */
uint32_t ble_vmsr_voltage_update(ble_vmsr_t * p_vmsr, float voltage_value);

#endif // BLE_VMSR_H__

/** @} */
```

```
/* Copyright (c) 2014 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <math.h>


#include "nordic_common.h"
#include "nrf.h"
#include "app_error.h"
#include "ble.h"
#include "ble_hci.h"
#include "ble_srv_common.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "ble_db_discovery.h"
#include "boards.h"
#include "softdevice_handler.h"
#include "app_timer.h"
#include "device_manager.h"
#include "pstorage.h"
#include "app_trace.h"
#include "bsp.h"
#include "bsp_btn_ble.h"
#include "sensorsim.h"
#include "nrf_gpio.h"
#include "ble_hci.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_vmsr_c.h"
#include "SEGGER_RTT.h"
#include "nrf_drv_pwm.h"
#include "app_util_platform.h"

#define IS_SRVC_CHANGED_CHARACT_PRESENT  1                                          /**<
Include or not the service_changed characteristic. if not enabled, the server's database
cannot be changed for the lifetime of the device*/

#define CENTRAL_LINK_COUNT               1                                          /**<
Number of central links used by the application. When changing this number remember to
adjust the RAM settings*/
#define PERIPHERAL_LINK_COUNT            0                                          /**<
Number of peripheral links used by the application. When changing this number remember to
adjust the RAM settings*/

#define DEVICE_NAME                      "MIRES_Central"                             /**<
Name of device. Will be included in the advertising data. */
#define MANUFACTURER_NAME                "NordicSemiconductor"                      /**<
Manufacturer. Will be passed to Device Information Service. */
#define APP_ADV_INTERVAL                 300                                        /**< The
advertising interval (in units of 0.625 ms. This value corresponds to 25 ms). */
#define APP_ADV_TIMEOUT_IN_SECONDS       180                                        /**< The
advertising timeout in units of seconds. */

#define APP_TIMER_PRESCALER              0                                          /**<
Value of the RTC1 PRESCALER register. */
#define APP_TIMER_OP_QUEUE_SIZE          4                                          /**<
Size of timer operation queues. */
```



```
#define MIN_CONN_INTERVAL                MSEC_TO_UNITS(100, UNIT_1_25_MS)           /**<
Minimum acceptable connection interval (0.1 seconds). */
#define MAX_CONN_INTERVAL                MSEC_TO_UNITS(200, UNIT_1_25_MS)           /**<
Maximum acceptable connection interval (0.2 second). */
#define SLAVE_LATENCY                    0                                          /**<
Slave latency. */
#define CONN_SUP_TIMEOUT                 MSEC_TO_UNITS(4000, UNIT_10_MS)            /**<
Connection supervisory timeout (4 seconds). */

#define FIRST_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(5000, APP_TIMER_PRESCALER) /**<
Time from initiating event (connect or start of notification) to first time
sd_ble_gap_conn_param_update is called (5 seconds). */
#define NEXT_CONN_PARAMS_UPDATE_DELAY    APP_TIMER_TICKS(30000, APP_TIMER_PRESCALER)/**<
Time between each call to sd_ble_gap_conn_param_update after the first call (30 seconds). */
#define MAX_CONN_PARAMS_UPDATE_COUNT     3                                          /**<
Number of attempts before giving up the connection parameter negotiation. */

#define SEC_PARAM_BOND                   1                                          /**<
Perform bonding. */
#define SEC_PARAM_MITM                   0                                          /**< Man
In The Middle protection not required. */
#define SEC_PARAM_LESC                   0                                          /**< LE
Secure Connections not enabled. */
#define SEC_PARAM_KEYPRESS               0                                          /**<
Keypress notifications not enabled. */
#define SEC_PARAM_IO_CAPABILITIES        BLE_GAP_IO_CAPS_NONE                       /**< No
I/O capabilities. */
#define SEC_PARAM_OOB                    0                                          /**< Out
Of Band data not available. */
#define SEC_PARAM_MIN_KEY_SIZE           7                                          /**<
Minimum encryption key size. */
#define SEC_PARAM_MAX_KEY_SIZE           16                                         /**<
Maximum encryption key size. */

#define DEAD_BEEF                        0xDEADBEEF                                 /**<
Value used as error code on stack dump, can be used to identify stack location on stack
unwind. */

#define MIN_CONNECTION_INTERVAL             MSEC_TO_UNITS(7.5,
UNIT_1_25_MS)                    /**< Determines minimum connection interval in millisecond. */
#define MAX_CONNECTION_INTERVAL             MSEC_TO_UNITS(30,
UNIT_1_25_MS)                   /**< Determines maximum connection interval in millisecond. */
#define SLAVE_LATENCY
0                                           /**< Determines slave latency in counts of
connection events. */
#define SUPERVISION_TIMEOUT                 MSEC_TO_UNITS(4000,
UNIT_10_MS)                   /**< Determines supervision time-out in units of 10
millisecond. */

#define SCAN_INTERVAL
0x00A0                                      /**< Determines scan interval in units of
0.625 millisecond. */
#define SCAN_WINDOW
0x0050                                      /**< Determines scan window in units of
0.625 millisecond. */

#define MOTOR_STEP_PIN
22                                                                          /**< Pin
used to control the motor steps */
#define MOTOR_DIR_PIN
23                                                                          /**< Pin
used to control the motor direction */

#define BLE_MESSAGE_FREQUENCY
20                                                                          /**<
Bluetooth LE message frequency (voltage value) */

#define MIN_ACCEPTABLE_VOLTAGE
3                                                                           /**<
Minimal required unregulated voltage */
#define MIN_SPEED_VOLTAGE
```

```
1.6                                                                    /**< When
the voltage is above 1.6 V (= device connected), use the min speed for accuracy */
#define MAX_VOLTAGE
6.8                                                                    /**< Max
possible voltage, limited by the Zener */

#define MIN_SPEED
125000/BLE_MESSAGE_FREQUENCY                                  /**< Motor speeds */
#define DOUBLE_SPEED
MIN_SPEED/2
#define QUADRUPLE_SPEED
DOUBLE_SPEED/2                                                        /**< Number of
different motor speed */

static dm_application_instance_t         m_app_handle;                        /**<
Application identifier allocated by device manager */
static ble_gap_scan_params_t
m_scan_param;                                  /**< Scan parameters requested for scanning
and connection. */
static ble_db_discovery_t                    m_ble_db_discovery;
/**< Structure used to identify the DB Discovery module. */

static uint16_t                           m_conn_handle = BLE_CONN_HANDLE_INVALID;   /**<
Handle of the current connection. */
static dm_handle_t
m_dm_device_handle;                                          /**< Device Manager handle.
*/

static ble_vmsr_c_t
m_vmsr;                                                              /**< Central
Voltage Measurement Service structure. */

static nrf_drv_pwm_t                                    m_pwm0 =
NRF_DRV_PWM_INSTANCE(0);                  /**< PWM0 used to control the motor. */

typedef
enum
                                                     /**< Available motor directions. */
{
        MOTOR_DIR_LEFT,
        MOTOR_DIR_RIGHT
} motor_dir_t;

static motor_dir_t
motor_dir;                                                           /**< Current motor
direction. */
static bool
motor_running;                                                    /**< Motor state. */
static bool                                                          ascending =
false;                                          /**< Is the measured voltage growing ? */
static bool                                                          dir_changed =
false;                                         /**< Motor direction has been changed
recently, wait some time before changing again */
static int                                                           dir_changed_counter
= 0;

static float
previous_voltage;                                                 /**< Previous voltage
value. */


// PWM config
nrf_drv_pwm_config_t pwm_config =
{
        .output_pins =
        {
                MOTOR_STEP_PIN,
                NRF_DRV_PWM_PIN_NOT_USED, // channel 1
                NRF_DRV_PWM_PIN_NOT_USED, // channel 2
                NRF_DRV_PWM_PIN_NOT_USED  // channel 3
        },
```



```
        .irq_priority = _PRIO_APP_HIGH,
        .base_clock   = NRF_PWM_CLK_125kHz,
        .count_mode   = NRF_PWM_MODE_UP,
        .top_value    = QUADRUPLE_SPEED,                          // Quadruple Speed at
        the beginning
        .load_mode    = NRF_PWM_LOAD_COMMON,
        .step_mode    = NRF_PWM_STEP_AUTO
};

static uint16_t /*const*/ seq_values[] =        // PWM duty cycle sequence (50%)
{
                0,
        0x8000,
};
nrf_pwm_sequence_t const m_reg_seq =
{
        .values.p_common = seq_values,
        .length          = NRF_PWM_VALUES_LENGTH(seq_values),
        .repeats         = 0,
        .end_delay       = 0
};


/**
 * @brief Connection parameters requested for connection.
 */
static const ble_gap_conn_params_t m_connection_param =
{
    (uint16_t)MIN_CONNECTION_INTERVAL,   // Minimum connection
    (uint16_t)MAX_CONNECTION_INTERVAL,   // Maximum connection
    0,                                   // Slave latency
    (uint16_t)SUPERVISION_TIMEOUT        // Supervision time-out
};

/**@brief Variable length data encapsulation in terms of length and pointer to data */
typedef struct
{
    uint8_t     * p_data;                                          /**< Pointer to data. */
    uint16_t      data_len;                                        /**< Length of data. */
}data_t;

/**@brief Callback function for asserts in the SoftDevice.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to
 analyze
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num   Line number of the failing ASSERT call.
 * @param[in] file_name  File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}


/**@brief Function for the Timer initialization.
 *
 * @details Initializes the timer module. This creates and starts application timers.
 */
static void timers_init(void)
{
    // Initialize timer module.
    APP_TIMER_INIT(APP_TIMER_PRESCALER, APP_TIMER_OP_QUEUE_SIZE, false);
}


/**@brief Function for the GAP initialization.
```

```
 *
 * @details This function sets up all the necessary GAP (Generic Access Profile) parameters
 of the
 *          device including the device name, appearance, and the preferred connection
 parameters.
 */
static void gap_params_init(void)
{
    uint32_t                err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;

    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *)DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for handling the Connection Parameters Module.
 *
 * @details This function will be called for all events in the Connection Parameters Module
 which
 *          are passed to the application.
 *          @note All this function does is to disconnect. This could have been done by simply
 *                setting the disconnect_on_fail config parameter, but instead we use the
 event
 *                handler mechanism to demonstrate its use.
 *
 * @param[in] p_evt  Event received from the Connection Parameters Module.
 */
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    uint32_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}


/**@brief Function for handling a Connection Parameters error.
 *
 * @param[in] nrf_error  Error code containing information about what went wrong.
 */
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for initializing the Connection Parameters module.
 */
static void conn_params_init(void)
{
    uint32_t               err_code;
    ble_conn_params_init_t cp_init;
```



```
    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = false;
    cp_init.evt_handler                    = on_conn_params_evt;
    cp_init.error_handler                  = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for putting the chip into sleep mode.
 *
 * @note This function will not return.
 */
static void sleep_mode_enter(void)
{
    uint32_t err_code = bsp_indication_set(BSP_INDICATE_IDLE);
    APP_ERROR_CHECK(err_code);

    // Prepare wakeup buttons.
    err_code = bsp_btn_ble_sleep_mode_prepare();
    APP_ERROR_CHECK(err_code);

    // Go to system-off mode (this function will not return; wakeup will cause a reset).
    err_code = sd_power_system_off();
    APP_ERROR_CHECK(err_code);
}

/**@brief Function to start scanning.
 */
static void scan_start(void)
{
    ble_gap_whitelist_t   whitelist;
    ble_gap_addr_t      * p_whitelist_addr[BLE_GAP_WHITELIST_ADDR_MAX_COUNT];
    ble_gap_irk_t       * p_whitelist_irk[BLE_GAP_WHITELIST_IRK_MAX_COUNT];
    uint32_t              err_code;
    uint32_t              count;

    // Verify if there is any flash access pending, if yes delay starting scanning until
    // it's complete.
    err_code = pstorage_access_status_get(&count);
    APP_ERROR_CHECK(err_code);

    // Initialize whitelist parameters.
    whitelist.addr_count = BLE_GAP_WHITELIST_ADDR_MAX_COUNT;
    whitelist.irk_count  = 0;
    whitelist.pp_addrs   = p_whitelist_addr;
    whitelist.pp_irks    = p_whitelist_irk;

    // Request creating of whitelist.
    err_code = dm_whitelist_create(&m_app_handle, &whitelist);
    APP_ERROR_CHECK(err_code);

    if (((whitelist.addr_count == 0) && (whitelist.irk_count == 0)))
    {
        // No devices in whitelist, hence non selective performed.
        m_scan_param.active       = 0;            // Active scanning set.
        m_scan_param.selective    = 0;            // Selective scanning not set.
        m_scan_param.interval     = SCAN_INTERVAL;// Scan interval.
        m_scan_param.window       = SCAN_WINDOW;  // Scan window.
        m_scan_param.p_whitelist  = NULL;         // No whitelist provided.
        m_scan_param.timeout      = 0x0000;       // No timeout.
    }
    else
    {
        // Selective scanning based on whitelist first.
```

```
        m_scan_param.active       = 0;            // Active scanning set.
        m_scan_param.selective    = 1;            // Selective scanning not set.
        m_scan_param.interval     = SCAN_INTERVAL;// Scan interval.
        m_scan_param.window       = SCAN_WINDOW;  // Scan window.
        m_scan_param.p_whitelist  = &whitelist;   // Provide whitelist.
        m_scan_param.timeout      = 0x001E;       // 30 seconds timeout.
    }

    err_code = sd_ble_gap_scan_start(&m_scan_param);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_indication_set(BSP_INDICATE_SCANNING);
    APP_ERROR_CHECK(err_code);
}


/**
 * @brief Parses advertisement data, providing length and location of the field in case
 *        matching data is found.
 *
 * @param[in]  Type of data to be looked for in advertisement data.
 * @param[in]  Advertisement report length and pointer to report.
 * @param[out] If data type requested is found in the data report, type data length and
 *             pointer to data will be populated here.
 *
 * @retval NRF_SUCCESS if the data type is found in the report.
 * @retval NRF_ERROR_NOT_FOUND if the data type could not be found.
 */
static uint32_t adv_report_parse(uint8_t type, data_t * p_advdata, data_t * p_typedata)
{
    uint32_t  index = 0;
    uint8_t * p_data;

    p_data = p_advdata->p_data;

    while (index < p_advdata->data_len)
    {
        uint8_t field_length = p_data[index];
        uint8_t field_type   = p_data[index+1];

        if (field_type == type)
        {
            p_typedata->p_data   = &p_data[index+2];
            p_typedata->data_len = field_length-1;
            return NRF_SUCCESS;
        }
        index += field_length + 1;
    }
    return NRF_ERROR_NOT_FOUND;
}


/**@brief Function for handling the Application's BLE Stack events.
 *
 * @param[in] p_ble_evt  Bluetooth stack event.
 */
static void on_ble_evt(ble_evt_t * p_ble_evt)
{
    uint32_t err_code;
    const ble_gap_evt_t   * p_gap_evt = &p_ble_evt->evt.gap_evt;
        char adv_uuid[16];

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_ADV_REPORT:
        {
            data_t adv_data;
            data_t type_data;

            // Initialize advertisement report for parsing.
            adv_data.p_data = (uint8_t *)p_gap_evt->params.adv_report.data;
```



```
            adv_data.data_len = p_gap_evt->params.adv_report.dlen;

                        // Get the complete 128-bit UUID from the advertisement
            err_code = adv_report_parse(BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE,
                                        &adv_data,
                                        &type_data);

                        memcpy(adv_uuid, type_data.p_data, type_data.data_len);


            // Verify if the uuid matches with the service's uuid.
            if (err_code == NRF_SUCCESS)
            {
                char service_uuid[] = VMSR_UUID_COMPLETE_128BIT;

                                    // Compare the uuids
                                    if(strcmp(service_uuid, adv_uuid) == 0)
                                    {
                                            // Stop scanning
                                            err_code = sd_ble_gap_scan_stop();
                                            APP_ERROR_CHECK(err_code);

                                            err_code = bsp_indication_set(BSP_INDICATE_IDLE);
                                            APP_ERROR_CHECK(err_code);

                                            m_scan_param.selective = 0;
                                            m_scan_param.p_whitelist = NULL;


                                            // Connect
                                            err_code =
                                            sd_ble_gap_connect(&p_gap_evt->params.adv_report.peer_
                                            addr,
                                                        &m_scan_param,
                                                        &m_connection_param);
                                            if (err_code != NRF_SUCCESS)
                                            {
                                                    char error[100];

                                                    sprintf(error, "[APPL]: Connection Request
                                                    Failed, reason %d\r\n", err_code);
                                                    SEGGER_RTT_WriteString(0, error);
                                            }
                }
            }
            break;
        }

        case BLE_GAP_EVT_TIMEOUT:
            if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_SCAN)          // Scan
            timeout
            {
                                    scan_start();
            }
            else if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_CONN) //
            Connection timeout
            {

            }
            break;
        case BLE_GAP_EVT_CONNECTED:
        {
            err_code = ble_vmsr_c_handles_assign(&m_vmsr, p_gap_evt->conn_handle, NULL);
            APP_ERROR_CHECK(err_code);
            break;
        }
        case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
            // Accepting parameters requested by peer.
            err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle,

                                                    &p_gap_evt->params.conn_param_update_reque
```

```
                                                st.conn_params);
            APP_ERROR_CHECK(err_code);
            break;

        default:
            break;
    }
}

/**
 * @brief Regulates the inductive loop using a stepper motor
 */
static void motor_regulation(float current_voltage)
{
        dir_changed_counter++;

        if(dir_changed_counter >= 5)
        {
                dir_changed = false;
        }

        // Device is disconnected because de voltage is too low
        // The motor has to turn until the device can connect, HALF_SPEED may be too fast
        if(m_vmsr.conn_handle == BLE_CONN_HANDLE_INVALID && (!motor_running ||
        pwm_config.top_value != QUADRUPLE_SPEED))
        {
                pwm_config.top_value = QUADRUPLE_SPEED;

                if(motor_running && pwm_config.top_value != QUADRUPLE_SPEED)
                {
                        nrf_drv_pwm_uninit(&m_pwm0);
                }

                motor_running = true;

                nrf_drv_pwm_init(&m_pwm0, &pwm_config, NULL);

                nrf_drv_pwm_simple_playback(&m_pwm0, &m_reg_seq, 1, NRF_DRV_PWM_FLAG_LOOP);
        }
        // At this point, the device should be powered and connected
        // Keep turning at MIN_SPEED
        else if(m_vmsr.conn_handle != BLE_CONN_HANDLE_INVALID &&
                        (!motor_running || pwm_config.top_value != MIN_SPEED) &&
                        current_voltage < MAX_VOLTAGE-0.02)
        {
                pwm_config.top_value = MIN_SPEED;

                if(motor_running && pwm_config.top_value != MIN_SPEED)
                {
                        nrf_drv_pwm_uninit(&m_pwm0);
                }

                motor_running = true;

                nrf_drv_pwm_init(&m_pwm0, &pwm_config, NULL);

                nrf_drv_pwm_simple_playback(&m_pwm0, &m_reg_seq, 1, NRF_DRV_PWM_FLAG_LOOP);

                ascending = true;
        }


        // Actively regulate the loop value at MIN_SPEED
        if(motor_running && pwm_config.top_value == MIN_SPEED)
        {

                if(current_voltage >= MAX_VOLTAGE-0.02)
                {
                        nrf_drv_pwm_uninit(&m_pwm0);

                        motor_running = false;
```



```
                        ascending = false;
                }
                // Voltage value is growing
                else if(ascending)
                {
                        // Voltage value is no more ascending
                        if(current_voltage < previous_voltage && !dir_changed)
                        {
                                ascending = false;

                                dir_changed = true;
                                dir_changed_counter = 0;

                                // Reverse the motor direction
                                if(motor_dir == MOTOR_DIR_LEFT)
                                {
                                        motor_dir = MOTOR_DIR_RIGHT;
                                        nrf_gpio_pin_clear(MOTOR_DIR_PIN);
                                }
                                else
                                {
                                        motor_dir = MOTOR_DIR_LEFT;
                                        nrf_gpio_pin_set(MOTOR_DIR_PIN);
                                }

                                //nrf_drv_pwm_uninit(&m_pwm0);
                                //motor_running = false;
                        }
                }
                else
                {
                        if(current_voltage > previous_voltage)
                        {
                                // Keep turning
                                ascending = true;
                        }
                        else if(current_voltage < previous_voltage)//- previous_voltage >
                        0.01)
                        {
                                // Reverse the motor direction
                                if(motor_dir == MOTOR_DIR_LEFT)
                                {
                                        motor_dir = MOTOR_DIR_RIGHT;
                                        nrf_gpio_pin_clear(MOTOR_DIR_PIN);
                                }
                                else
                                {
                                        motor_dir = MOTOR_DIR_LEFT;
                                        nrf_gpio_pin_set(MOTOR_DIR_PIN);
                                }
                        }
                }
        }

        previous_voltage = current_voltage;
}

/**@brief Function for dispatching a BLE stack event to all modules with a BLE stack event
handler.
 *
 * @details This function is called from the BLE Stack event interrupt handler after a BLE
 stack
 *          event has been received.
 *
 * @param[in] p_ble_evt  Bluetooth stack event.
 */
static void ble_evt_dispatch(ble_evt_t * p_ble_evt)
{
    dm_ble_evt_handler(p_ble_evt);
    ble_conn_params_on_ble_evt(p_ble_evt);
        ble_db_discovery_on_ble_evt(&m_ble_db_discovery, p_ble_evt);
```

```
    bsp_btn_ble_on_ble_evt(p_ble_evt);
    on_ble_evt(p_ble_evt);
    ble_advertising_on_ble_evt(p_ble_evt);
        ble_vmsr_c_on_ble_evt(&m_vmsr, p_ble_evt);
}


/**@brief Function for dispatching a system event to interested modules.
 *
 * @details This function is called from the System event interrupt handler after a system
 *          event has been received.
 *
 * @param[in] sys_evt  System stack event.
 */
static void sys_evt_dispatch(uint32_t sys_evt)
{
    pstorage_sys_event_handler(sys_evt);
    ble_advertising_on_sys_evt(sys_evt);
}


/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    uint32_t err_code;

    nrf_clock_lf_cfg_t clock_lf_cfg = NRF_CLOCK_LFCLKSRC;

    // Initialize the SoftDevice handler module.
    SOFTDEVICE_HANDLER_INIT(&clock_lf_cfg, NULL);

    ble_enable_params_t ble_enable_params;
    err_code = softdevice_enable_get_default_config(CENTRAL_LINK_COUNT,
                                                    PERIPHERAL_LINK_COUNT,
                                                    &ble_enable_params);
    APP_ERROR_CHECK(err_code);

    //Check the ram settings against the used number of links
    CHECK_RAM_START_ADDR(CENTRAL_LINK_COUNT,PERIPHERAL_LINK_COUNT);

    // Enable BLE stack.
    err_code = softdevice_enable(&ble_enable_params);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.
    err_code = softdevice_ble_evt_handler_set(ble_evt_dispatch);
    APP_ERROR_CHECK(err_code);

    // Register with the SoftDevice handler module for BLE events.
    err_code = softdevice_sys_evt_handler_set(sys_evt_dispatch);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling events from the BSP module.
 *
 * @param[in]   event   Event generated by button press.
 */
void bsp_event_handler(bsp_event_t event)
{
    uint32_t err_code;

    switch (event)
    {
        case BSP_EVENT_SLEEP:
            sleep_mode_enter();
            break;
```



```
        case BSP_EVENT_DISCONNECT:
            err_code = sd_ble_gap_disconnect(m_conn_handle,
            BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            if (err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;

        case BSP_EVENT_WHITELIST_OFF:
            err_code = ble_advertising_restart_without_whitelist();
            if (err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;
        default:
            break;
    }
}


/**@brief Function for handling the Device Manager events.
 *
 * @param[in] p_evt  Data associated to the device manager event.
 */
static uint32_t device_manager_evt_handler(dm_handle_t const * p_handle,
                                           dm_event_t const  * p_event,
                                           ret_code_t        event_result)
{
            APP_ERROR_CHECK(event_result);

            uint8_t err_code;

        switch(p_event->event_id)
        {
                case(DM_EVT_CONNECTION):
                    bsp_indication_set(BSP_INDICATE_CONNECTED);

                    m_conn_handle = p_event->event_param.p_gap_param->conn_handle;

                    m_dm_device_handle = (*p_handle);

                    // Discover peer's services.
          err_code = ble_db_discovery_start(&m_ble_db_discovery,
                                            p_event->event_param.p_gap_param->conn_handle);

                break;

                case(DM_EVT_DISCONNECTION):
                    bsp_indication_set(BSP_INDICATE_IDLE);

                    // When disconnected, starts to regulate and to scan
                    motor_regulation(0);
                    scan_start();
                break;
        }

#ifdef BLE_DFU_APP_SUPPORT
    if (p_event->event_id == DM_EVT_LINK_SECURED)
    {
        app_context_load(p_handle);
    }
#endif // BLE_DFU_APP_SUPPORT

    return NRF_SUCCESS;
}


/**@brief Function for the Device Manager initialization.
 *
```

```
 * @param[in] erase_bonds  Indicates whether bonding information should be cleared from
 *                         persistent storage during initialization of the Device Manager.
 */
static void device_manager_init(bool erase_bonds)
{
    uint32_t               err_code;
    dm_init_param_t        init_param = {.clear_persistent_data = erase_bonds};
    dm_application_param_t register_param;

    // Initialize persistent storage module.
    err_code = pstorage_init();
    APP_ERROR_CHECK(err_code);

    err_code = dm_init(&init_param);
    APP_ERROR_CHECK(err_code);

    memset(&register_param.sec_param, 0, sizeof(ble_gap_sec_params_t));

    register_param.sec_param.bond         = SEC_PARAM_BOND;
    register_param.sec_param.mitm         = SEC_PARAM_MITM;
    register_param.sec_param.lesc         = SEC_PARAM_LESC;
    register_param.sec_param.keypress     = SEC_PARAM_KEYPRESS;
    register_param.sec_param.io_caps      = SEC_PARAM_IO_CAPABILITIES;
    register_param.sec_param.oob          = SEC_PARAM_OOB;
    register_param.sec_param.min_key_size = SEC_PARAM_MIN_KEY_SIZE;
    register_param.sec_param.max_key_size = SEC_PARAM_MAX_KEY_SIZE;
    register_param.evt_handler            = device_manager_evt_handler;
    register_param.service_type           = DM_PROTOCOL_CNTXT_GATT_SRVR_ID;

    err_code = dm_register(&m_app_handle, &register_param);
    APP_ERROR_CHECK(err_code);
}

/**@brief Voltage Measurement Service Client Event Handler.
 */
void vmsr_c_evt_handler(ble_vmsr_c_t * p_vmsr_c, ble_vmsr_c_evt_t * p_evt)
{
        uint32_t err_code;

        switch(p_evt->evt_type)
        {
                case BLE_VMSR_C_EVT_DISCOVERY_COMPLETE:     // Voltage Service discovered
                    err_code = ble_vmsr_c_vm_notif_enable(p_vmsr_c);    // Activate
                    notification

                    APP_ERROR_CHECK(err_code);

                    break;
                case BLE_VMSR_C_EVT_MEAS_NOTIFICATION:
                    // Received value from notification
                    // Regulate the loop
                    motor_regulation(p_evt->params.voltage_measure);

                    break;
                case BLE_VMSR_C_EVT_MEAS_READ_RESP:
                    // Received readed value

                    break;
                default:
                    break;
        }
}

/**
 * @brief Voltage measurement collector initialization.
 */
static void vmsr_c_init(void)
{
        ble_vmsr_c_init_t vmsr_c_init_obj;

        vmsr_c_init_obj.evt_handler = vmsr_c_evt_handler;
```



```
        uint32_t err_code = ble_vmsr_c_init(&m_vmsr, &vmsr_c_init_obj);
        APP_ERROR_CHECK(err_code);
}



/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds  Will be true if the clear bonding button was pressed to wake
 the application up.
 */
static void buttons_leds_init(bool * p_erase_bonds)
{
    bsp_event_t startup_event;

    uint32_t err_code = bsp_init(BSP_INIT_LED | BSP_INIT_BUTTONS,
                                 APP_TIMER_TICKS(100, APP_TIMER_PRESCALER),
                                 bsp_event_handler);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_btn_ble_init(NULL, &startup_event);
    APP_ERROR_CHECK(err_code);

    *p_erase_bonds = (startup_event == BSP_EVENT_CLEAR_BONDING_DATA);
}

static void regulation_init()
{
        // Configuring the motor direction and step pins as output
        nrf_gpio_cfg_output(MOTOR_STEP_PIN);
        nrf_gpio_cfg_output(MOTOR_DIR_PIN);

        // Set both pins at 1
        nrf_gpio_pin_set(MOTOR_STEP_PIN);
        nrf_gpio_pin_set(MOTOR_DIR_PIN);

        // Control buttons 0 and 1 (port 13 and 14)
        nrf_gpio_cfg_sense_input(13, NRF_GPIO_PIN_PULLUP, NRF_GPIO_PIN_SENSE_LOW);
        nrf_gpio_cfg_sense_input(14, NRF_GPIO_PIN_PULLUP, NRF_GPIO_PIN_SENSE_LOW);

        motor_dir = MOTOR_DIR_LEFT;

        motor_running = false;

        previous_voltage = 0.0;

        motor_regulation(0);
}

static void db_disc_handler(ble_db_discovery_evt_t * evt_handler)
{
        ble_vmsr_on_db_disc_evt(&m_vmsr , evt_handler);
}

static void db_discovery_init(void)
{
        uint32_t err_code = ble_db_discovery_init(db_disc_handler);

        APP_ERROR_CHECK(err_code);
}

/**@brief Function for the Power manager.
 */
static void power_manage(void)
{
    uint32_t err_code = sd_app_evt_wait();
    APP_ERROR_CHECK(err_code);
}
```

```
/**@brief Function for application main entry.
 */
int main(void)
{
    bool erase_bonds;

    // Initialize.
        timers_init();
    buttons_leds_init(&erase_bonds);
    ble_stack_init();
    device_manager_init(erase_bonds);
    gap_params_init();
    conn_params_init();
    db_discovery_init();
        vmsr_c_init();
        regulation_init();

    // Start execution.
        scan_start();

    // Enter main loop.
    for (;;)
    {
       power_manage();
    }
}

/**
 * @}
 */
```

```c
/* Copyright (c) 2012 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

#include "ble_vmsr_c.h"
#include <string.h>
#include <stdio.h>

#include "ble_db_discovery.h"
#include "nordic_common.h"
#include "ble_srv_common.h"
#include "app_util.h"
#include "sdk_common.h"
#include "SEGGER_RTT.h"


#define TX_BUFFER_MASK       0x07                  /**< TX Buffer mask, must be a mask of
contiguous zeroes, followed by contiguous sequence of ones: 000...111. */
#define TX_BUFFER_SIZE       (TX_BUFFER_MASK + 1)  /**< Size of the send buffer, which is 1
higher than the mask. */
#define WRITE_MESSAGE_LENGTH BLE_CCCD_VALUE_LEN    /**< Length of the write message for
CCCD. */


typedef enum
{
    READ_REQ,      /**< Type identifying that this tx_message is a read request. */
    WRITE_REQ      /**< Type identifying that this tx_message is a write request. */
} tx_request_t;

/**@brief Structure for writing a message to the peer, i.e. CCCD.
 */
typedef struct
{
    uint8_t                  gattc_value[WRITE_MESSAGE_LENGTH];  /**< The message to write. */
    ble_gattc_write_params_t gattc_params;                       /**< The GATTC parameters
    for this message. */
} write_params_t;

/**@brief Structure for holding the data that will be transmitted to the connected central.
 */
typedef struct
{
    uint16_t     conn_handle;  /**< Connection handle to be used when transmitting this
    message. */
    tx_request_t type;         /**< Type of message. (read or write). */
    union
    {
        uint16_t       read_handle;  /**< Read request handle. */
        write_params_t write_req;    /**< Write request message. */
    } req;
} tx_message_t;

static tx_message_t  m_tx_buffer[TX_BUFFER_SIZE];  /**< Transmit buffer for the messages
that will be transmitted to the central. */
static uint32_t      m_tx_insert_index = 0;        /**< Current index in the transmit buffer
where the next message should be inserted. */
static uint32_t      m_tx_index        = 0;        /**< Current index in the transmit buffer
containing the next message to be transmitted. */

ble_uuid_t ble_uuid;

/**@brief Function for passing any pending request from the buffer to the stack.
```



```c
 */
static void tx_buffer_process(void)
{
    if (m_tx_index != m_tx_insert_index)
    {
        uint32_t err_code;

        if (m_tx_buffer[m_tx_index].type == READ_REQ)
        {
            err_code = sd_ble_gattc_read(m_tx_buffer[m_tx_index].conn_handle,
                                         m_tx_buffer[m_tx_index].req.read_handle,
                                         0);
        }
        else
        {
            err_code = sd_ble_gattc_write(m_tx_buffer[m_tx_index].conn_handle,

                                          &m_tx_buffer[m_tx_index].req.write_req.gattc_params)
                                          ;
        }
        if (err_code == NRF_SUCCESS)
        {
            m_tx_index++;
            m_tx_index &= TX_BUFFER_MASK;
        }
        else
        {
                        //error
        }
    }
}

/**@brief Disconnect event handler.
 *
 * @param[in]   p_vmsr      Voltage Measurement Service structure.
 * @param[in]   p_ble_evt   Event received from the BLE stack.
 */
static void on_disconnect(ble_vmsr_c_t * p_vmsr, ble_evt_t * p_ble_evt)
{
    UNUSED_PARAMETER(p_ble_evt);
    p_vmsr->conn_handle                                    = BLE_CONN_HANDLE_INVALID;
        p_vmsr->peer_vmsr_db.vm_cccd_handle     = BLE_GATT_HANDLE_INVALID;
        p_vmsr->peer_vmsr_db.vm_handle              = BLE_GATT_HANDLE_INVALID;
}


/**@brief     Function for handling Handle Value Notification received from the SoftDevice.
 *
 * @details   This function will handle the Handle Value Notification received from the
 SoftDevice
 *            and checks if it is a notification of the Voltage Measurement from the peer. If
 *            so, this function will decode the battery level measurement and send it to the
 *            application.
 *
 * @param[in] p_ble_vmsr_c  Pointer to the Voltage Client structure.
 * @param[in] p_ble_evt     Pointer to the BLE event received.
 */
static void on_hvx(ble_vmsr_c_t * p_ble_vmsr_c, const ble_evt_t * p_ble_evt)
{
    // Check if the event is on the link for this instance
    if (p_ble_vmsr_c->conn_handle != p_ble_evt->evt.gattc_evt.conn_handle)
    {
        return;
    }

    // Check if this notification is a voltage measurement notification.
    if (p_ble_evt->evt.gattc_evt.params.hvx.handle == p_ble_vmsr_c->peer_vmsr_db.vm_handle)
    {
                float voltage;

                // Get the float value from the receive data (4 bytes)
```

```
                memcpy(&voltage, p_ble_evt->evt.gattc_evt.params.hvx.data, sizeof(voltage));

                ble_vmsr_c_evt_t ble_vmsr_c_evt;
                ble_vmsr_c_evt.conn_handle = p_ble_evt->evt.gattc_evt.conn_handle;
                ble_vmsr_c_evt.evt_type    = BLE_VMSR_C_EVT_MEAS_NOTIFICATION;

                ble_vmsr_c_evt.params.voltage_measure = voltage;

                p_ble_vmsr_c->evt_handler(p_ble_vmsr_c, &ble_vmsr_c_evt);


    }
}

void ble_vmsr_on_db_disc_evt(ble_vmsr_c_t * p_ble_vmsr_c, const ble_db_discovery_evt_t *
p_evt)
{
      // Check if the Voltage Service was discovered.
    if (p_evt->evt_type == BLE_DB_DISCOVERY_COMPLETE
        &&
        p_evt->params.discovered_db.srv_uuid.uuid == VMSR_UUID_SERVICE
        &&
        p_evt->params.discovered_db.srv_uuid.type == p_ble_vmsr_c->uuid_type)
    {
        // Find the CCCD Handle of the Voltage Measurement characteristic.
        uint8_t i;

        ble_vmsr_c_evt_t evt;
        evt.evt_type    = BLE_VMSR_C_EVT_DISCOVERY_COMPLETE;
        evt.conn_handle = p_evt->conn_handle;
        for (i = 0; i < p_evt->params.discovered_db.char_count; i++)
        {
            if (p_evt->params.discovered_db.charateristics[i].characteristic.uuid.uuid ==
                VMSR_UUID_VOLTAGE_VALUE_CHAR)
            {
                // Found Voltage Measurement characteristic. Store CCCD handle and break.
                evt.params.vmsr_db.vm_cccd_handle =
                    p_evt->params.discovered_db.charateristics[i].cccd_handle;
                evt.params.vmsr_db.vm_handle =
                    p_evt->params.discovered_db.charateristics[i].characteristic.handle_value;
                break;
            }
        }

        //If the instance has been assigned prior to db_discovery, assign the db_handles
        if(p_ble_vmsr_c->conn_handle != BLE_CONN_HANDLE_INVALID)
        {
            if ((p_ble_vmsr_c->peer_vmsr_db.vm_cccd_handle == BLE_GATT_HANDLE_INVALID)&&
                (p_ble_vmsr_c->peer_vmsr_db.vm_handle      == BLE_GATT_HANDLE_INVALID))
            {
                p_ble_vmsr_c->peer_vmsr_db = evt.params.vmsr_db;
            }
        }
        p_ble_vmsr_c->evt_handler(p_ble_vmsr_c, &evt);
    }
    else
    {
        // Discovery failed
    }
}

void ble_vmsr_c_on_ble_evt(ble_vmsr_c_t * p_vmsr_c, ble_evt_t * p_ble_evt)
{
        if ((p_vmsr_c == NULL) || (p_ble_evt == NULL))
    {
        return;
    }

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GATTC_EVT_HVX:
```



```
            on_hvx(p_vmsr_c, p_ble_evt);                        // Read notification
            break;
        case BLE_GAP_EVT_DISCONNECTED:
            on_disconnect(p_vmsr_c, p_ble_evt);     // Disconnection
            break;

        default:
            break;
    }
}


/**@brief Function for creating a message for writing to the CCCD.
 */
static uint32_t cccd_configure(uint16_t conn_handle, uint16_t handle_cccd, bool
notification_enable)
{
    tx_message_t * p_msg;
    uint16_t       cccd_val = notification_enable ? BLE_GATT_HVX_NOTIFICATION : 0;

    p_msg              = &m_tx_buffer[m_tx_insert_index++];
    m_tx_insert_index &= TX_BUFFER_MASK;

    p_msg->req.write_req.gattc_params.handle   = handle_cccd;
    p_msg->req.write_req.gattc_params.len      = WRITE_MESSAGE_LENGTH;
    p_msg->req.write_req.gattc_params.p_value  = p_msg->req.write_req.gattc_value;
    p_msg->req.write_req.gattc_params.offset   = 0;
    p_msg->req.write_req.gattc_params.write_op = BLE_GATT_OP_WRITE_REQ;
    p_msg->req.write_req.gattc_value[0]        = LSB_16(cccd_val);
    p_msg->req.write_req.gattc_value[1]        = MSB_16(cccd_val);
    p_msg->conn_handle                         = conn_handle;
    p_msg->type                                = WRITE_REQ;

    tx_buffer_process();
    return NRF_SUCCESS;
}


uint32_t ble_vmsr_c_init(ble_vmsr_c_t * p_vmsr_c, const ble_vmsr_c_init_t * p_vmsr_c_init)
{
        VERIFY_PARAM_NOT_NULL(p_vmsr_c);
    VERIFY_PARAM_NOT_NULL(p_vmsr_c_init);

    uint32_t   err_code;

    // Initialize service structure
    p_vmsr_c->conn_handle                                 = BLE_CONN_HANDLE_INVALID;
    p_vmsr_c->evt_handler                                     = p_vmsr_c_init->evt_handler;
        p_vmsr_c->peer_vmsr_db.vm_cccd_handle   = BLE_GATT_HANDLE_INVALID;
        p_vmsr_c->peer_vmsr_db.vm_handle                = BLE_GATT_HANDLE_INVALID;

    // Add base UUID to softdevice's internal list.
    ble_uuid128_t base_uuid = VMSR_UUID_BASE;
    err_code = sd_ble_uuid_vs_add(&base_uuid, &p_vmsr_c->uuid_type);
    if (err_code != NRF_SUCCESS)
    {
        return err_code;
    }

    ble_uuid.type = p_vmsr_c->uuid_type;
    ble_uuid.uuid = VMSR_UUID_SERVICE;

    return ble_db_discovery_evt_register(&ble_uuid);
}

uint32_t ble_vmsr_c_vm_notif_enable(ble_vmsr_c_t * p_ble_vmsr_c)
{
    VERIFY_PARAM_NOT_NULL(p_ble_vmsr_c);

    if (p_ble_vmsr_c->conn_handle == BLE_CONN_HANDLE_INVALID)
    {
```

```
        return NRF_ERROR_INVALID_STATE;
    }

    return cccd_configure(p_ble_vmsr_c->conn_handle,
    p_ble_vmsr_c->peer_vmsr_db.vm_cccd_handle, true);
}


uint32_t ble_vmsr_c_vm_read(ble_vmsr_c_t * p_ble_vmsr_c)
{
    VERIFY_PARAM_NOT_NULL(p_ble_vmsr_c);

    if (p_ble_vmsr_c->conn_handle == BLE_CONN_HANDLE_INVALID)
    {
        return NRF_ERROR_INVALID_STATE;
    }

    tx_message_t * msg;

    msg                  = &m_tx_buffer[m_tx_insert_index++];
    m_tx_insert_index   &= TX_BUFFER_MASK;

    msg->req.read_handle = p_ble_vmsr_c->peer_vmsr_db.vm_handle;
    msg->conn_handle     = p_ble_vmsr_c->conn_handle;
    msg->type            = READ_REQ;

    tx_buffer_process();
    return NRF_SUCCESS;
}

uint32_t ble_vmsr_c_handles_assign(ble_vmsr_c_t *    p_ble_vmsr_c,
                                   uint16_t          conn_handle,
                                   ble_vmsr_c_db_t * p_peer_handles)
{
    VERIFY_PARAM_NOT_NULL(p_ble_vmsr_c);

    p_ble_vmsr_c->conn_handle = conn_handle;
    if (p_peer_handles != NULL)
    {
        p_ble_vmsr_c->peer_vmsr_db = *p_peer_handles;
    }
    return NRF_SUCCESS;
}
```

```
/* Copyright (c) 2012 Nordic Semiconductor. All Rights Reserved.
 *
 * The information contained herein is property of Nordic Semiconductor ASA.
 * Terms and conditions of usage are described in detail in NORDIC
 * SEMICONDUCTOR STANDARD SOFTWARE LICENSE AGREEMENT.
 *
 * Licensees are granted free, non-transferable use of the information. NO
 * WARRANTY of ANY KIND is provided. This heading must NOT be removed from
 * the file.
 *
 */

/** @file
 *
 * @brief Voltage Measurement Service module.
 *
 * @details This module implements the Voltage Measurement Service with the Voltage
 characteristic.
 *          During initialization it adds the Voltage Measurement Service and Voltage
 characteristic
 *          to the BLE stack database.
 *
 *
 * @note Attention!
 *  To maintain compliance with Nordic Semiconductor ASA Bluetooth profile
 *  qualification listings, this section of source code must not be modified.
 */

#ifndef BLE_VMSR_H__
#define BLE_VMSR_H__

#include <stdint.h>
#include "ble.h"
#include "ble_srv_common.h"
#include "ble_db_discovery.h"

#define VMSR_UUID_BASE                      {0x55, 0xB8, 0x6C, 0xCB, 0x10, 0x03, 0x24, 0xAF,
0xA7, 0x43, 0x4D, 0xE0, 0x00, 0x00, 0x6C, 0xC9}
#define VMSR_UUID_COMPLETE_128BIT       {0x55, 0xB8, 0x6C, 0xCB, 0x10, 0x03, 0x24, 0xAF,
0xA7, 0x43, 0x4D, 0xE0, 0x00, 0x00, 0x00, 0x20, 0x6C, 0xC9}
#define VMSR_UUID_SERVICE                   0x2000
#define VMSR_UUID_VOLTAGE_VALUE_CHAR    0x2001

// Forward declaration of the ble_vmsr_t type.
typedef struct ble_vmsr_c_s ble_vmsr_c_t;

typedef struct ble_vmsr_c_evt_s ble_vmsr_c_evt_t;

/**@brief Voltage Measurement Service event handler type. */
typedef void (*ble_vmsr_c_evt_handler_t) (ble_vmsr_c_t * p_vmsr, ble_vmsr_c_evt_t * p_evt);

/**@brief Voltage Measurement Service init structure. */
typedef struct
{
    ble_vmsr_c_evt_handler_t evt_handler; /**< Called when the voltage value is updated. */
} ble_vmsr_c_init_t;

/**@brief Voltage Service Client event type. */
typedef enum
{
    BLE_VMSR_C_EVT_DISCOVERY_COMPLETE,  /**< Event indicating that the Voltage Service has
    been discovered at the peer. */
    BLE_VMSR_C_EVT_MEAS_NOTIFICATION,   /**< Event indicating that a notification of the
    Voltage Measurement characteristic has been received from the peer. */
    BLE_VMSR_C_EVT_MEAS_READ_RESP       /**< Event indicating that a read response on
    Voltage Measurement characteristic has been received from peer. */
} ble_vmsr_c_evt_type_t;

/**@brief Structure containing the handles related to the Voltage Service found on the peer.
*/
typedef struct
```



```
{
    uint16_t                vm_cccd_handle;  /**< Handle of the CCCD of the Voltage
    Measurement characteristic. */
    uint16_t                vm_handle;       /**< Handle of the Voltage Measurement
    characteristic as provided by the SoftDevice. */
} ble_vmsr_c_db_t;

/**@brief Voltage Service Client Event structure. */
struct ble_vmsr_c_evt_s
{
    ble_vmsr_c_evt_type_t evt_type;  /**< Event Type. */
    uint16_t conn_handle;           /**< Connection handle relevent to this event.*/
    union
    {
        ble_vmsr_c_db_t     vmsr_db;                /**< Voltage Service related handles
        found on the peer device. This will be filled if the evt_type is @ref
        BLE_VMSR_C_EVT_DISCOVERY_COMPLETE.*/
        float                   voltage_measure;    /**< Voltage measurement received from
        peer. This field will be used for the events @ref BLE_VMSR_C_EVT_MEAS_NOTIFICATION
        and @ref BLE_VMSR_C_EVT_MEAS_READ_RESP.*/
    } params;
} ;

/**@brief Voltage Measurement Service structure. This contains various status information
for the service. */
struct ble_vmsr_c_s
{
  uint16_t                                    conn_handle;
    uint16_t                                    service_handle;
    uint8_t                                                     uuid_type;
    ble_vmsr_c_db_t                                         peer_vmsr_db;
    ble_vmsr_c_evt_handler_t                            evt_handler;
};

/**@brief Function for initializing the Voltage Measurement Service.
 *
 * @param[out]  p_vmsr      Voltage Measurement Service structure. This structure will have
 to be supplied by
 *                          the application. It will be initialized by this function, and
 will later
 *                          be used to identify this particular service instance.
 * @param[in]   p_vmsr_init  Information needed to initialize the service.
 *
 * @return      NRF_SUCCESS on successful initialization of service, otherwise an error code.
 */
uint32_t ble_vmsr_c_init(ble_vmsr_c_t * p_vmsr, const ble_vmsr_c_init_t * p_vmsr_init);


/**@brief     Function for handling events from the database discovery module.
 *
 * @details   Call this function when getting a callback event from the DB discovery modue.
 *            This function will handle an event from the database discovery module, and
 determine
 *            if it relates to the discovery of Battery service at the peer. If so, it will
 *            call the application's event handler indicating that the Battery service has
 been
 *            discovered at the peer. It also populates the event with the service related
 *            information before providing it to the application.
 *
 * @param     p_ble_bas_c Pointer to the Battery Service client structure.
 * @param[in] p_evt Pointer to the event received from the database discovery module.
 *
 */
void ble_vmsr_on_db_disc_evt(ble_vmsr_c_t * p_ble_vmsr_c, const ble_db_discovery_evt_t *
p_evt);

/**@brief Function for handling the Application's BLE Stack events.
 *
 * @details Handles all events from the BLE stack of interest to the Voltage Measurement
 Service.
 *
```

```
 * @param[in]   p_vmsr      Voltage Measurement Service structure.
 * @param[in]   p_ble_evt   Event received from the BLE stack.
 */
void ble_vmsr_c_on_ble_evt(ble_vmsr_c_t * p_vmsr, ble_evt_t * p_ble_evt);

/**@brief Function for sending a voltage value notification.
 *
 * @param[in] p_vmsr        Voltage Measurement Service structure.
 * @param[in] new_value         New voltage value.
 *
 * @retval NRF_SUCCESS If the notification was sent successfully. Otherwise, an error code
 is returned.
 */
uint32_t ble_vmsr_voltage_update(ble_vmsr_c_t * p_vmsr, float voltage_value);


/**@brief   Function for enabling notifications on the Voltage Measurement characteristic.
 *
 * @details This function will enable to notification of the Voltage Measurement
 characteristic at the
 *          peer by writing to the CCCD of the Voltage Measurement Characteristic.
 *
 * @param   p_ble_vmsr_c Pointer to the Voltage Measurement client structure.
 *
 * @retval  NRF_SUCCESS     If the SoftDevice has been requested to write to the CCCD of the
 peer.
 *          NRF_ERROR_NULL  Parameter is NULL.
 *                          Otherwise, an error code returned by the SoftDevice API @ref
 *                          sd_ble_gattc_write.
 */
uint32_t ble_vmsr_c_vm_notif_enable(ble_vmsr_c_t * p_ble_vmsr_c);


/**@brief     Function for assigning handles to a this instance of vmsr_c.
 *
 * @details   Call this function when a link has been established with a peer to
 *            associate this link to this instance of the module. This makes it
 *            possible to handle several link and associate each link to a particular
 *            instance of this module. The connection handle and attribute handles will be
 *            provided from the discovery event @ref BLE_VMSR_C_EVT_DISCOVERY_COMPLETE.
 *
 * @param[in] p_ble_vmsr_c    Pointer to the Battery client structure instance to associate.
 * @param[in] conn_handle    Connection handle to associated with the given Battery Client
 Instance.
 * @param[in] p_peer_handles Attribute handles on the BAS server you want this BAS client to
 *                           interact with.
 */
uint32_t ble_vmsr_c_handles_assign(ble_vmsr_c_t *    p_ble_vmsr_c,
                                   uint16_t         conn_handle,
                                   ble_vmsr_c_db_t * p_peer_handles);


#endif // BLE_VMSR_H__

/** @} */
```

Comment	Description	Manufacturer	Model	Quantity	Prix unité	Prix total
100 uF	Tantalum Capacitor	VISHAY	298W107X0010Q2T	3	1.95	5.85
1 uF	Tantalum Capacitor	VISHAY	TP8M105M010C	1	1.28	1.28
	Bridge Quad Schottky Redresser	AVAGO	HSMS-2828	1	1.58	1.58
	High Performance Single 150mA LDO	MICREL	MIC5317-3.0YM5	1	0.53	0.53
6.8 V	Zener Diode	PANASONIC	DZ2J068M0L	1	0.15	0.15
	NEMA17 - Stepper-Motor	Act-Motor	17HS5413	1	29.9	29.9
	Bracket 90 for Nema17 Motor	-	-	1	4.9	4.9
	Big Easy Driver	SparkFun	v.1.2	1	24.9	24.9
	Total					69.09