|  |
| --- |
| Heig-vd |
| VIRTUOSO |
| Travail de bachelor |

|  |
| --- |
| Etudiant : Quentin Salomon  Professeur : François Birling  09/04/2016 |



Table des matières

[2 Enoncé 2](#_Toc454143194)

[3 Clause de confidentialité 2](#_Toc454143195)

[4 Résumé 2](#_Toc454143196)

[5 Introduction 2](#_Toc454143197)

[6 Pré-étude 2](#_Toc454143198)

[6.1 Cahier des charges 2](#_Toc454143199)

[6.1.1 Interface utilisateurs 2](#_Toc454143200)

[6.1.2 Programme de traitement des fichiers midi 5](#_Toc454143201)

[6.2 Comparatif des plateformes 6](#_Toc454143202)

[6.3 Evaluation des risques 7](#_Toc454143203)

[6.3.1 I2C 7](#_Toc454143204)

[6.3.2 Midi 7](#_Toc454143205)

[6.3.3 Performance iot 8](#_Toc454143206)

[6.3.4 Woopsa 8](#_Toc454143207)

[6.3.5 Concept 8](#_Toc454143208)

[6.4 Choix du matériel 8](#_Toc454143209)

[6.5 UML 9](#_Toc454143210)

[7 Projet 9](#_Toc454143211)

[7.1 Protocole de communication 9](#_Toc454143212)

[7.1.1 Composition des messages 9](#_Toc454143213)

[7.1.2 Types de messages 9](#_Toc454143214)

[7.2 Serial 9](#_Toc454143215)

[7.3 I2C 10](#_Toc454143216)

[7.4 Programme de conversion 11](#_Toc454143217)

[7.4.1 Conversion midi 🡺 XML 11](#_Toc454143218)

[7.4.2 Intensité des notes 11](#_Toc454143219)

[7.4.3 Tempos de la partition 11](#_Toc454143220)

[7.5 Programme Xylophone 11](#_Toc454143221)

[7.6 Visuel interface 11](#_Toc454143222)

[7.6.1 Prototype 11](#_Toc454143223)

[7.6.2 Possibilités offertes par concept 11](#_Toc454143224)

[7.6.3 Visuel final 11](#_Toc454143225)

[7.7 Serveur WEB 11](#_Toc454143226)

[8 Conclusion 11](#_Toc454143227)

[9 Liste des références 11](#_Toc454143228)

[10 Dictionnaire de symboles et abréviations 12](#_Toc454143229)

[11 Figures 12](#_Toc454143230)

[12 Journal de travail 12](#_Toc454143231)

[13 Annexes 12](#_Toc454143232)

# Enoncé

La HEIG-VD a déjà réalisé un robot Xylophoniste anthropomorphe, appelé Virtuoso. Du fait de sa structure, il peut jouer à une vitesse proche de l'humain.

Pour pouvoir jouer la musique à une rapidité absolument impressionnante, ce nouveau travail de diplôme a pour but d'automatiser un autre xylophone de 37 touches en l'équipant d'un actionneur électromécanique en face de chaque touche, et en pilotant l'exécution musicale par un logiciel embarqué d'une part, et un logiciel PC d'autre part permettant de préparer et transmettre les partitions.

En complément, la réalisation d'une interface utilisateur permettant de contrôler le système depuis un smartphone et de sélectionner le morceau à jouer peut s'inscrire dans ce travail de bachelor.

# Clause de confidentialité

-

# Résumé

?

# Introduction



Figure : Virutoso

Dans le cadre des travaux de bachelor à la heig-vd, nous avons dû choisir un projet.

Le projet choisi consiste en un xylophone automatisé nommé Virtuoso. Le but est de pouvoir lui faire jouer à peu près n’importe qu’elle partition et que la musicalité soit parfaite. Gérant ainsi les tempos de la musique ainsi que l’intensité de chaque note. Dans cette optique, les partitions doivent pouvoir être modifier et converties afin de les adapter au xylophone et ses 3 octaves. Son autre but est de faire la promotion de l’heig-vd aux portes ouvertes. Pour que les visiteurs aient une interactivité avec, un wifi avec un page web est prévue.

Dans l’état, le Virutoso est composé d’un boitier électronique gérant 37 actuateurs correspondant chacun à une touche. Il a aussi des leds permettant d’éclairés les actuateurs. Du boitier électronique, il en sort 4 fils. Deux d’entre eux sont la masse et le 5 volts et les deux autres donne accès à une communication en i2c.

À l’arrivée du temps impartit pour ce projet, il est composé d’un NUC (mini pc) avec un écran tactile. Communiquant en USB vers un Arduino Mega. L’Arduino est là pour faire la gestion du temps réel et la communication en i2c à la partie électronique. Le NUC fait la gestion HMI composé d’un programme C# avec son interface graphique. Il gère aussi l’émission du Wifi et fait office de serveur web. Dû au manque de temps, la modification des fichiers midi se fait par le biais d’un tierce logiciel. Néanmoins, la conversion est gérée par un autre programme C#.

# Pré-étude

## Cahier des charges

Pour la réalisation du projet, il a fallu définir un cahier des charges. Celui-ci est là pour savoir qu’est-ce qui devra être implémenté et qu’est-ce qu’il devrait possible d’avoir si le temps n’est pas manquant.

### Démarche

Afin de réaliser le cahier des charges, un diagramme des cas d’utilisations a tout d’abord été fait et modifié afin de correspondre aux attentes voulues. Pour bien séparé le Virtuoso du programme pour la conversion des fichiers, deux diagrammes distincts ont été réalisé. Ceux-ci sont montrés aux Figure 2 et Figure 3. Puis à l’aide de ces diagrammes, pour chaque cas d’utilisation des charges ont été définies donnant lieu au cahier des charges.

### Interface utilisateurs



Figure : Diagramme cas d'utilisation Virtuoso

#### Interface web

**Superviser l’exécution**

|  |  |
| --- | --- |
| 10.10 | L’interface web doit permettre de suivre l’exécution sur l’écran en visualisant le titre du morceau joué ainsi que sa position dans la playlist. |
| 10.20 | L’interface web devrait permettre de voir un aperçu du Virtuoso avec chaque note et montrer quelles notes sont jouées en directe. |

**Ajouter partition à la playlist**

|  |  |
| --- | --- |
| 20.10 | L’interface web devrait permettre d’ajouter une partition à la playlist. |

**Demande d’ajout de partition inexistante**

|  |  |
| --- | --- |
| 30.10 | L’interface web devrait permettre de faire une demande d’ajout d’une partition qui n’est pas dans la liste des partitions enregistrées. Cette demande pourra être vue par l’animateur et être traitée ultérieurement. |

#### Interface NUC

**Superviser l’exécution**

|  |  |
| --- | --- |
| 10.10 | L’écran tactile doit permettre de suivre l’exécution sur l’écran en visualisant le titre du morceau joué ainsi que sa position dans la playlist. |
| 10.20 | L’écran tactile devrait permettre de voir un aperçu du Virtuoso avec chaque note et montrer quelles notes sont jouées en directe. |

**Effacer playlist**

|  |  |
| --- | --- |
| 20.10 | Il doit être possible de supprimer la playlist. |

**Pause Start/Stop partition**

|  |  |
| --- | --- |
| 30.10 | L’écran tactile doit permettre de mettre sur pause la partition qui est jouée. |
| 30.20 | L’écran tactile doit permettre de la lecture d’une partition. |
| 30.30 | Le start permettra de jouer, depuis le début, la partition de la playlist qui a été stoppée. |

**Test de toutes les notes**

|  |  |
| --- | --- |
| 40.10 | L’écran tactile doit permettre de lancer un programme test qui actionne chaque note à la suite. |

**Réglage du temps de frappe des notes**

|  |  |
| --- | --- |
| 50.10 | L’interface doit permettre de régler le temps de frappe de chaque note indépendamment. |

**Changer la vitesse de lecture**

|  |  |
| --- | --- |
| 60.10 | Il devrait être possible de changer la vitesse de lecture du Virtuoso. |

**Partition en boucle**

|  |  |
| --- | --- |
| 70.10 | L’interface devrait permettre de jouer une partition en boucle. |

**Visualiser xylophone**

|  |  |
| --- | --- |
| 80.10 | Il devrait être possible de voir chaque note du xylophone en directe (voir les notes qui s’active). |

### Programme de traitement des fichiers midi



Figure : diagramme cas d’utilisations programme de conversion des fichiers

**Charger un fichier midi**

|  |  |
| --- | --- |
| 10.10 | Le programme doit permettre de charger un fichier au format midi et d’en extraire les notes. |

**Visualisation la partition**

|  |  |
| --- | --- |
| 20.10 | Le programme doit pouvoir visualiser les notes, avec leur hauteur et leur tic, de la partition chargée. Les notes venant de différents canaux (instruments) doivent être distinguées. |

**Arranger la partition**

|  |  |
| --- | --- |
| 30.10 | Le programme permettra de transposer toute la partition d’un nombre de ton(s) choisi(s). |
| 30.11 | Il doit pouvoir transposer chaque instrument d’un nombre de ton(s) choisi(s). |
| 30.20 | Il pourra agir sur chaque note indépendamment pour modifier sa hauteur ou son tic. |
| 30.30 | Le programme devrait permettre de régler le tempo de la partition. |

**Rogner la partition**

|  |  |
| --- | --- |
| 40.10 | Il devrait être possible de sélectionner la zone que l’on va garder sur une partition. |

**Ecouter la partition**

|  |  |
| --- | --- |
| 50.10 | Le programme devrait pouvoir écouter la partition. |

**Sauver la partition**

|  |  |
| --- | --- |
| 60.10 | Le programme doit pouvoir sauver une partition sous forme XML. |

**Envoyer la partition au Virtuoso**

|  |  |
| --- | --- |
| 50.10 | Le programme devrait permettre de transmettre une partition au Virtuoso à l’aide de Woopsa. |

## Evaluation des risques

### Communication I2C

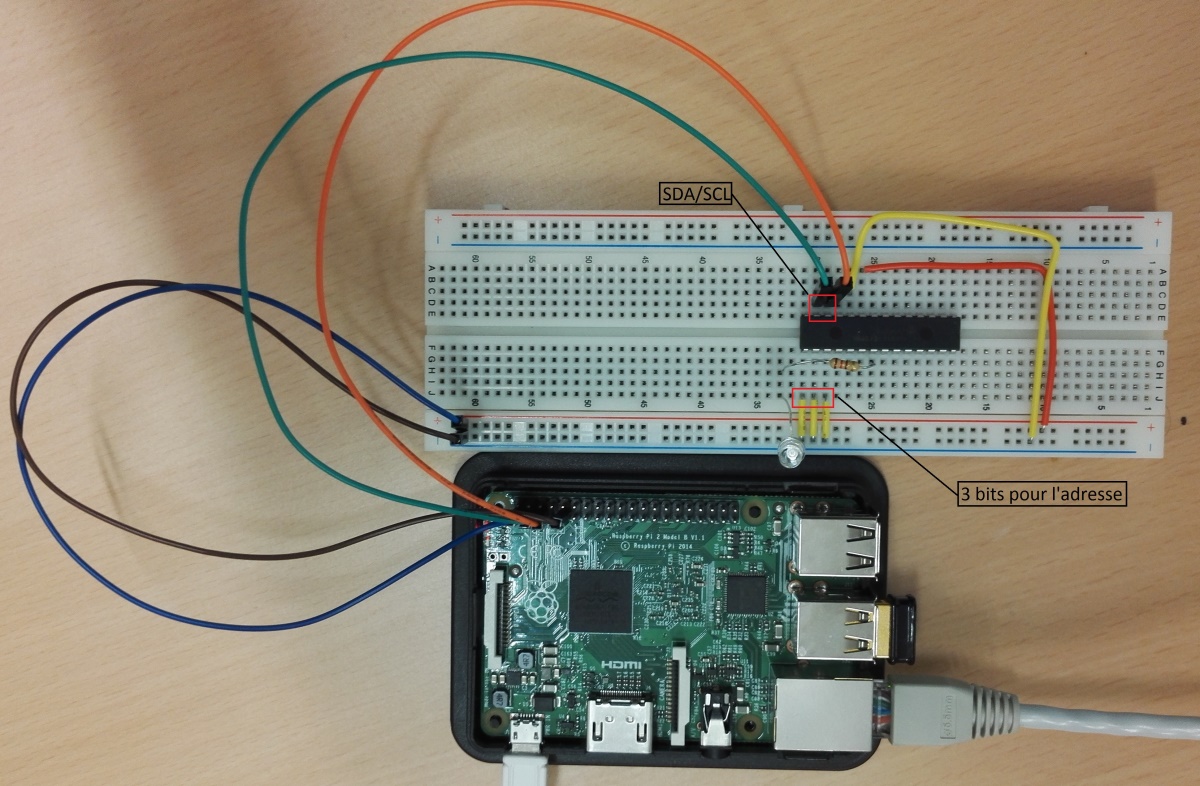
#### Problème

Comme la partie électronique du Virtuoso était implémentée avec des MCP23017 et prévu pour communiquer en I2C, le choix de la communication pour piloter les actuateurs s’est donc fait d’office.

#### Démarche

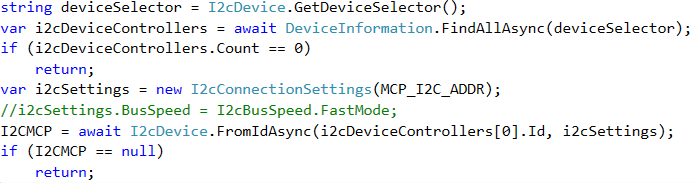
Ainsi pour tester l’I2C, un programme a été codé sur un Raspberry. Celui-ci fut créé juste pour tester l’I2C et savoir comment l’implémenté sur Windows IoT.

L’image suivante représente le câblage implémenté pour le programme de test. Avec cette configuration, nous avons l’adresse 0x20 pour le MCP décrite juste après.



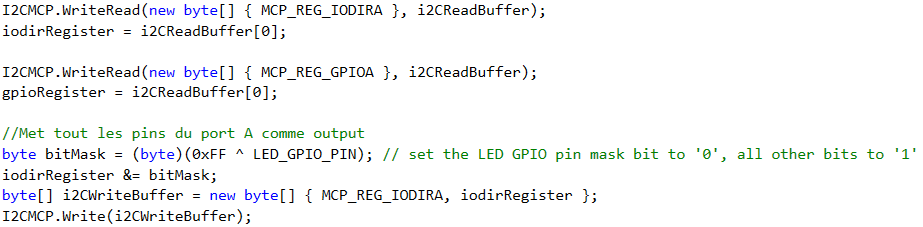
L’adressage pour le MCP23017 est sur 7 bits. Les 4 msb sont fixés et les 3 derniers peuvent être choisis. Dans notre cas, les 3 lsb valent 0, nous avons comme adresse 010 00002 = 0x20.

L’utilisation de l’I2C est plutôt simple car il suffit de trouver les bons registres à modifier. Pour le MCP23017, le registre des directions IO A est 0x00 et celui des valeurs IO A est 0x14. Ces registres nous permettent d’initialiser l’I2C puis de communiquer. Le programme initialise l’I2C :



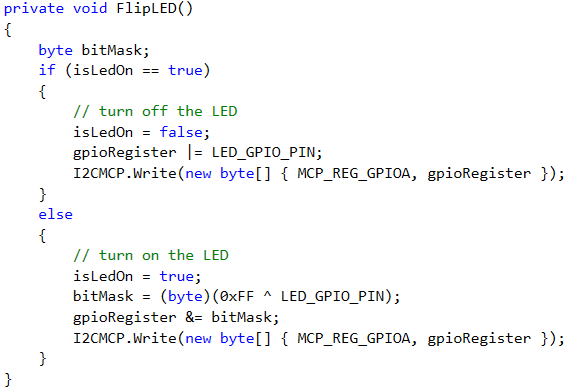
En donnant l’adresse du MCP (« MCP\_I2C\_ADDR » qui est 0x20), on obtient une variable « I2cDevice » qui nous permet d’écrire au MCP.

Puis le programme initialise les registres du MCP :



Avec les registres du MCP MCP\_REG\_IODIRA = 0x00 et MCP\_REG\_GPIOA = 0x14.

Et ensuite la communication est simple, on donne l’adresse et la valeur du registre que l’on veut écrire :



### Type de fichier convertit pour le Virtuoso : Midi

#### Problème

Pour « alimenter » le Virtuoso en musique, il faut trouver un type de fichier à convertir. Dans l’idéal, il faut un type qui aurait une grande « bibliothèque » de musique. De plus, on a besoin de fichiers qui sont sous forme de partition pour récupérer facilement les notes (et d’autre information).

#### Démarche

Après quelques recherches et propositions, le midi a été choisi pour les raisons suivantes :

Pour les fichiers son, le midi allait parfaitement étant donné qu’il permet d’obtenir chaque note jouée et de les séparées par instrument (les channels). De plus, il y a énormément de fichier midi téléchargeable sur internet, de quoi couvrir tous les goûts.

Pour traiter les fichiers midi en C#, une librairie a été trouvée. Elle se nomme miditoolkit et à l’avantage d’être libre d’utilisation. Elle dispose de toute les fonctionnalités nécessaires pour lire les fichiers midi.

Un des points faibles est qu’il n’y a que très peu d’exemples disponible pour son utilisation. Par conséquent, cela a demandé plus d’investissement en temps pour son implémentation.

Après avoir trouvé cette librairie, en voyant le peu d’exemple disponible et testant rapidement, des recherches rapides ont été faite pour regarder s’il était possible de faire une librairie maison rapidement. Il en est venu à la conclusion qu’il serait plus simple de rester sur la librairie existante.

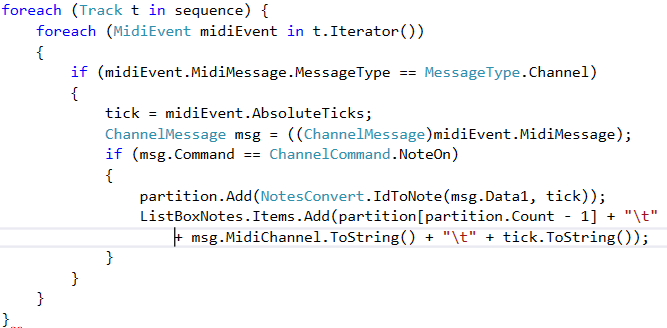
#### Test de la librairie miditoolkit

Afin de s’assurer que la librairie fonctionne bien et qu’elle offre les fonctionnalités nécessaires, un programme test a été réalisé.

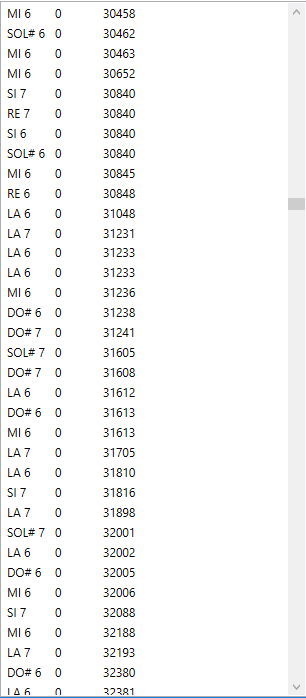
Le programme commence par charger le fichier midi avec un chemin en brut dans le code :



Puis il parcourt le fichier et charge les données dans une ListBox :



Le code ci-dessus explore chaque note de la partition en enregistrant la note dans l’instance « partition » et l’affiche à l’aide de la ListBox. Il affiche aussi le canal pour chaque note ainsi que le tick.



Pour l’affichage, le programme implémente une interface graphique très rudimentaire permettant de visualiser les résultats avec 3 colonnes (la note, le canal et le tick).

### Windows IoT

#### Qu’est-ce que c’est ?

Windows IoT est une version de Windows simplifiée pour les microcontrôleurs comme le Raspberry pi. Il a l’avantage de pouvoir développer à l’aide de Visual studio sur un pc et de compiler facilement sur le Raspberry. De plus, pour son utilisation, toutes les démarches sont décrites sur le site de Windows (il suffit de chercher un tout petit peu).

#### Performances de Windows IoT

##### Problème

Pour savoir si les performances de Windows IoT seraient concluantes pour le pilotage du xylophone, des tests ont été effectué.

##### Démarche

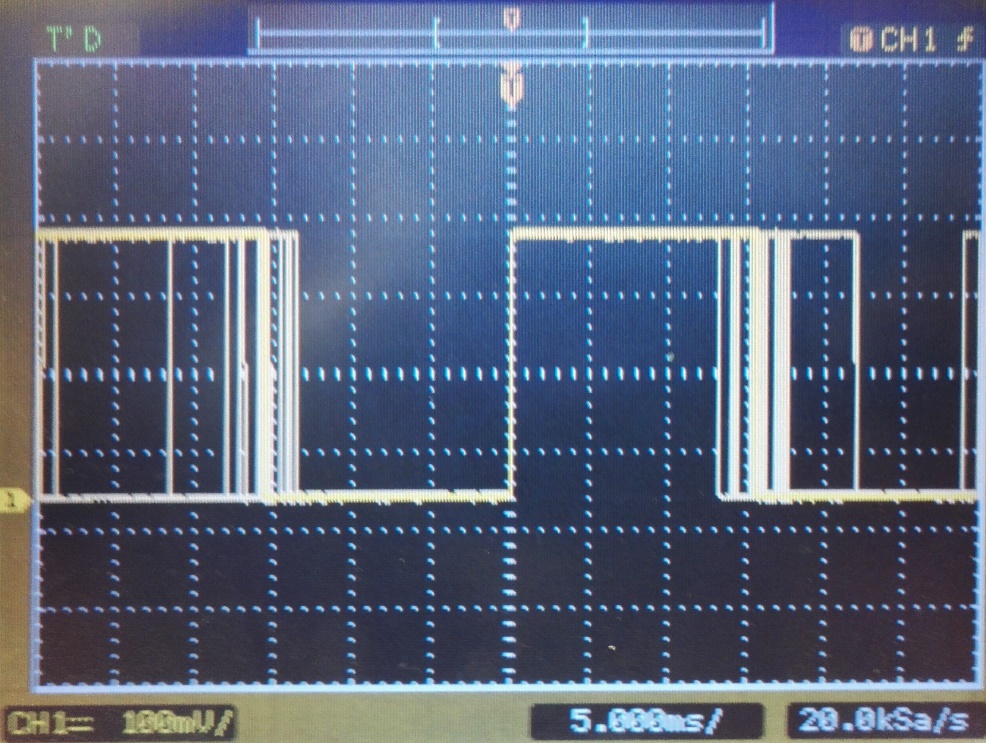
Afin de pouvoir piloter le Vituoso comme il faut, nous avons besoin d’un temps de réponse précis à la ms. Par conséquence, des tests ont été réalisé avec différentes manière.

Les tests sont des clignotement d’une sortie du Raspberry pi qui ont été mesuré à l’oscilloscope avec une rémanence infinie pour observer la gigue.

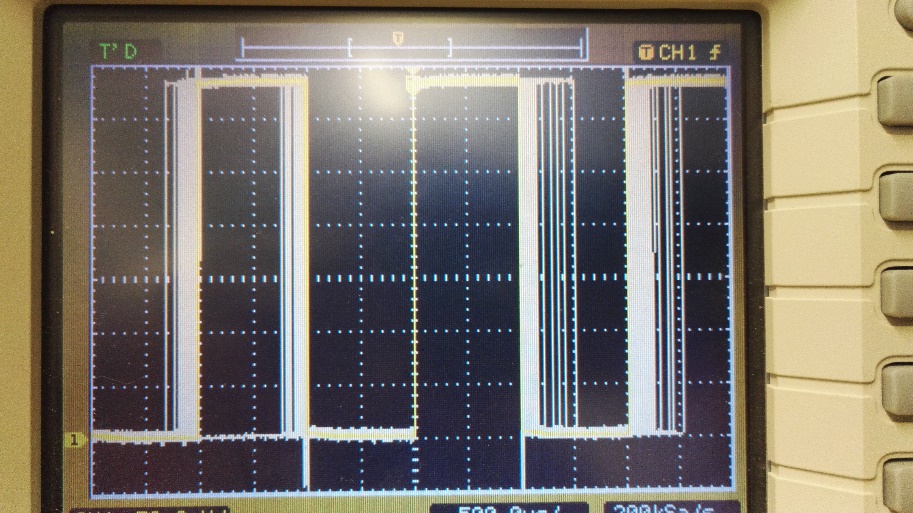
Le premier test est avec une application normal sans thread :



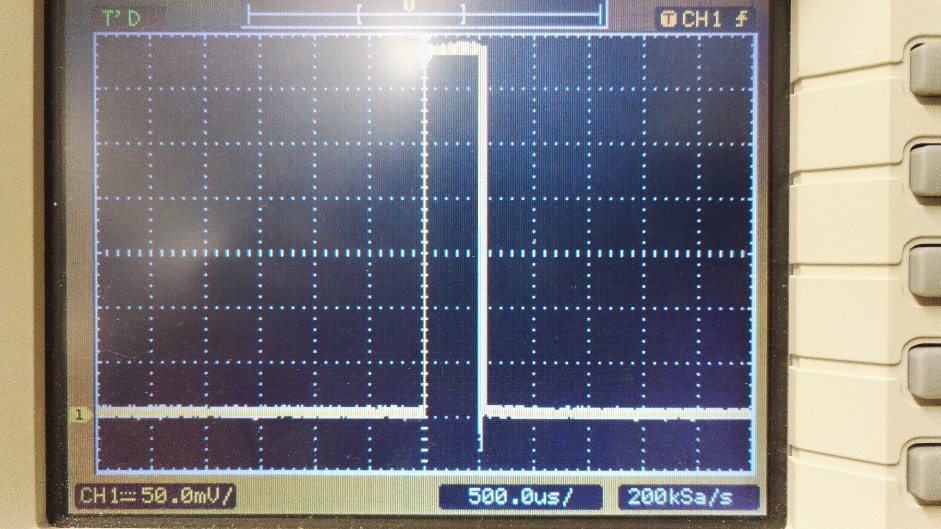
Le deuxième est avec un thread qui tourne avec des attentes passives :



Et le troisième test est avec un thread qui tourne à 100% avec un attente active :



Un autre test a été réalisé avec un application qui tourne en arrière-plan aussi avec un thread qui fonctionne à 100% :



Après ces tests, il a été constaté les faits suivants :

En attente active, Windows IoT permettrait de piloter le Virtuoso avec des temps de l’ordre de la milliseconde avec une gigue de 500 microsecondes. Malheureusement l’attente active n’est l’objectif souhaité car il fait tourner un thread à 100%.

Pour ce qui est de l’attente passive, Windows IoT ne tient pas la route car il offre une précision de l’ordre des 15ms.

### Woopsa & html

#### Qu’est-ce que c’est ?

Woopsa est un protocole de communication pour le web qui est open source. Il fonctionne sur le principe de l’orienté objet ce qui le rend pratique pour des programmes codés en objet.

#### Problème

Woopsa fait partit des exigences pour le développement du Virutoso. Car le Virtuoso doit émettre un réseau wifi et gérer la communication à l’aide de Woopsa. Afin d’utiliser Woopsa, des connaissances en html sont aussi obligatoires.

#### Démarche

Dans le but de découvrir un minimum Woopsa, html et JavaScript, car ils étaient jusqu’à maintenant inconnus, nous avons généré un programme impliquant un serveur Woopsa et implémentant un page web grâce à l’html eau JavaScript.

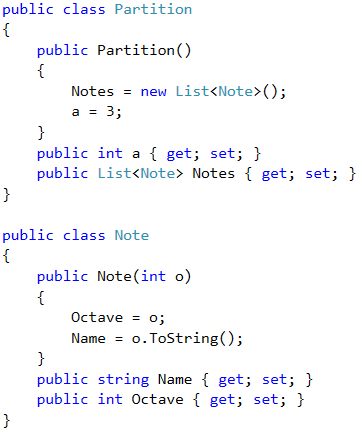
##### Programme de test

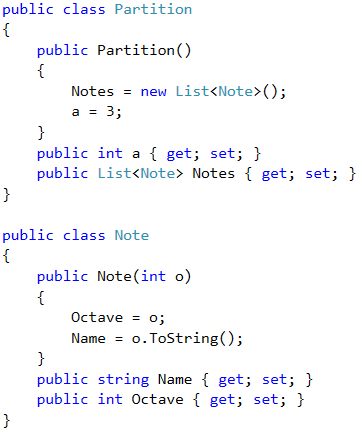
Le programme est simple, il implémente une instance de WoopsaServer qui prend une partition p. Puis il ajoute les routes avec les pages html dans le dossier les contenant.



À l’aide du code ci-dessus, pour accèder aux page html qui sont dans le dossier \Page\, il faut écrire <http://localhost/files/nomDeLaPage.html>

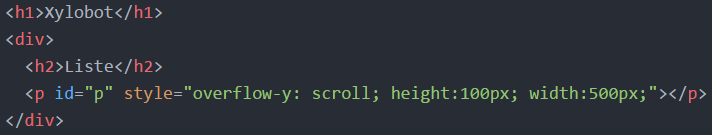
La classe partition est une classe « brouillon » avec une liste de notes.

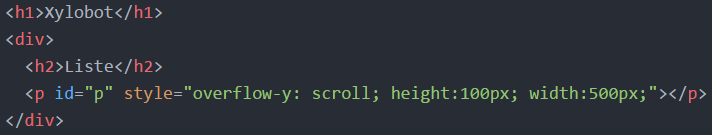




##### Html/JavaScript

L’html est simple, car pour tester l’affichage d’une liste, tout ce qu’il implémente c’est une liste



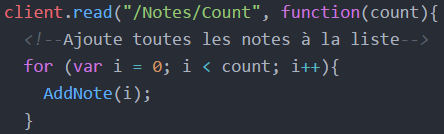
Remarque, il implémente aussi un titre pour la liste avec .

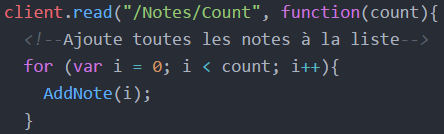
En revanche le JavaScript est un peu plus complexe.

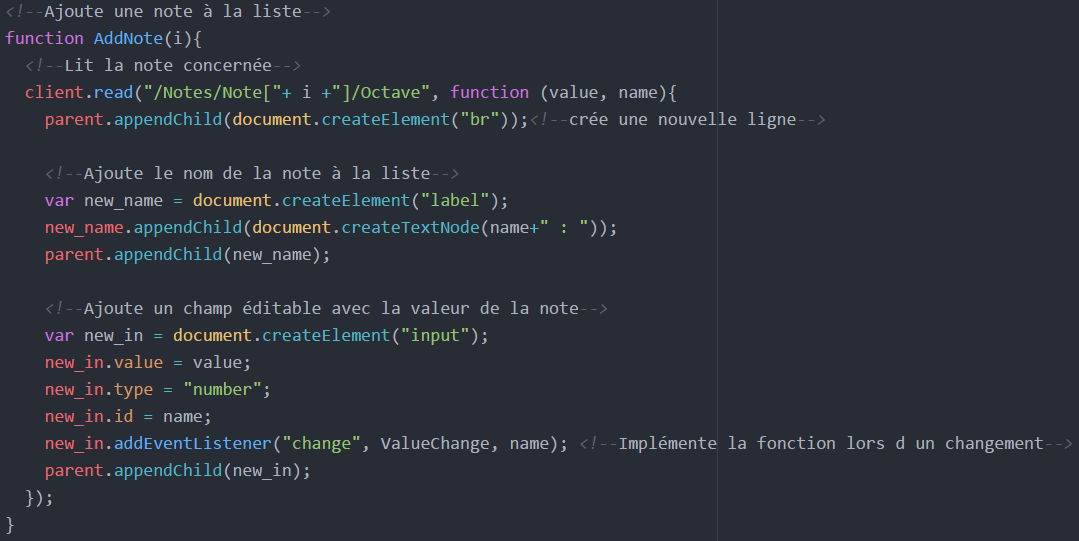
Pour commencer, il crée une instance de la liste avec l’id « p » ainsi qu’une instance d’un client Woopsa.



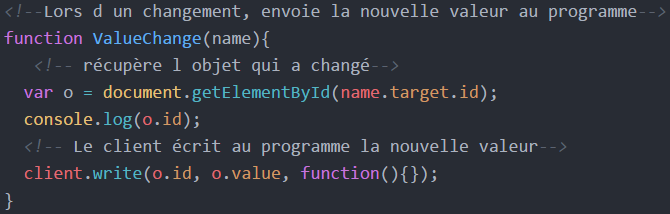
Puis au chargement de la page web, il ajoute toutes les notes existantes dans la liste.



La fonction  crée un label avec le nom de la note ainsi qu’un champ éditable pour la valeur de l’octave. Le nom ainsi que l’octave sont lus avec l’instance client de Woopsa.

Et pour finir, lorsqu’un changement intervient dans le champ éditable, on appelle la fonction « ValueChange » avec le nom de la note impliquée.

Cette fonction récupère l’objet grâce au nom et renvoie au serveur la nouvelle valeur pour l’objet concerné.



### Concept HMI

#### Qu’est-ce que c’est ?

Concept HMI est une librairie pour le C# permettant de faciliter et d’accélérer le codage d’un programme grâce aux fonctionnalités qu’il implémente.

#### Problème

La question s’est posé de savoir s’il serait utile d’utiliser ou non concept pour les applications du Virtuoso et de la conversion des fichiers midi.

#### Démarche

Pour expérimenté concept HMI, un rendez-vous chez Objectis, qui est la boîte développant concept, a été pris. Objectis nous a fourni des tutoriels pour l’utilisation de concept.

Les premiers points du tutoriel ont été réalisé ainsi que ceux ciblant plus les objectifs du Virtuoso. Notamment la partie XML.

À la fin des tests, la décision fut prise d’utiliser concept surtout pour la partie XML gérant la sauvegarde et les chargements automatiquement.

Par contre, pour la suite du projet, il reste à voir comment changer les styles de concept pour correspondre aux attentes de l’application.

## Comparatif des plateformes

### Problème

Après une évaluation des risques, il s’est avéré que l’idée de base qui était d’utiliser Windows IoT ne correspondait pas aux objectifs du Virutoso. Il a donc fallu procéder à un choix d’une nouvelle plateforme.

### Démarche

Afin de résoudre le problème de la plateforme IoT, un comparatif a été accompli. Il compare chaque possibilité de pilotage (ex : Linux avec mono) avec les éléments requis pour le bon déroulement du projet. Et permet de faire un choix rapide et efficace.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Linux (mono)** | **Windows IoT** | **Windows IoT + Arduino** | **NUC + Arduino** |
| **Interface graphique** | - En html | + En C#, langage maîtrisé | + En C#, langage maîtrisé | + En C#, langage maîtrisé |
| **Lecture/écriture des fichiers** | + Fonctionne de base | - Test exemple de msdn ne fonctionne pas | - Test exemple de msdn ne fonctionne pas | + Fonctionne sur Windows |
| **Temps de réponse** | + suffisant pour le Virtuoso | - Pour obtenir un temps de réponse satisfaisant, il faut faire tourner un thread à 100% | + Gérer par l’Arduino, suffisant | + Gérer par l’Arduino, suffisant |
| **I2C** | + Implémenter directement de mono au xylophone | + Implémenter directement du Raspberry au xylophone | - Passe par l’intermédiaire de l’Arduino | - Passe par l’intermédiaire de l’Arduino |
| **Woopsa** | + Aucune modification à reprendre | - Modification à faire | - Modification à faire | + Aucune modification à reprendre |
| **Accès hardware** | - Inconnu | + Déjà tester et fonctionne parfaitement | - Inconnu sur la partie Arduino |  |
| **Debugging** |  |  |  |  |
| **Temps à investir** | - Apprendre à utiliser mono  - Revoir certain point déjà connu sur IoT | - Apporter des modifications à Woopsa | - Apporter des modifications à Woopsa  - Apprendre à utiliser Arduino | - Apprendre à utiliser Arduino  + Interface sur C#, utilisation de concept possible |

Suite au tableau ci-dessus, le choix du NUC de Intel avec un Arduino est le choix optimal. L’autre choix qui pourrait être pris, serait le Raspberry avec mono mais celui-ci n’offre pas la fonctionnalité importante du debugging. Par contre, pour la partie I2C, il n’y pas besoin de traitement particulier pour déléguer la communication avec le xylophone.

Un inconvénient du NUC est le prix qui est facilement le triple des autres.

Avec le NUC, comme il utilise Windows, il sera possible de mettre le programme pour les fichiers midi directement dessus et par conséquent, pouvoir ajouter dans partition sans un ordinateur externe. Tout sera donc regroupé dans une machine (ordinateur).

En plus de cela, il sera possible d’utiliser concept HMI pour la partie Virtuoso.

## Choix du matériel

### Problème

À la fin du comparatif des plateformes, après avoir fait le choix de partir avec un NUC combiné avec un Arduino, il a fallu faire un choix pour le matériel.

### Démarche

Le choix pour l’Arduino a été vite fait, car nous avons repris celui déjà disponible avec le Virtuoso. (Un Arduino était fourni de base avec le Virtuoso pour le travail de bachelor)

Du côté du NUC, après des recherches, le NUC5PGYH fut le meilleur choix pour les raisons suivantes :

* Un disque dur ssd 32Go implémenté de base
* Windows 10 déjà installé
* 2Go ram DDR3 mis de base
* Un prix pas trop élevé (260.-)
* Disponible et en stock sur digitec.ch



## UML

Le diagramme UML décrit ici, est le diagramme obtenu à la fin de la pré-étude. Ce diagramme m’a servi, d’une part, comme base pour le planning du projet, car il décrit dans les grosses lignes qu’est-ce que devras implémenter le projet. Et d’autre part, pour structurer le code des applications dès le début.

L’UML complet contient les UserControls ainsi que toutes les classes « prédites » pour les codes du projet. Il en découle donc plusieurs diagrammes.

### Programme Virtuoso

Pour la partie « Xylophone » : le diagramme HMI donne :



Ces différent UserControls Sont les vues offertes par l’application. Les vues « Home » et « About » n’utilise aucune autre classe.

Par contre, les vues « EditPlaylist », « Supervision » et « Settings » utilisent d’autres classes comme montrer ci-dessous.



La partie « EditPlaylist » utilise la playlist pour accéder à ces propriétés et les modifiées. Pouvant ainsi permettre à l’utilisateur de modifié la playlist du xylophone par le bief du UserControl.

La partie « Supervision » utilise le séquencer offrant à l’utilisateur les fonctions play/pause, stop, partition précédente ou suivante.

Et enfin, la partie « Settings » est là pour les réglages du Xylobot comme par exemple le temps frappe de chaque note. Permettant ainsi l’ajustement de l’intensité.

Après cela, nous avons les classes que composera le programme Virtuoso :



Je ne vais pas décrire toutes les classes, mais en bref nous avons le Xylobot, qui représente le xylophone avec ces 37 notes et actuateurs, qui contient une gestion de la communication avec le XyloCommunication. Le Xylobot offre la possibilité de test toutes les notes pour vérifier qu’il n’y ait rien de cassé.

Ensuite vient la Playlist, qui naturellement contient plusieurs Partitions (partition faîte pour le xylophone) et chaque une de ces partitions ont des Notes.

Et pour finir, le Sequencer, qui gère un peu tout derrière pour la lecture des partitions.

Le catalogue est là pour la gestion de la recherche. Il liste toutes les partitions disponibles.

Il reste le diagramme pour le serveur web :



Avec le Server qui utilisent la Playlist, car les clients se connectant dessus pourront seulement faire des propositions de partitions à jouer et voir la playlist en cours. Il inclut aussi le Sequencer, car il offrira la possibilité de voir l’avancement de la partition courante (comme décrit dans le cahier des charges).

### Programme midi🡺XML

Comme pour la partie Xylophone, nous avons le diagramme avec les UserControls :



La partie principale du programme est l’édition de partitions. Elle permet de modifier une partition midi pour ensuite l’enregistrer et l’envoyer au xylophone à l’aide du FileManagement.

On obtient les deux diagrammes suivant :



Comme le Virtuoso utilise des PartitionXylo, le FileManagement utilise aussi une PartitionXylo, car c’est lui qui les envoie au Virtuoso.

L’édition de partition se fait avec une PartitionMidi obtenue directement depuis un fichier midi.

La différence entre les PartitionXylo et PartitionMidi sont les canaux. Une PartitionMidi a plusieurs canaux représentant plusieurs instruments tandis que la PartitionXylo n’a pas de canal étant donné qu’elle ne compose qu’un instrument.



# Projet

## Protocole de communication

Afin de pouvoir communiquer entre l’Arduino et le NUC, un protocole de communication USB était nécessaire.

### Composition des messages

Message envoyé par le NUC :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte de Start | Numéro de message | Type | Taille des données | Données |
| 1 byte | 1 byte | 1 byte | 2 bytes | X bytes |

Message envoyé par l’Arduino:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Byte de Start | Numéro de message | Type | Taille disponible dans le buffer | Tick courant (Tick pour les notes) |
| 1 byte | 1 byte | 1 byte | 1 byte | 4 bytes |

### Types de messages

|  |  |  |  |
| --- | --- | --- | --- |
| **NUC** | | **Arduino** | |
| Tempo | 0000 00002 | Ok | 0000 00002 |
| Notes | 0000 00012 | Too many data | 0000 00012 |
| Start | 0000 00102 | Erreur Start byte | 0000 00102 |
| Stop | 0000 00112 | Erreur Type | 0000 00112 |
| Pause | 0000 01002 |  |  |

Le NUC envoie un paquet à l’Arduino avec un numéro de message. L’Arduino traite le message et renvoie au NUC un message avec le même numéro et ainsi de suite.

Si le NUC ne reçoit pas le message de réponse après un delay, il renvoie une fois les données et après cela, s’il n’a toujours pas de réponse, il génère une erreur.

Quand l’Arduino reçoit un message qu’il a déjà reçu, il ne le traite pas mais renvoie quand même une réponse. Si la taille des données qu’il reçoit est trop grande, l’Arduino renvoie « Too many data ».

Si le NUC ne reçoit pas de réponse avant un certain temps, il génère une erreur.

La vitesse de transmission des données n’est pas modifiable. Elle est définie dans le code.

## Réglages USB

### Problème

Afin de pouvoir communiquer en USB de l’Arduino au NUC, il faut régler les paramètres de la communication USB des deux côtés.

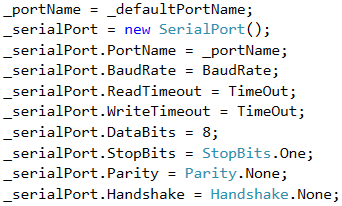
### Démarche

Dans le but de réaliser la communication le plus simplement possible, les paramètres du côté Arduino n’ont pas été changé hormis la vitesse de communication (baud rate).



Avec BAUD\_RATE\_SERIAL qui est une constante valant 115'200.

Pour ce qui est du programme du NUC, les paramètres sont ajustés en correspondance à ceux de l’Arduino.



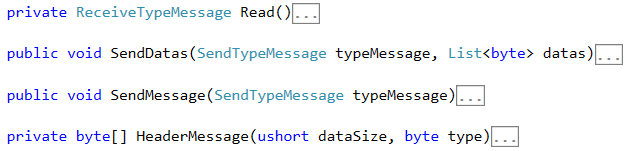
La BaudRate est réglé identiquement à l’Arduino, c’est-à-dire 115'200.

## Communication USB

### Partie NUC

La communication USB utilise le protocole décrit précédemment.

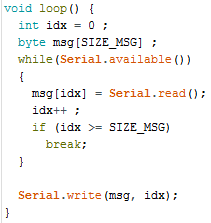
La partie NUC est gérée par la classe XyloCommunication. Cette classe implémente les fonctions suivantes permettant de gérer la communication :



Remarquez que les seules fonctions publiques sont celles pour l’envoi de messages. Ceci est dû au faite que lors de l’envoie d’un message, l’Arduino répond directement. La lecture des données est donc faîte et contrôlé dans les fonctions « Send » à l’aide de la fonction « Read ».

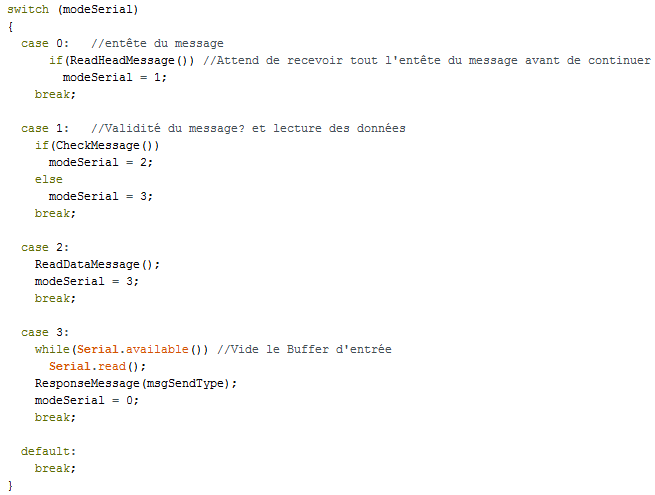
### Partie Arduino

L’implémentation pour l’Arduino a posé quelques problèmes. Car un premier code a été fait, basé sur une faute. Le code envoyait des données à l'Arduino et celui-ci renvoyait les mêmes. Les données envoyées et reçues concordaient bien mais plus tard il a été remarqué que l'Arduino traitait déjà des données sans en avoir reçu la totalité. À cause de cela, en checkant s'il y avait des données dans le buffer du serial (fonction Arduino: Serial.available()) on les détectait mais elles n’étaient pas encore toutes arrivées. Ainsi dans l’exemple suivant, le programme recevait les bytes données et on les renvoyait une par une au lieu de paqué en un bloc d’une taille de SIZE\_MSG :



Il a fallu réadapter le code en prenant compte cet effet.

Ainsi on obtient un code avec la partie de réception qui est fragmenté en parties agissant comme une machine d’état.



Le siwtch-case ci-dessus est dans la fonction de base «  ».

À chaque passage dans loop, le buffer est checké et s’il y a des données qui sont arrivées, on regarde s’il est valide au prochain passage dans loop. Si le message est valide, on lit les données. Finalement, on répond au NUC et on attend le prochain message.

Un autre problème qui survenu et qu’il a fallu débugger est la vitesse de communication. Après quelque recherche rapide sur le site Arduino pour les vitesses disponibles pour l'Arduino méga et une vitesse a été choisie. Or il s'est avéré que cette vitesse de communication n'était enfaite pas adaptée au méga que je possède. En conséquence, quand on exécutait le code de test (envoie des données à l'Arduino et celui-ci renvoie les même) deux bytes en trop apparaissait au début des messages reçus dans l'ordinateur.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| byte 0 | byte 1 | byte 2 | byte 3 | byte 4 |
| 255 | 0 | 2 | 128 | 0 |

Message envoyé :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| byte 0 | byte 1 | byte 2 | byte 3 | byte 4 |
| 171 | 171 | 255 | 0 | 2 |

Message reçu :

### Remarque

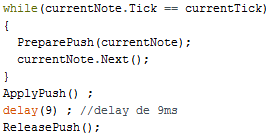
L’Arduino offre une possibilité de débogage grâce au port USB par lequel la compilation du programme est faîte. Or cette fonctionnalité n’a pas pu être utilisé étant donné que le port USB était déjà occupé par la communication NUC – Arduino. Ce point non- négligeable a posé quelque problème notamment pour le débogage de la communication I2C.

## I2C

La partie i2c a donné plus de fil à retordre que prévu. Ceci dû au mauvais choix de partir avec des interruptions. Et il s'est avéré qu'il n’était pas possible de communiquer en i2c à l'intérieur d'une interruption car les fonctions de bases sont bloquantes.

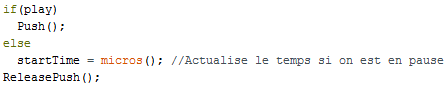
Les interruptions étaient à première vue un bon choix, car pour jouer la musique, il intervient un tempo régulier donnant le rythme.

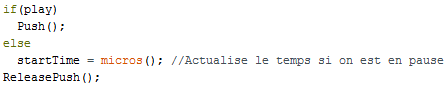
Le programme a donc été réorienté une première fois en utilisant la fonction  faisant une attente passive d’un nombre de millisecondes donné. Ainsi, le programme préparait toutes les notes qui devait être jouée, puis activait les actuateurs, attendait un certain temps (ici 9ms) et pour finir, désactivait les actuateurs.



Le principal problème de cette méthode est qu’il n’est pas possible d’ajuster le temps de frappe de chaque note indépendamment des autres.

Le code a donc été adapté de manière à corriger ce problème. Pour ce faire, la fonction  a été remplacée par la fonction  qui retourne le nombre de microseconde depuis le lancement du programme.



Nous avons donc le code ci-dessus dans la fonction  qui regarde à chaque passage si le programme est en train de jouer ou non. Si le programme est en train de jouer, on appelle la fonction  qui active les actuateurs et met à jours le tick courant (qui correspond à la progression dans la partition).

Même si nous sommes plus en train de jouer, le programme désactive au moment voulu les notes qui ont été poussée. Le temps de frappe des notes est géré avec les fonctions Push/ReleasePush.

## Programme de conversion

### Problème

Dans le but de pouvoir exploiter les fichier midi, il est nécessaire d’avoir un outil permettant de les convertir sous un format exploitable pour le xylophone. Afin de pouvoir adapter les partitions pour qu’elles puissent être jouée par le xylophone, il faut aussi pouvoir éditer le fichier midi ou le fichier convertit.

### Démarche

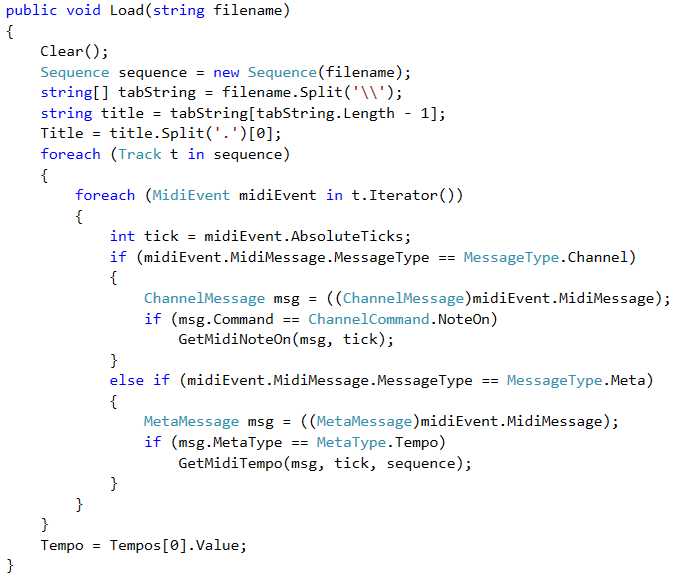
Dans un premier temps, seule la partie de conversion midi à XML fut réalisée. L’édition des fichiers est faîte en midi grâce au logiciel « MidiEditor ». Avant la partie conversion, une partie obligatoire a été réalisé, c’est la partie de chargement.

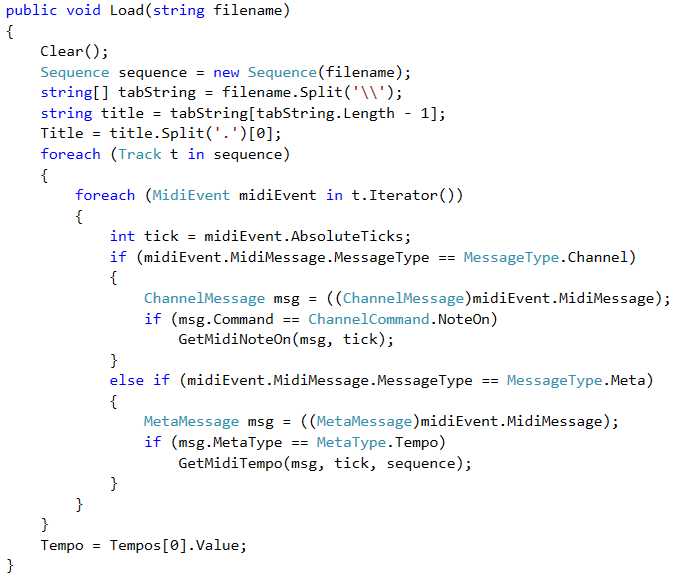
#### Chargement des fichiers midi

Pour faciliter le chargement des fichiers, une librairie open source a été installer. Celle-ci se nomme miditoolkit.

##### Chargement

Pour charger les partition midi, la classe PartitionMidi implémente la fonction Load qui prend en paramètre un nom de fichier.

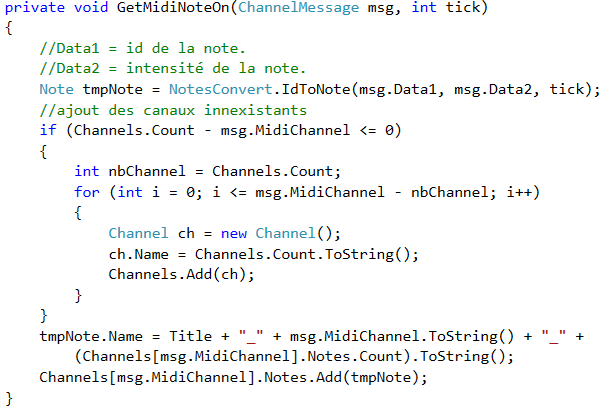


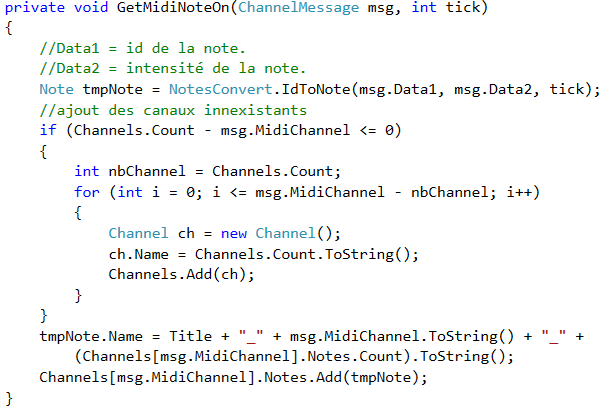
Cette fonction efface l’ancienne partition chargée s’il y en avait une. Puis charge le fichier midi dans une instance de Sequence : 

En parcourant cette séquence, on prend tous les messages « NoteOn » et « Tempo » qui sont les seuls messages qui nous intéresse pour le moment.

##### Notes du midi

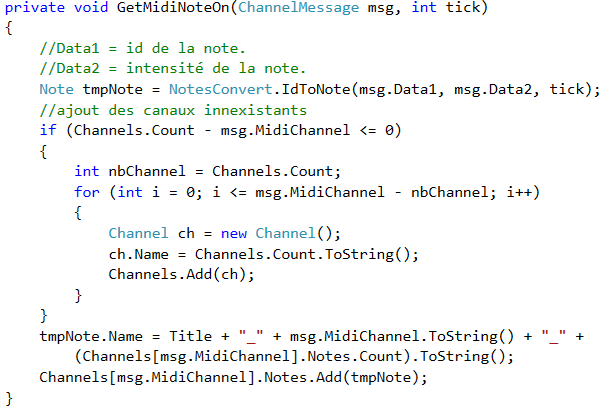
Lors du chargement du fichier midi, on a vu qu’on récupérait les messages « NoteOn ». Lorsqu’un de ces messages apparaît, on enregistre la note dans notre PartitionMidi avec la fonction « GetMidiNoteOn ».



La première étape est la conversion de la note : 

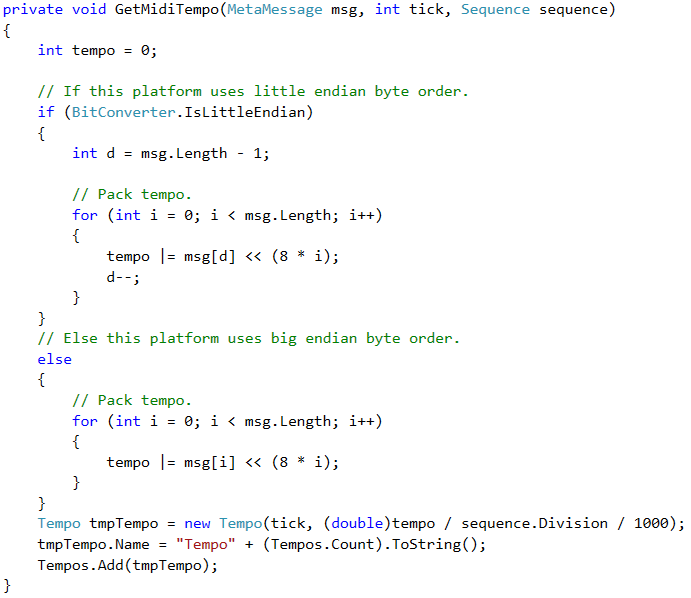
Ceci nous retourne une note avec l’ID qui est « msg.Data1 », l’intensité qui est « msg.Data2 » et le tick.

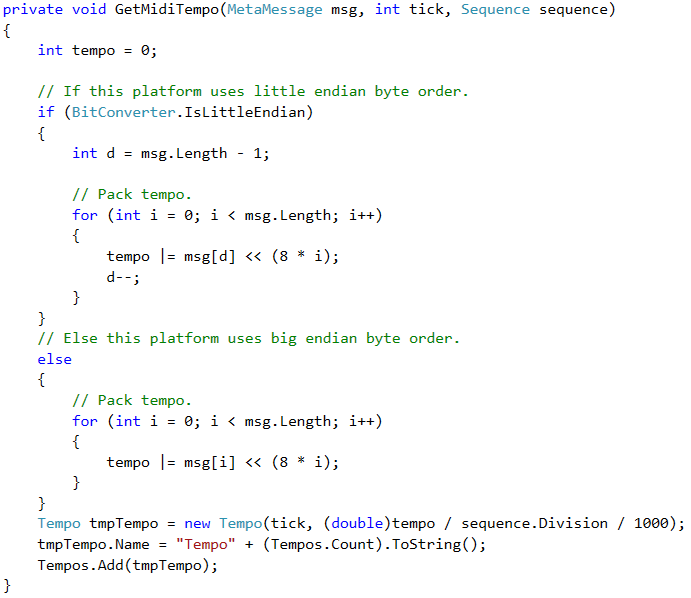
De cette note, si elle fait partie d’un canal qui n’existe pas encore (numéro de canal inexistant), on ajoute des canaux jusqu’au canal de la note en question.

Puis on enregistre la note dans le canal lui correspondant :

##### Tempos du midi

Comme il est possible que le tempo d’une musique change au cours du temps, on récupère tous les tempos ainsi que leurs ticks.



Comme les tempos sont en plusieurs bytes, on concatène tous ces bytes dans un entier : 

Puis on enregistre le tempo avec son tick et sa valeur convertie en milliseconde.

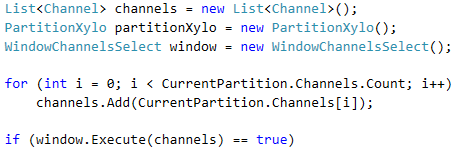
#### Conversion midi 🡺 XML

##### Choix des canaux à convertir

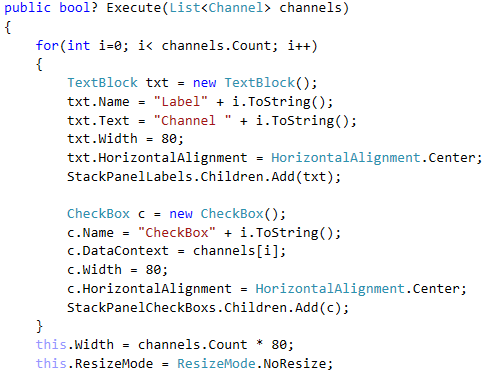
Comme les fichiers midi peuvent contenir plusieurs canaux représentant différent instrument, pour la conversion, il faut que l’utilisateur choisissent les canaux qu’il veut garder. Car jouer un orchestre sur un xylophone ne serait pas très musical.

Il y a donc une fenêtre qui s’ouvre lors de la conversion, demandant à l’utilisateur de sélectionner les canaux désirés.

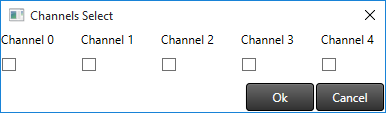
A l’appel de la fenêtre, on envoie une liste de tous les canaux disponibles de la partition midi.



Puis la fenêtre génère dynamiquement des check box pour la sélection des canaux. Comme l’affectation d’un content au check box ne s’affiche pas, la fenêtre crée aussi un label pour chaque canal.

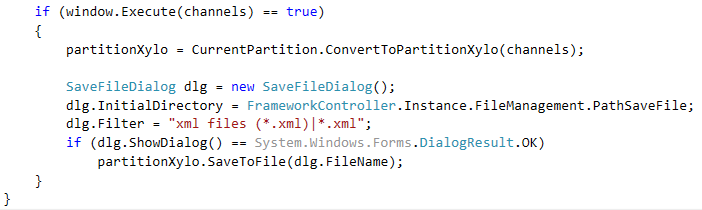


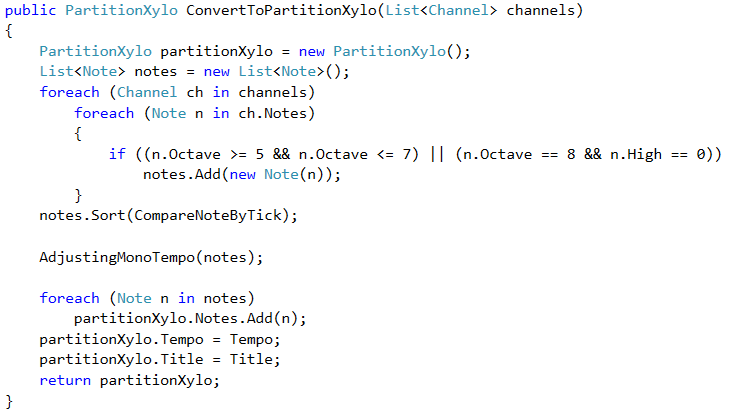
On obtient alors ceci :

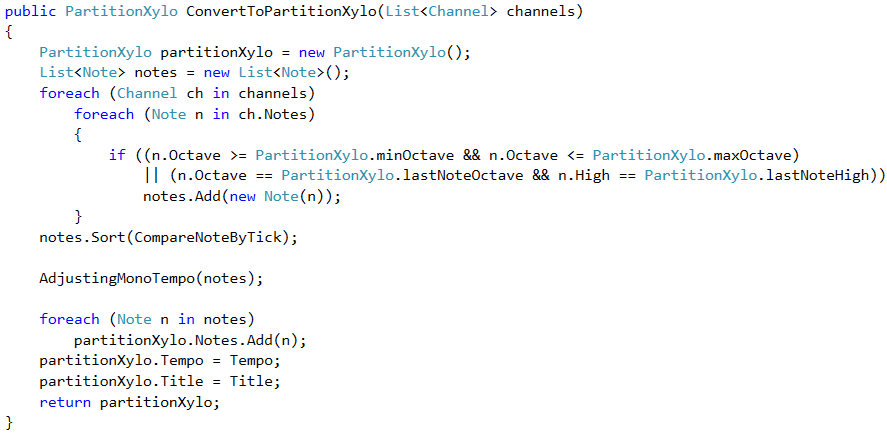


##### Conversion et sauvegarde

Une fois les canaux validés, si l’utilisateur a confirmé son choix, on convertit la partition midi en partition xylo et on demande le nom pour la sauvegarde du fichier.



Pour la partie de conversion, la fonction demande les canaux à convertir puis pour chaque canal, elle ajoute les notes comprises dans les octaves du xylophone dans une seule et même liste : .



Comme les notes proviennent de différents canaux et qu’elles sont triées par canal, il faut les retrier par « tick ». Pour cela, la classe « list » offre la fonction « Sort » qui prend en paramètre une fonction de tri.

Enfin, avant de ranger les notes dans la « PartitionXylo », il faut ajuster le tempo pour en obtenir un seul à l’aide de « AdjustingMonoTempo ». Cette fonction ajuste le tick des notes pour obtenir un seul et même tempo qui est le premier.

## Remote debugging

### Problème

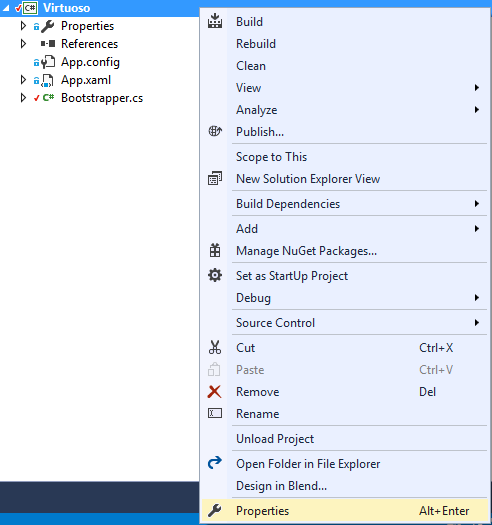
Pour éviter de devoir installer visual studio sur le NUC, nous avons choisi de partir sur le remote debugging. Et comme nous pouvions aussi développer directement sur un pc lambda avant de recevoir le NUC, cela simplifiait le développement.

### Démarche

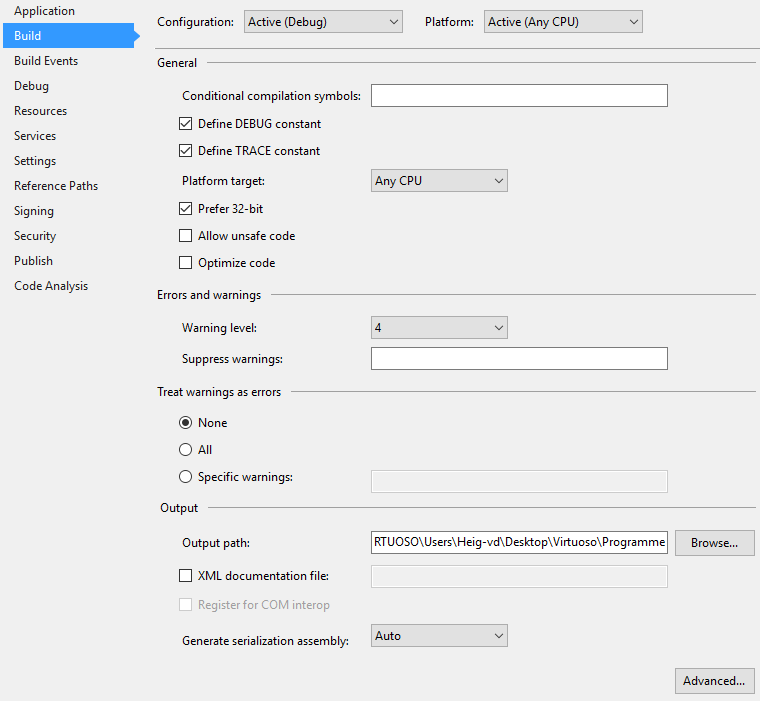
Après plusieurs recherches sur internet, nous avons installé l’outils « Visual Studio 2015 Remote Debugger » disponible sur le site microsoft. Après plusieurs essais, nous obtenions aucuns résultats. Nous n’arrivions pas à établir la connexion entre le pc et le NUC. Nous avons essayé le remote debugging sans authentification, à ce moment-là la connexion se faisait mais ensuite il était impossible de lancer l’application.

Pour finir, nous avons trouvé le problème. Celui-ci était dû au faite que nous prenions un utilisateur sans mot de passe. Après l’application d’un mot de passe, un autre problème bloquait le déploiement de l’application. Nous n’utilisions pas de dossier partagé dans le NUC et par conséquence, le projet ne pouvait pas s’installer dans un dossier dans le NUC.

Finalement après ces deux problèmes qui ont pris un peu de temps à corriger, le remote debugging fonctionne. Et pour le paramétrer, il faut aller dans les propriétés du projet :

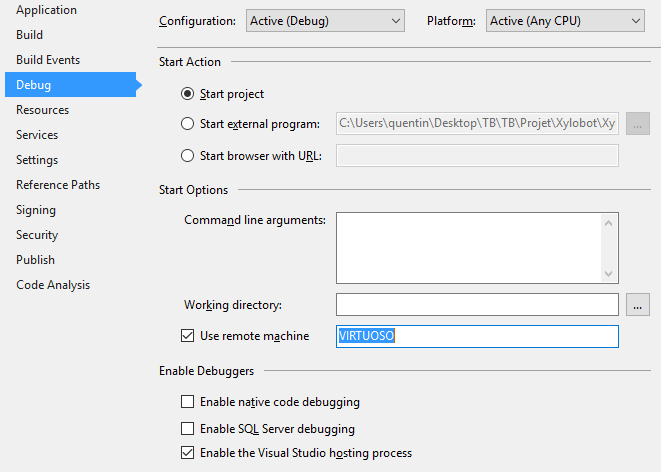


Et modifier l’ « Output path » dans build :



Le chemin doit être celui du dossier partagé.

Il faut cocher la case « Use remote machine » et préciser le nom du NUC :



Et finalement, il ne faut pas oublier de lancer « Visual Studio 2015 Remote Debugger » sur le NUC avant de starter le projet.

### Solution

Finalement le remote debugging fonctionne très bien, malgré les petits soucis rencontrés. Il aura quand même pris un peu plus de temps que prévu.

## Programme Virtuoso

## Visuel des interfaces

### Problème

Comme la principale fonction du Virtuoso sera son exposition aux portes ouvertes de la heig-vd, il faut que l’interface graphique soit belle et attractive. Elle doit aussi être simple pour les utilisateurs afin qu’il ne passe pas 30 minutes à essayer de jouer une musique.

### Démarche

En premier lieu, des prototypes ont été fait pour avoir une idée de la direction dans la quelle partir. Puis on est venu se calqué sur ces prototypes pour créer l’interface en amenant quelles que modifications.

### Prototypes

Le fait de voir les prototypes permet aussi de faire de l’ordre dans les idées d’implémentations et de voir si l’interface sera « UserFriendly » ou non.

L’idée de l’interface est de partir sur un design plutôt flat dans le style Windows 8. (13.1 Prototypes Visuel)

### Possibilités offertes par concept

Après un rendez-vous chez Objectis pour voir ce qui était possible de faire, nous avons vu que nos prototypes était réalisable. Malgré cela, il n’a pas été aisé de faire et d’appliquer des styles.

Pour commencer, le composant principal de concept est le NavigationControl. C’est le seul composant wpf de concept qui a été utilisé. Le fichier de style nous a été fourni pas concept et modifié par nos soins pour correspondre à nos attentes. La seule partie du fichier qui a été modifié est le « Datatemplate ».



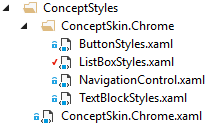
Les couleurs ont été ajustée à l’aide des propriétés « Background » et le « TextBlock » permettant d’afficher le nom sous les images ont été retirés. Ce qui nous donne :



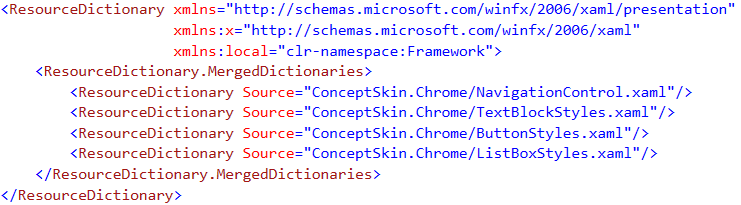
Le résultat est quasiment identique au prototype.

### Spécificité pour appliquer les styles

Avec concept, il a fallu faire quelque modification pour pouvoir appliquer les styles. Notamment mettre tous les styles créés dans un dossier nommé ConceptStyles/ConceptSkin.Chrome



Et les appeler dans le fichiers ConceptSkin.Chrome.xaml comme décrit ici :

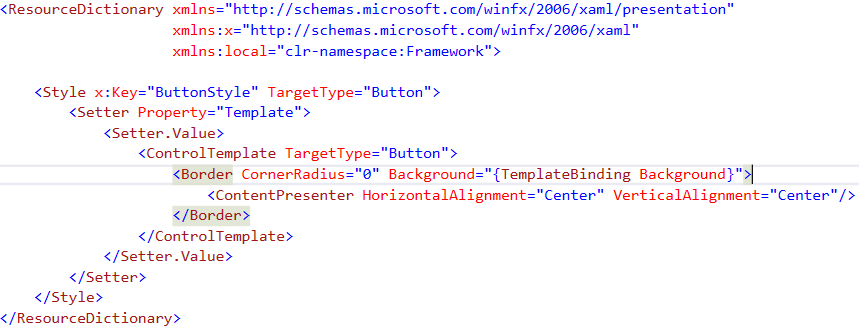


Ceci a été fait pour que concept prenne en compte les styles créés sans devoir tout remodifier.

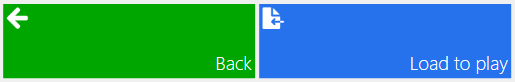
### Styles créés pour le Virtuoso

Pour « contrer » les styles de concept, nous avons créé plusieurs styles pour les composant voulu.

#### Style des « Button »



Le style est simple, il permet juste de mettre un « Border » dans les boutons. Permettant ainsi d’avoir des boutons sans angle et avec un style « flat ».



#### Style des « TextBlocks »



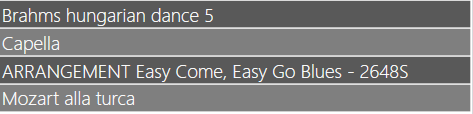
Comme pour chaque application, il faut plusieurs styles de texte différents, c’est pourquoi nous en retrouvons deux ici. Le premier est pour les titre et le deuxième pour le texte « normal » (autrement le reste des texts). Les deux implémentent la même police et la même couleur pour suivre le style des prototypes.

#### Style des « ListBox »

##### Alternation des couleurs des lignes



Ce style donne aux ListBoxs un style moins macabre en alternant la couleur des lignes. Ici les lignes sont grise-foncée et grise-claire.



Remarque : Comme le style est défini pour une cible (  ), lors de la création d’un composant ListBox, il s’applique tout seul. Nous avons fait ce choix car toutes les ListBoxs de l’application ont cette alternation de couleurs.

##### Numéros de ligne et images

Il a fallu ensuite ajouter les images et les numéros de lignes. Ces modifications ont été implémentées directement dans le wpf ou se situe les ListBoxs, car il fallait un « converter » pour le numéro de la ligne et que l’image devait pouvoir se changer pour la ListBoxs des Playlists (ces deux points se font en C# et les fichiers XAML pour les styles n’ont pas de C#).

Nous avons donc :

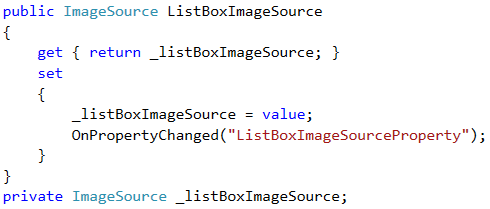


Ainsi, en appliquant DataTemplateListBox à la propriété ItemTemplate des ListBoxs :

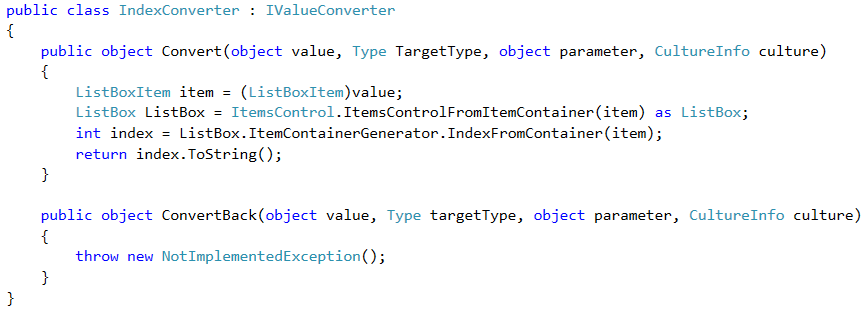


Nous avons des lignes composées d’une image (qui est la même pour toutes les lignes), un numéro de ligne entouré d’un cercle et le titre des items.

L’image est binder sur une propriété :

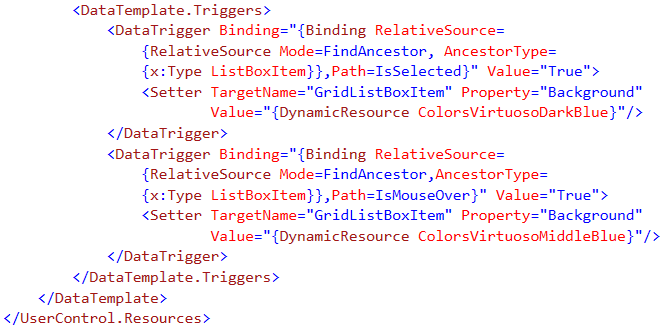


Permettant pour la partie de gestion des playlists de modifié l’image des lignes.

Nous avons ensuite le numéro de ligne qui est une conversion de l’index de la ligne en string :

##### Selection et « Mouse Over »

Afin de changer le style de l’item selectionner et la couleur de l’objet sur lequel on passe avec la souris, dans le DataTemplateListBox, nous avons ajouté des triggers.



Le premier DataTrigger change la couleur de l’item sélectionné en bleu foncé. Et le deuxième change la couleur de l’item sur lequel la souris passe.

Remarque : comme l’application final est sur un écran tactile, le trigger pour le passage de la souris n’a pas vraiment lieu d’être mis à part dans le cas où l’on glisserait le doigt sur l’écran.

### Visuel final

Au final, le visuel de l’application correspond bien au prototype hors mis quelques modifications apportées pour le principe d’utilisation.

Par conséquent, nous sommes satisfaits du résultat obtenu.

## Serveur WEB

### ConceptWeb

#### Qu’est-ce que c’est ?

ConceptWeb est le travail de bachelor d’un collègue de la heig-vd. Il a pour but la simplification du développement de site web. Il est développé grâce à Woopsa et a pour but d’être open source.

#### Utilité

Pour la partie web du Vituoso, il a une grande utilité car il nous évite pas mal d’heures d’apprentissage pour le développement web.

Il a aussi l’avantage d’avoir un style plus orienté pour les smartphones, ce qui nous est bien pratique car le serveur web du Virtuoso cible les smartphones.

### Problème

Dans le cahier des charges, il a été défini la création d’un serveur web. Permettant à une tierce personne d’interagir avec le Virtuoso en voyant la partition en cours de lecture ainsi que sa progression et de faire une demande pour l’ajout d’une musique.

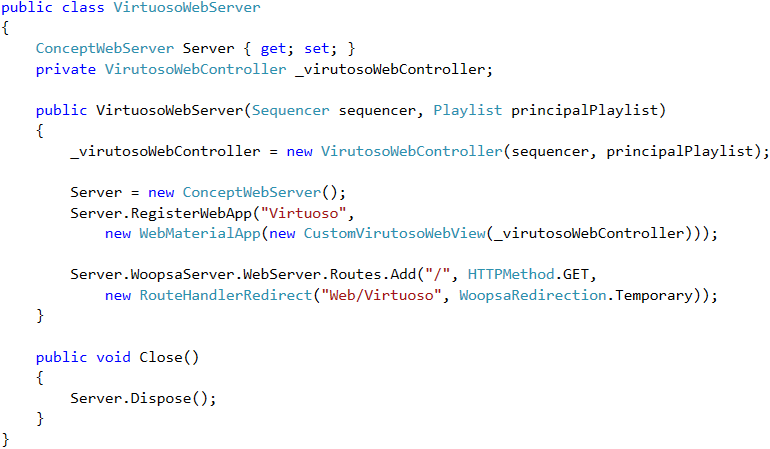
### Démarche

Pour la création du serveur web, nous avons décidé d’utiliser ConceptWeb afin de gagner du temps. Comme il n’implémentait pas encore la gestion pour les listes, nous avons commencé par l’affichage de la partition courante et sa progression. Puis nous nous sommes attaqué à la partie des listes en affichant d’abord la liste de lecture du Virtuoso puis en affichant une liste pour la requête d’ajout.

#### Implémentation du serveur en C#

Grâce à ConceptWeb, l’implémentation au niveau taille du code est assez simple.

Nous avons, tout d’abord, une classe VirtuosoWebServer :



Cette classe prend en paramètres, pour son constructeur, les objets utiles au serveur. Dans notre cas la playlist principale permettant de savoir la liste des partitions qui vont être jouées ainsi que le séquenceur qui est là pour connaître la partition courante avec sa progression.

Ces deux instances sont passée au contrôleur (VirutosoWebController) qui sert à gérer les propriétés qui seront affichées ainsi que la communication avec le serveur.

Le constructeur crée ensuite le ConceptWebServer et lui enregistre une vue à l’aide du contrôleur.

Et pour finir, on crée une redirection depuis le chemin de base du server vers la page avec notre contrôleur. Exemple en local : <http://localhost/> 🡺 <http://localhost/Web/Virtuoso>.

#### Fonctionnement du contrôleur

Lors de la création d’un VirtuosoWebServer, nous avons instancié une vue avec notre contrôleur. Par conséquent, grâce au principe de ConceptWeb, toutes les propriétés et fonctions public de notre contrôleur seront affichés sur la vue.

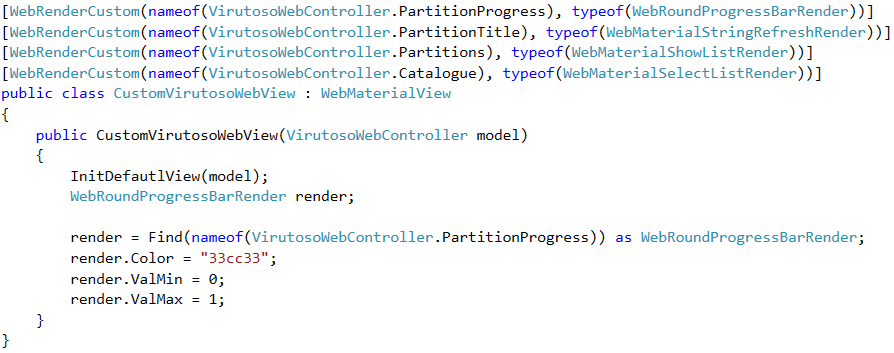
C’est aussi l’une des raisons pour laquelle le contrôleur existe, permettant ainsi de faire le tri des propriétés et fonctions de objets qui lui sont donnés.

Les propriétés du contrôleur seront définies plus tard, mise à part celle de la progression qui a une petite particularité.

#### Fonctionnement de la vue

Comme la vue par défaut de ConeptWeb utilise ses rendus et que nous voulons les modifier par d’autres rendus, il faut crée une « custom » vue. Cette vue permet par exemple de voir la progression qui est un double sous la forme d’une ProgressBar.

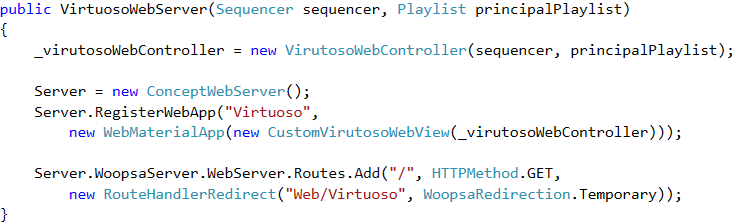
Pour faire cela, c’est très simple, nous avons créé une classe qui hérite de WebMaterialView et qui redéfini les rendus utilisés.



Les rendus pour nos propriétés sont modifiés grâce au balises WebRenderCustom.

Comme certain rendu sont paramétrable, la vue permet aussi de les paramétrer. Comme par exemple, dans le code ci-dessus, la couleur et les bornes de la ProgressBar sont redéfinies.

Pour finir, notre vue customisée est appliquée dans le constructeur de notre VirtuosoWebServer.

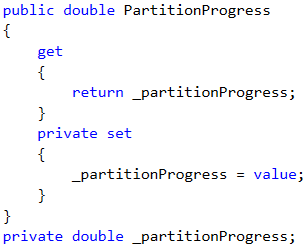


#### Partition courante et progression

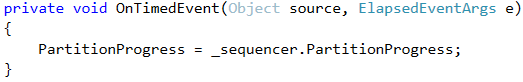
La partition courante avec sa progression a été le point les plus facile à implémenter car les rendus avaient déjà été créé. Une seule subtilité pour le rafraichissement de la progression.

Comme la progression de la partition courante est mise à jour trop souvent dans le séquenceur et qu’en conséquence la progress bar n’avançait pas assez vite, il a fallu adapté le temps de rafraichissement de la propriété dans le contrôleur.

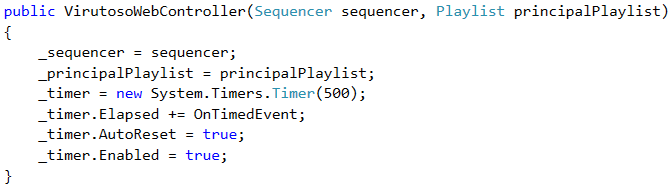
Nous avons donc en premier lieu une propriété standard :



Mais qui est mise à jour par l’intermédiaire d’un de la fonction Elapsed d’un timer :



Ce timer est une variable ce qui permet de ne pas la voir dans la vue et qui est initialisé dans le constructeur.

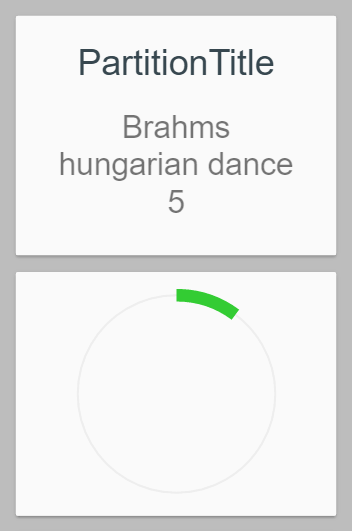


Comme il n’est pas nécessaire d’avoir un rafraîchissement très rapide pour la progression d’une partition qui habituellement de l’ordre de quelques minutes, nous avons choisi d’actualisé la propriété toutes les 500 millisecondes.

Comme les rendus étaient déjà implémenté dans les dll, pour afficher comme les faut des deux propriétés sur la vue, il suffit de mettre la commande montrée dans le « Fonctionnement de la vue » (Chapitre précèdent) :



Nous obtenons ainsi (sur la page web) :



#### Développement des listes Web

#### Développement de la demande d’ajout de musique

### Rendu final

# Conclusion

# Liste des références

# Dictionnaire de symboles et abréviations

(Facultatif)

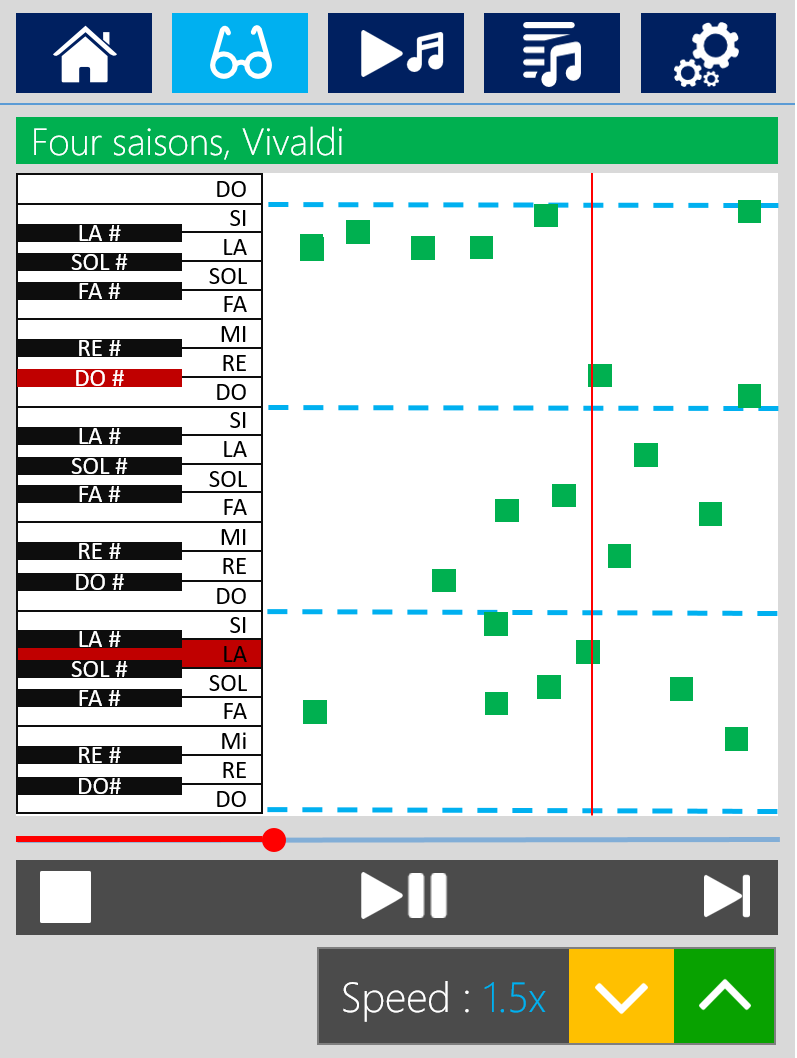
# Figures

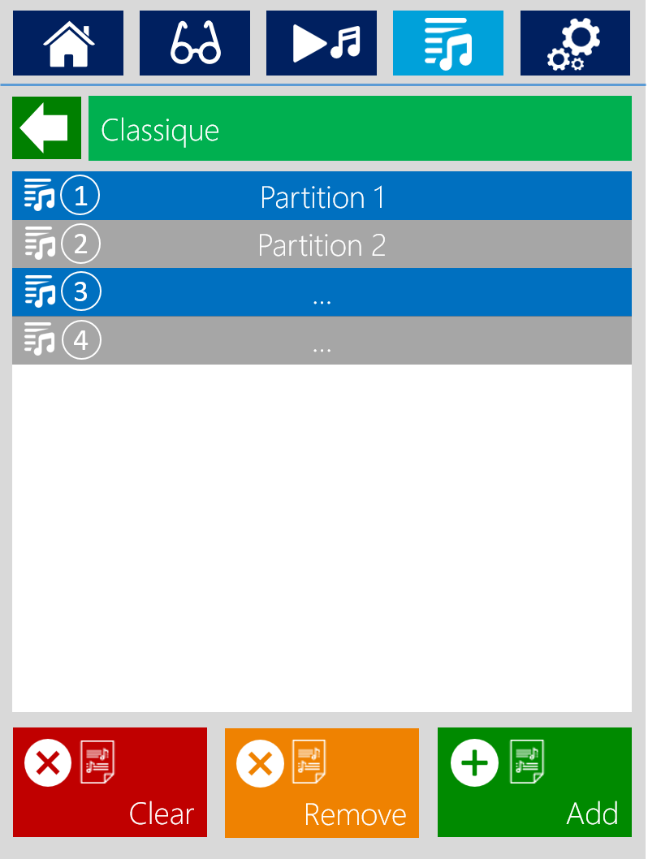
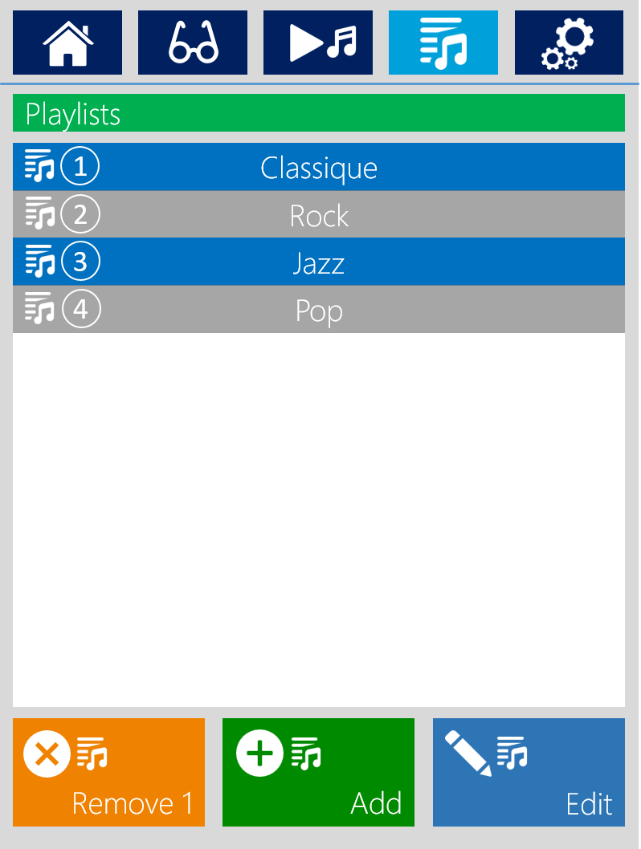
(Facultatif)

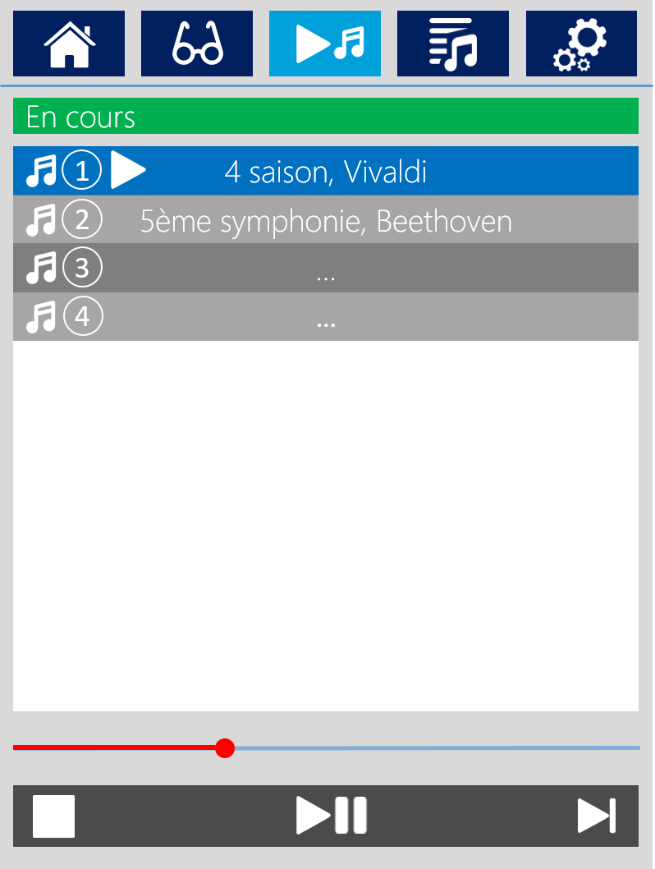
# Journal de travail

# Annexes

## Prototypes Visuel





## Visuels de l’application