Guide du débutant pour C#

et

.NET Micro Framework

19 Juin 2010

Rev 1.04

www.GHIElectronics.com

www.TinyCLR.com

Par : Gus Issa

Version française : Christophe Gerbier

Table des matières

1. Changements

Version 1.01

- Ajout d'une section pour la traduction du livre
- Correction d'une petite erreur dans l'exemple de la matrice de boutons

Version 1.00

- Première version officielle.

2. A propos du livre

2.1. Public visé

Ce livre est destiné aux débutants désirant apprendre .NET Micro Framework. Aucune connaissance préalable n'est nécessaire. Ce livre couvre .NET Micro Framework,Visual C# et même C#!

Si vous êtes un programmer, un bricoleur ou un ingénieur, vous trouverez une mine d'information dans ce livre. Il ne présume rien quant à ce que vous, le lecteur, connaissez déjà et dp,c tout est expliqué en détail.

J'ai passé de mon temps libre personnel (si tant est que j'en ai !) pour faire ce livret. Vous allez certainement trouver des fautes de frappe ou de grammaire, alors je vous demande de me les rapporter sur le forum pour que je puisse améliorer ce livre.

2.2. Traduction du livre

Ce livre est mis à disposition de la communauté dans le but de rendre NETMF plus facile pour tous les utilisateurs. Si vous pensez pouvoir traduire ce livre dans une autre langue, alors nous aimerions avoir votre contribution. Les règles et le document original sont sur cette page :

http://www.microframeworkprojects.com/index.php?title=Free_eBook

3. Introduction

N'avez-vous jamais eu une idée géniale pour un produit mais que vous ne pouviez pas réaliser parce que vous n'aviez pas la technologie pour le faire ? Ou peut-être pensé « Il doit y avoir un moyen plus facile de le faire ! » Peut-être êtes-vous un programmeur qui voulait faire un système de sécurité mais qui du coup pensait qu'utiliser un PC était trop cher pour faire fonctionner un système simple ? La réponse est : Microsoft .NET Micro Framework !

Voici un scénario : vous voulez faire un petit appareil qui enregistre des données GPS sur une carte mémoire (position, accélération et température) et qui les affiche sur un petit écran. Les appareils GPS peuvent envoyer leur position via un port série donc vous pouvez facilement écrire un code sur un PC qui lit ces données et les sauvegarde dans un fichier. Sauf qu'un PC ne tient pas dans une poche ! Un autre problème concerne la mesure de la température et l'accélération sur PC qui ne bouge pas ! Si vous réalisez ce projet en utilisant des micro-contrôleurs classique, comme les AVR ou PIC, alors vous aurez besoin d'un compilateur pour le micro que vous aurez choisi (probablement payant), une semaine pour comprendre le processeur, une semaine pour écrire le driver série, un mois ou plus pour décrypter le système FAT, encore plus de temps pour les cartes mémoires, etc... En fait, ça peut être fait, mais en plusieurs semaines de travail.

Une autre option consiste à utiliser des méthodes plus simples : BASIC STAMP, PICAXE, Arduino, etc. Elles permettent de simplifier le design mais chacune a ses limitations. Seule une poignée d'entre elles ont des possibilités de débogage. Enfin, ces produits ne sont généralement pas adaptés pour une production de masse.

3.1. Avantages

Si vous utilisez le .NET Micro Framework, alors il y a plusieurs avantages :
1. Ca fonctionne avec Visual C# express, qui est gratuit en plus d'être une IDE performante
2. .NET Micro Framework est gratuit et open-source
3. Votre code fonctionnera sur tous ces appareils avec quasiment peu de modifications
4. Débogage complet (Points d'arrêt, pas-à-pas dans le code, variables, etc.)
5. A déjà été testé dans de nombreux produits commerciaux, donc la qualité est assurée
6. Inclus de nombreux pilotes pour divers BUS (SPI, UART , I2C...etc.)
7. Pas besoin de se référer à au datasheet du processeur grâce au framework standard
8. Si vous êtes déjà un programmeur C# sur PC, alors vous êtes déjà un programmeur sur systèmes embarqués avec .NetMF !

Dans ce document, j'utiliserai NetMF pour faire référence au .NET Micro Framework.

4. Portage

Il y a 2 façons de travailler avec NetMF : le porter ou l'utiliser. Par exemple, écrire un jeu en Java sur un téléphone portable est plus facile que d'écrire la machine virtuelle Java (JVM) sur un téléphone. Le fabricant a déjà fait tout le travail de portage de Java sur leur téléphone, ainsi le programmeur de jeu peut l'utiliser avec moins d'efforts. NetMF fonctionne de la même manière : le porter n'est pas facile alors que l'utiliser est très facile.

NETMF peut être séparé en 2 composants principaux : le noyau et le HAL (couche d'abstraction matérielle). Les librairies du noyau sont faites de telle façon qu'elles sont indépendantes du matériel utilisé. En général, aucune modification n'est nécessaire sur les librairies du noyau. Les développeurs d'un matériel doivent par contre concevoir eux-même leurs couches d'abstraction pour pouvoir utiliser NetMF.

4.1. Offres standard GHI

Nous travaillons au plus près de nos clients pour nous assurer que tout fonctionne au mieux. GHI offre de nombreuses fonctionnalités exclusives qui sont installées en standard et gratuitement sur nos produits (Host USB, BDD, PPP, RLP, Wi-Fi, one wire, etc...)

Un support dédié et gratuit est disponible par mail, téléphone et sur un forum.

5. Choisir un produit

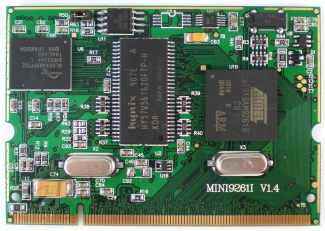
GHI Electronics propose une grande variété de produits allant du plus simple au plus avancé.

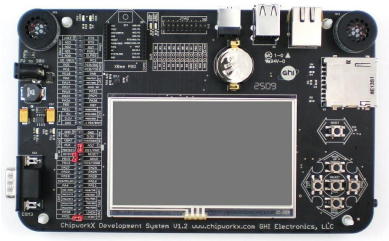
- ChipworkX
- EMX
- USBizi
- FEZ Family
  - FEZ Domino and FEZ Mini
  - FEZ Cobra

5.1. ChipworkX

Si la puissance de traitement et la personnalisation sont des éléments décisifs, alors c'est le bon choix.

ChipworkX utilise un processeur ARM à 200Mhz ARM avec 64MB de SDRam 32-bit et 8MB pour les applications utilisateurs. Il contient également 256MB de mémoire flash interne pour stocker le système de fichiers. Il incluse également les fonctionnalités avancées de NetMF et y ajoute des exclusivités GHI comme le WiFi et le support Hôte USB.

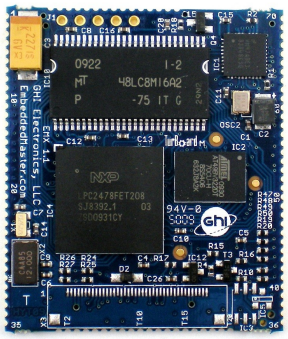


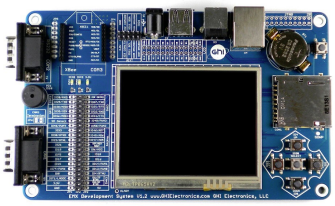


ChipworkX propose égelement la base de données SQLite et autorise les utilisateurs à charger leur propre code natif (C ou Assembly) dans l'appareil en utilisant le RLP (Runtime Loadable Procedures).
RLP permet la programmation d'applications intensives et en temps-réel.

5.2. EMX

Ce petit module inclue toutes les fonctionnalités majeurs de NETMF et y ajoute des exclusivités GHI. Côté logiciel : système de fichiers, TCP/IP, SSL, Graphiques, débogage et bien d'autres fonctionnalités NETMF sont incluses. GHI ajoute également : WiFi, PPP, hôte USB, constructeur de périphérique USB, CAN, E/S Analogiques, PWM et plus. Concernant le matériel : c'est un processeur ARM à 72Mhz avec 8MB SDRAM et 4.5MB FLASH.

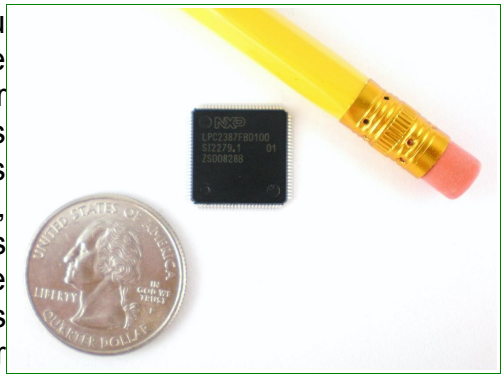




Le processeur sur l'EMX contient nativement la couche Ethernet avec les transferts DMA, ce qui accélère très nettement en comparaison du classique chipset SPI utilisé par les autres.

5.3. Chipset USBizi

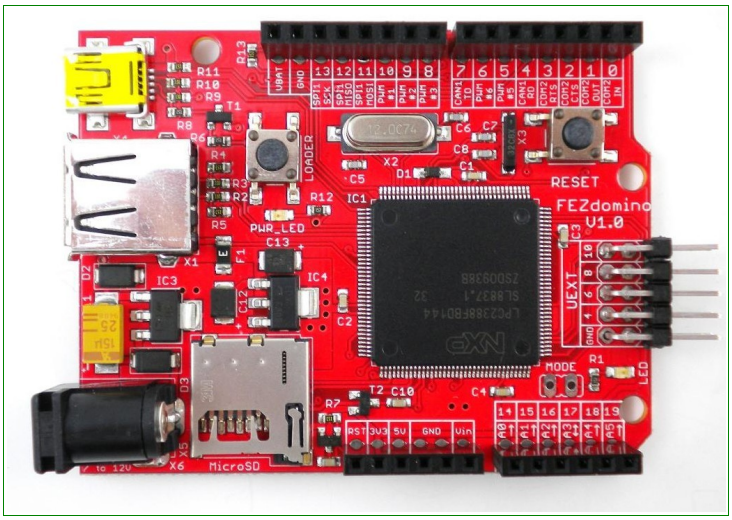
USBixi est le plus petit et le seul processeur mono-chip au monde qui fait tourner NETMF. Le logiciel qui tourne dessus est une version réduite en fonctionnalités par rapport à l'Embedded Master. Il inclut toutes les fonctionnalités sauf le réseau (TCP/IP et PPP) et les graphiques. Même si ces fonctionnalités sont absentes, USBizi peut être connecté à un réseau en utilisant des chipset TCP/IP comme WIZnet et peuvent faire fonctionner des afficheurs simples. Des exemples fournis de projets montrent comment USBizi peut être utilisé sur un réseau et afficher des graphiques.

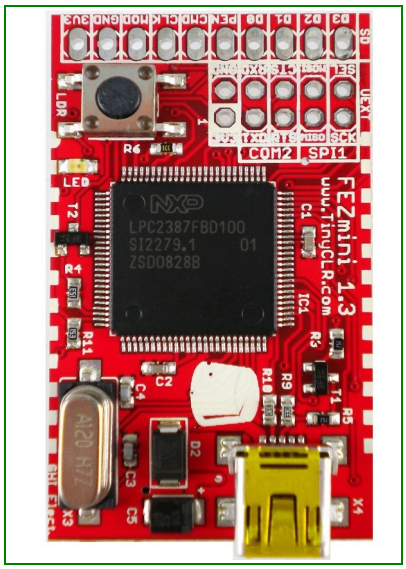


5.4. Gamme FEZ

FEZ Domino et FEZ Mini

FEZ Domino et FEZ Mini sont de très petites cartes open-source destinées aux débutants. Elle sont basées sur le chipset USBizi. FEZ offre plusieurs périphériques comme l'hôte USB ou l'interface SD, généralement non disponibles sur les cartes destinées aux bricoleurs. Bien que les cartes FEZ soient destinées aux débutants, elles sont également un point de départ bon marché pour les professionels désirant explorer NETMF.



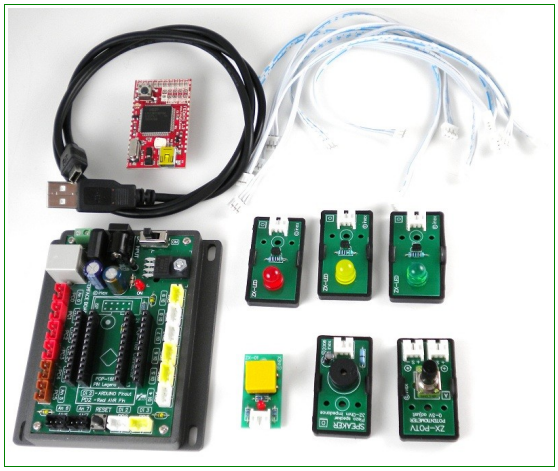


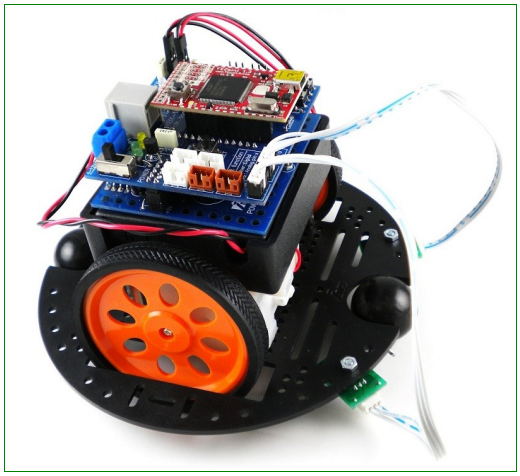
FEZ veut dire “Freakin' Easy!”

FEZ offre plusieurs fonctionnalités non présentes sur les cartes Arduino, BASIC STAMP ou autres :

- Basé sur le .NET Micro Framework de Microsoft
- Tourne sur une processeur ARM NXP à 72Mhz
- Permet le débogage temps-réel (points d'arrêt, inspections de variables, pas-à-pas, etc.)
- Utilise Visual Studio 2008 C# Express Edition pour le développement
- Possibilités avancées comme FAT, périphérique USB device et hôte USB
- Evolue facilement vers des produits comme Embedded Master
- Plans et typons en Open source
- Utilise des modules d'extensions existants (shields)
- Basé sur le chipset USBizi (idéal pour une utilisation commerciale)
- Le brochage de FEZ Mini est compatible avec celui de la carte BS2.
- Celui de FEZ Domino est compatible avec les cartes Arduino.

En utilisant FEZ, les possibilités sont infinies...



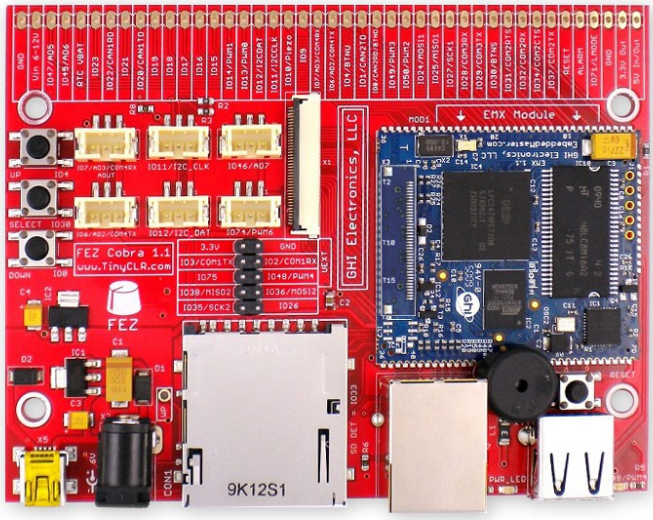


Un nombre impressionnant de capteurs peut se connecter directement sur les kits. Des simples LED et boutons aux capteurs de température ou de distance.

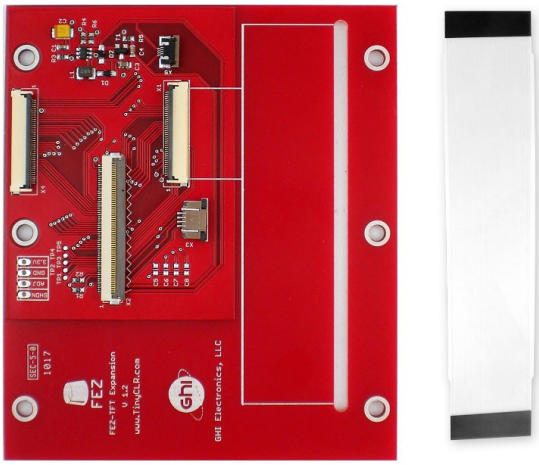
Les exemples présents dans ce livre sont destinés aux cartes FEZ. En règle générale, ces exemples fonctionnent pour le .NET Micro Framework, ce qui ne nécessite que peu de modifications pour le porter sur une autre système NETMF.

FEZ Cobra

FEZ Cobra est une carte open-source basée sur le module EMX. Incluant de multiples écrans optionnels, un boitier optionnel, le support natif des graphiques, le support natif de Ethernet avec SSL et plusieurs MB de mémoire, FEZ Cobra est idéale pour les projets à haute valeur ajoutée. Sans oublier qu'il s'agit encore de FEZ, c'est à dire “Super facile à utiliser”



Une carte d'extension LCD est disponible pour recevoir soit un écran 3,5” (320x240) soit un écran 4,3” (480x272).



6. Comment débuter

Note importante 1: Si vous venez de recevoir votre matériel ou si vous n'êtes pas sûr du firmware chargé dessus, vous DEVEZ mettre à jour le firmware. La documentation (manuel ou tuto) de votre matériel vous montre comment le mettre à jour. C'est une étape requise. Egalement, veuillez vous assurer d'avoir lu la section "Concordance Firmware/Assemblies" de ce livre.

Note importante 2: Actuellement, NETMF ne supporte que Visual Studio 2008 SP1. Visual Studio 2010 ne fonctionne PAS. Microsoft fournira le support pour VS2010 dans NETMF 4.1, prévu pour Juillet/Août 2010.

6.1. Configuration du système

Avant d'essayer quoi que ce soit, nous devons nous assurer que le PC est configuré avec les logiciels nécessaires. Pour celà, il faut commencer par télécharger et installer **Visual C# express 2010** (VS2008 ne fonctionnera pas)

http://www.microsoft.com/express/vcsharp/

Ensuite, téléchargez et installez .NET Micro Framework 4.1 SDK (pas le kit de portage).

http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=cff5a7b7-c21c-4127-ac65-5516384da3a0

Si le lien ci-dessus ne fonctionne pas, cherchez ".NET Micro Framework 4.1 SDK"

Enfin, installez le SDK GHI NETMF. Vous pouvez l'obtenir ici :

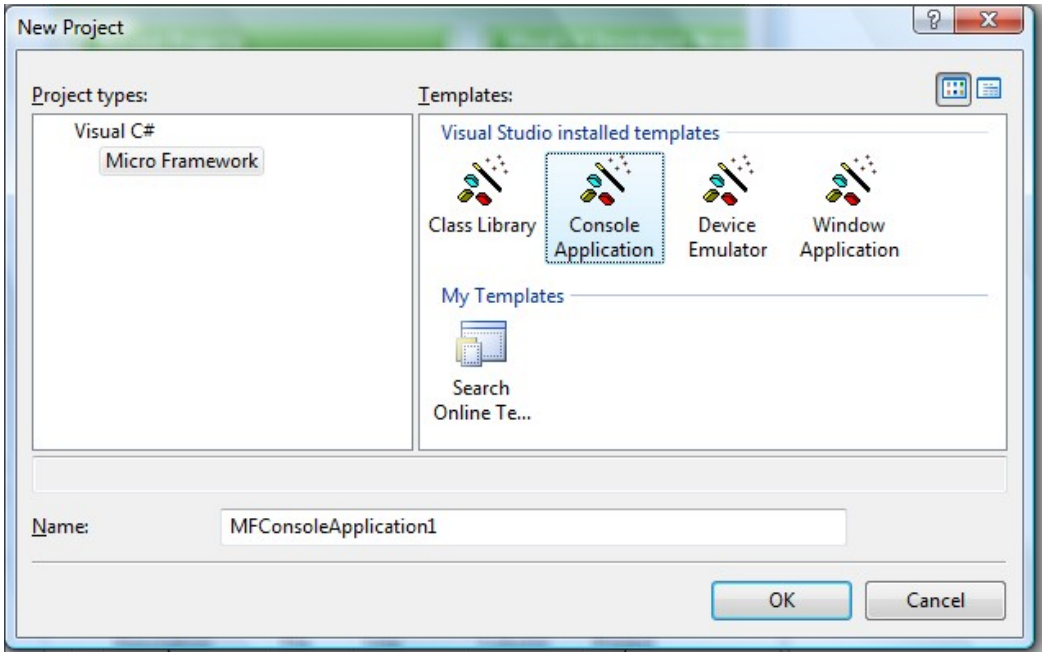
http://www.tinyclr.com/dl/

6.2. L'émulateur

NETMF inclue un émulateur qui permet l'exécution d'application NETMF directement sur le PC. Pour notre premier projet, nous utiliserons l'émulateur pour faire une tourner une application très simple.

Créer un projet

Ouvrez Visual C# express et, depuis le menu, choisissez **Fichier -> Nouveau Projet.** L'assistant doit mentionner "Micro Framework" dans le menu de gauche. Cliquez dessus et, dans les modèles, choisissez "Application console".



Cliquez sur le bouton “OK” et vous aurez un nouveau projet prêt à être exécuté. Le projet contient un seul fichier C#, appelé Program.cs, qui ne contient que très peu de code. Le fichier est visible dans la fenêtre “Explorateur de solution”. Si cette fenêtre n'est pas affichée, vous pouvez l'ouvrir en cliquant sur “Afficher->Explorateur de solution” depuis le menu.

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Debug.Print(
            Resources.GetString(Resources.StringResources.String1));
      }

   }
}
```

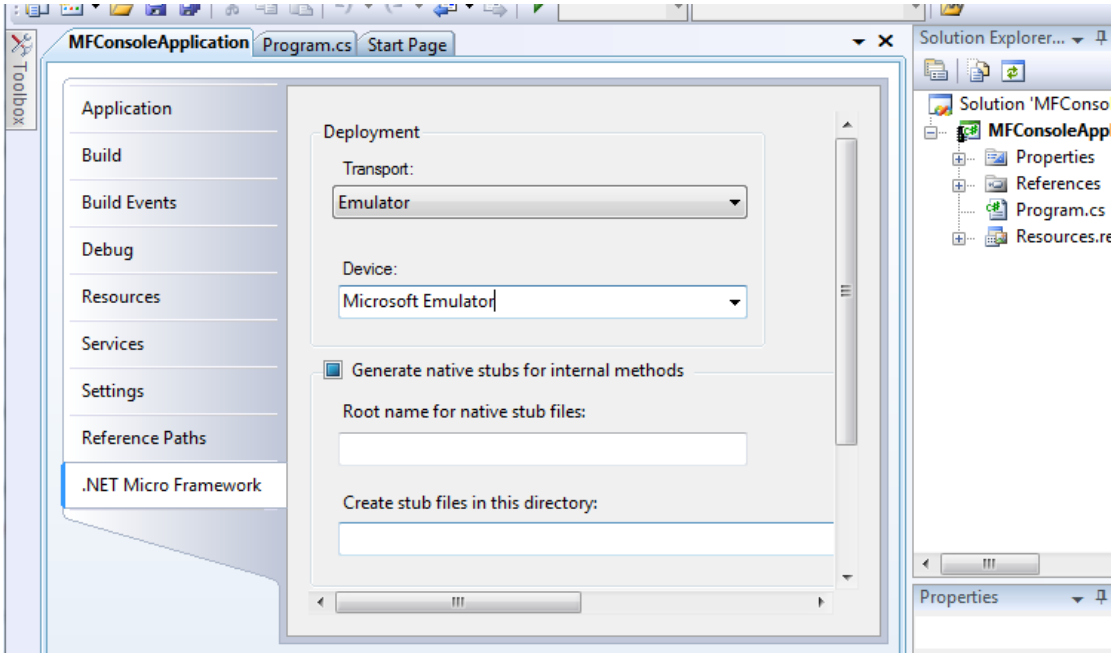
Pour simplifier, changez le code pour qu'il soit identique à celui ci-dessous :

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Debug.Print("Amazing!");
      }

   }
}
```

Choisir le Transport

Pas de panique si vous ne comprenez pas le code. Je l'expliquerai plus tard. Pour l'instant, nous voulons l'exécuter sur l'émulateur. Assurons-nous d'avoir tout configuré correctement. Cliquez sur “Projet->Propriétés” dans le menu. Dans la fenêtre qui s'affiche, on va sélectionner l'émulateur. Sur l'onglet de gauche, sélectionnez “.NET MicroFramework” et assurez vous que la fenêtre ressemble à celle ci-dessous.

Transport: Emulateur



Device: Microsoft Emulator

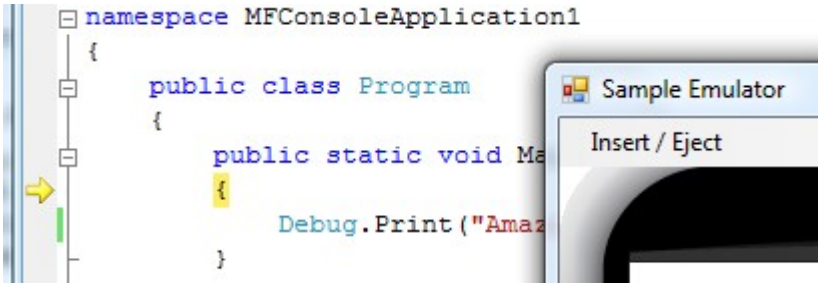
Assurez-vous que la fenêtre de sortie est visible : cliquez sur “Afficher->Sortie”

Exécuter

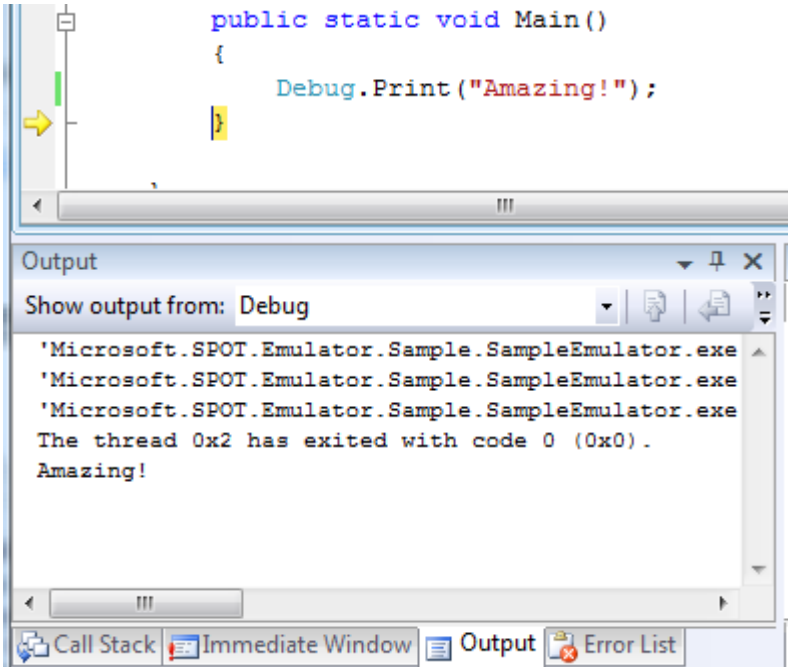
Nous sommes enfin prêt à lancer notre première application. Appuyez sur la touche F5. C'est un raccourci très utile et vous allez l'utiliser très souvent pour exécuter vos applications. Après avoir appuyé sur F5, l'application sera compilée et chargée dans l'émulateur et quelques secondes plus tard, tout s'arrêtera ! C'est parce que notre programme s'est terminé tellement vite qu'on n'a pas vu grand'chose.

Nous allons déboguer le code, maintenant. Déboguer signifie que vous pouvez voir pas-à-pas dans le code ce que fait le programme. C'est une des fonctionnalités majeures de NETMF.

Cette fois, on utilise F11 au lieu de F5, ce qui permettra de suivre à la trace plutôt que simplement exécuter le programme. Cela va déployer l'application sur l'émulateur et s'arrêtera à la toute première ligne de code. Ceci est indiqué par la flèche jaune.



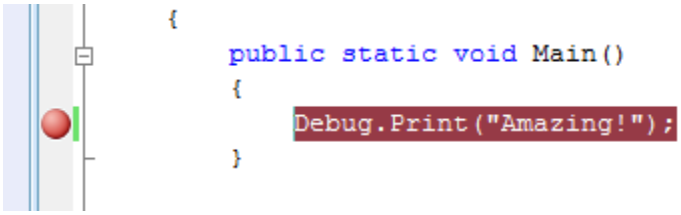
Les applications C# démarrent toujours à partir d'une méthode appelée Main et c'est à cet endroit que la flèche nous arrête. Appuyez de nouveau sur F11 et le debogueur exécutera la ligne de code suivante, que vous avez modifiée précédemment. Vous l'avez probablement déjà deviné, cette ligne écrira “Amazing!” dans la fenêtre debug. La fenêtre debug est la fenêtre de sortie sur Visual C# Express. Assurez-vous que la fenêtre de sortie est visible, comme expliqué précédemment et appuyez de nouveau sur F11. Quand vous allez passer sur cette ligne, le mot “Amazing!” s'affichera dans la fenêtre de sortie.



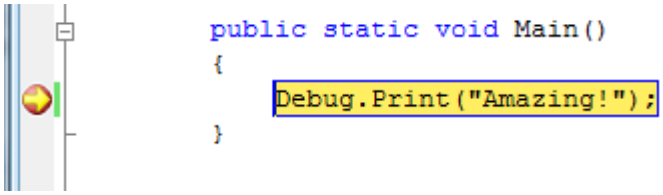
Si vous appuyez sur F11 à nouveau, le programme s'arrêtera et l'émulateur se fermera.

Points d'arrêt

Les points d'arrêt sont une autre fonctionnalité très utile quand vous déboguer du code. Pendant que l'application s'exécute, le débogueur vérifie si l'exécution a atteint un point d'arrêt. Si c'est le cas, l'application se met en pause. Cliquez sur la barre juste à gauche de la ligne qui écrit "Amazing!". Cela affichera un rond rouge, qui est le point d'arrêt.



Maintenant appuyez sur F5 pour exécuter le programme et quand l'application atteindra le point d'arrêt, le débogueur s'arrêtera comme sur l'image ci-dessous.



A cet instant, vous pouvez exécuter pas-à-pas avec F11 ou continuer l'exécution avec F5.

6.3. Exécuter sur le matériel

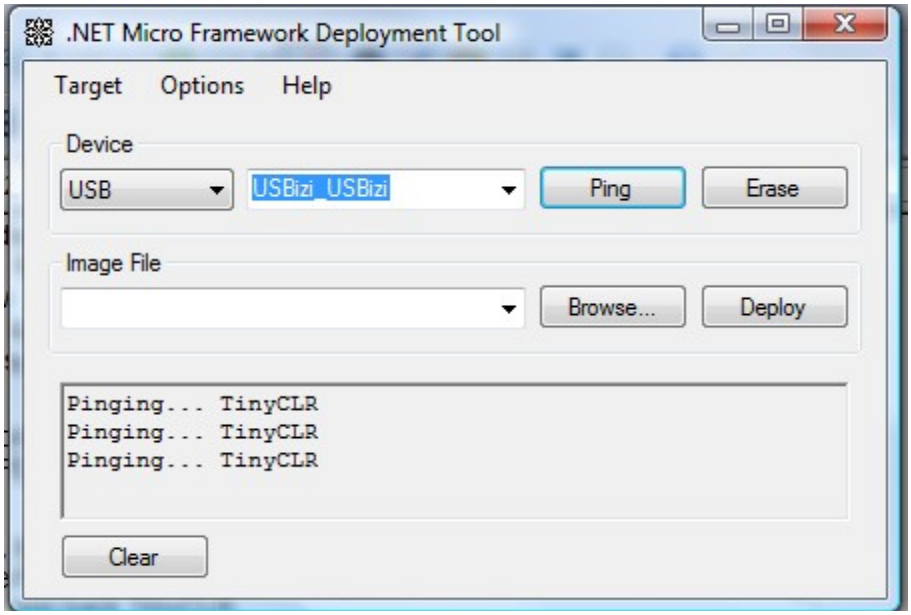
Exécuter des applications NETMF sur le matériel est très simple. La façon de faire diffère sensiblement selon le matériel utilisé, cependant. Ce livre utilise FEZ pour la démonstration mais tout autre matériel suivra plus ou moins la même méthode.

MFDeploy peut envoyer un Ping!

Avant d'utiliser le matériel, assurons-nous qu'il est correctement connecté. Le SDK NETMF est livré avec un programme Microsoft appelé MFDeploy. Il y a plusieurs façons d'utiliser MFDeploy mais pour l'instant nous avons uniquement besoin de “ping-er” la carte. Pour simplifier, MFDeploy va dire “Coucou !” à la carte et vérifier que celle-ci lui répondra également par “Coucou !”. C'est pratique pour vérifier que le matériel est bien connecté et qu'il n'y a pas de problèmes de communications.

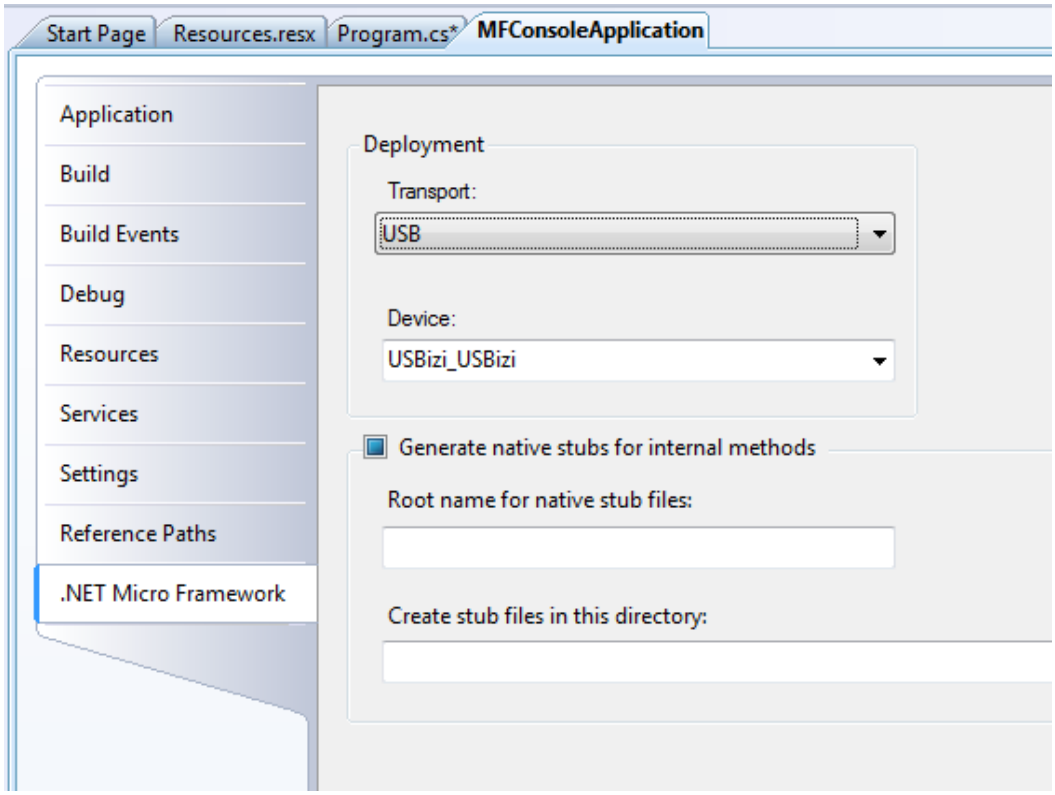
Lancez MFDeploy et connectez FEZ au PC avec le câble USB livré. Si c'ets la première fois que vous branchez FEZ, Windows vous demandera un pilote. Prenez celui présent dans le répertoire d'installation du SDK et attendez que Windows ait fini son installation.

Dans la liste déroulante, sélectionnez USB. Dans la liste des matériels, vous devriez voir USBizi. Vous verrez USBizi parce que FEZ est basé sur le chipset USBizi. Choisissez doc USBizi et cliquez sur le bouton “Ping”. Vosu devriez maintenant voir en retour “TinyCLR”.



Déploiement sur le matériel

Maintenant que nous savons que le matériel est bien connecté grâce à MFDeploy, nous devons retourner dans Visual C# Express. Dans les propriétés du projet, sélectionnez USB pour le transport et USBizi pour le matériel. Assurez-vous que votre configuration ressemble à celle de l'écran ci-dessous.



L'appui sur F5 va maintenant envoyer notre petite application sur FEZ et elle s'exécutera sur le vrai matériel. Basculer de l'émulateur au matériel réel est aussi simple que ça !

Essayez les manipulations décrites avec l'émulateur, comme mettre des points d'arrêt et utiliser F11 pour le pas-à-pas. Notez que “Debug.Print” continuera d'envoyer les messages de débogage provenant du matériel vers la fenêtre de sortie de Visual C# express.

7. Pilotes de composants

Les composants FEZ (LEDs, boutons, capteur de température, relais, pilotes de servos...etc.) et les interfaces FEZ (Ethernet, LCD, pilotes de moteurs...etc.) sont fournis avec des pilotes exemples. Ces pilotes assument le fait que vous ne connaissiez rien au matériel. Par exemple, pour faire clignoter une LED, vous demandez simplement au pilote de le faire. Il n'y a aucune mention de broche du processeur et comment changer l'état d'une broche, etc...

D'un autre côté, ce livre vous apprend les bases. Utilisez les pilotes composants pour démarrer et utilisez ce livre pour comprendre ce que font effectivement ces pilotes.

8. C# Niveau 1

Ce livre n'est pas destiné à enseigner le C# mais couvrira la plupart des bases nécessaires pour démarrer.

Pour que l'apprentissage de C# ne soit pas trop ennuyeux, je ferai une division en plusieurs niveaux pour que nous puissions faire des choses plus intéressantes avec NETMF et revenir au C# quand cela sera nécessaire.

8.1. C'est quoi .NET?

Microsoft a développé le Framework .Net pour standardiser la programmation. (Notez que je parle ici du .Net Framework complet and pas du **Micro**Framework). Il y a des ensembles de librairies que les développeurs peuvent utiliser depuis de nombreux langages. Le .Net Framework fonctionne sur les PC et pas sur les matériels plus petits car c'est un très gros ensemble. Egalement, le framework complet contient plein de choses qui ne seraient pas très utiles sur des petits matériels. C'est ainsi que le .Net Compact Framework a vu le jour. Le framework compact a supprimé les librairies inutiles pour diminuer la taille du framework. Cette version réduite fonctionne sur Windows CE et les smartphones. Cependant, le Compact Framework est encore trop gros pour les petits matériels à cause de sa taille et aussi parce qu'il requiert un OS pour fonctionner.

Le .NET Micro Framework est la plus petite version de ces frameworks. Plus de librairies ont été supprimées et il est devenu indépendant d'un OS. Grâce aux similitudes entre les ces trois framework, le même code peut quasiment tourner sur un PC et un petit matériel; avec peu ou pas de modifications.

Par exemple, utiliser un port série sur un PC, sur un produit WinCE ou sur une carte FEZ (USBizi) se fait de la même manière quand on utilise .NET.

8.2. C'est quoi C#?

C et C++ sont les langages de programmation les plus populaires. C# est une version améliorée et plus moderne de C et C++. Il inclut tout ce qu'on est en droit d'attendre d'un langage moderne comme le “ramasse-miettes” et la validation à l'exécution. Il est également orienté objet, ce qui rend les programmes portables et facilite le débogage Bien que C# applique beaucoup de règles de programmation pour réduire les possibilités de bugs, il offre la majorité des fonctions les plus puissantes de C/C++.

“Main” est le point de départ

Comme nous l'avons vu avant, les programmes démarrent par une méthode appelée Main.

Une méthode est un petit morceau de code qui effectue une certaine tâche. Les méthodes débutent et finissent par des accolades. Dans notre premier programme, nous n'avions qu'une seule ligne de code entre les accolades.
La ligne était Debug.Print("Amazing!");

Vous pouvez voir que la ligne finit par un point-virgule. Toutes les lignes se terminent ainsi. Cette ligne appelle la méthode Print présente dans l'objet Debug. Elle l'appelle en lui fournissant la chaine de caractères “Amazing!”

Troublé ? Essayons de clarifier un petit peu. Supposons que vous soyez un objet. Vous allez avoir plusieurs méthodes pour vous contrôler. Une méthode peut être “Asseoir” et une autre serait “Courir”. Maintenant, comment faire pour vous faire “dire” Amazing ? J'appellerai votre méthode “Dire” avec la phrase (chaine de caractères) “Amazing!”. Et le code donnerait ceci :

Vous.Dire(“Amazing!”);

Pourquoi avons-nous besoin des guillemets autour du mot Amazing ? Parce C# ne sait pas forcément si le texte que vous tapez est une commande ou un vrai texte. Quand c'est le cas, vous verrez la coloration syntaxique mettre ces caractères en rouge, rendant le code plus facile à lire.

Commentaires

Comment faire pour ajouter des notes, commentaires, avertissements dans votre code ?Ces commentaires vous aideront à comprendre ce que fait le code. C# ignore complètement ces commentaires. Il y a deux façons d'écrire des commentaires : ligne ou bloc. Les commentaires sont écrits en vert.
Pour commenter une ligne ou une partie de la ligne, ajouter // devant le texte du commentaire. La couleur du texte deviendra verte et indiquera que le texte est bien un commentaire et sera ignoré par C#.

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         // Ceci est un comentaire
         Debug.Print("Amazing!");//Ce texte également !
      }

   }
}
```


Vous pouvez également faire un commentaire sur un bloc. Commencez votre commentaire par /* et terminez le par */

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         /* Ceci est un commentaire
          Toujours dans le commentaire
          Le block s'arrête ici */
         Debug.Print("Amazing!");
      }

   }
}
```

Boucle while

C'est le moment de notre premier mot clé : “while”. Une boucle while commence et finit par des accolades avec du code entre les deux. Tout ce qui est entre les accolades sera exécuté tant qu'une condition est vérifiée. Par exemple, je peux vous demander de continuer à lire ce livre tant que vous êtes éveillé !
Donc : faisons en sorte que le programme affiche continuellement “Amazing!”. Sans arrêt. Cette boucle n'ayant pas de fin, la condition sera toujours “Vrai” (true).

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         while(true)
         {
            Debug.Print("Amazing!");
         }
      }

   }
}
```


Dans le code ci-dessus, l'exécution démarrera à Main, comme d'habitude et ira ensuite à la ligne suivante qui est la boucle while. Cette boucle dit que le code situé entre les accolades doit s'exécuter tant que la condition est “vraie”. Dans les faits, nous n'avons pas de condition mais directement la proposition “Vrai”, qui indique que la boucle tournera indéfiniment.

N'appuyez par sur F5 sinon vous allez remplir la fenêtre de sortie avec le mot “Amazing!” A la place, appuyez sur F11 et avancer pas-à-pas dans la boucle pour voir comment elle fonctionne. Notez que ce programme ne s'arrêtera jamais, donc vous aurez besoin d'appuyer sur Shift-F5 pour forcer l'arrêt.

Note: Vous pouvez accéder à tous ces raccourcis depuis le menu, option “Débogage”.

Variables

Les variables sont des portions de mémoire réservées pour votre usage. La quantité de mémoire réservée dépend du type de la variable. Je ne vais pas couvrir chaque type ici mais n'importe quel livre sur C# vous l'expliquera en détails.

Nous allons utiliser des variables “int”. Ce type de variable est utilisé pour contenir des nombres entiers.

Donc :

int MyVar;

dira au système que vous voulez qu'il vous réserve de la mémoire. Cette mémoire sera référencée par “MyVar”. Vous pouvez lui donner n'importe quel nom du moment que ce nom ne contient pas d'espaces. Maintenant vous pouvez mettre n'importe quel nombre entier en mémoire :

MyVar = 1234;

Vous pouvez également utiliser des fonctions mathématiques pour calculer d'autres nombres :

MyVar = 123 + 456;

ou incrémenter le nombre de 1 :

MyVar++;

ou le décrémenter de 1 :

MyVar- -;

Avec ça, pouvons-nous écrire un programme qui affiche 3 fois le mot “Amazing!” ? Voici le code :

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
```

```
   public class Program
   {
      public static void Main()
      {
         int MyVar;
         MyVar = 3;
         while(MyVar>0)
         {
            MyVar--;
            Debug.Print("Amazing!");
         }
      }

   }
}
```

Remarquez que la proposition de la boucle while n'est plus tout le temps vraie mais uniquement tant que MyVar>0. Ceci pour dire : continue de boucler tant que la valeur de MyVar est supérieure à 0.
Dans la toute première boucle, MyVar égale 3. Dans chaque bocle ensuite nous décrémentons Myar de un. La boucle tournera donc exactement 3 fois et donc affichera 3 fois le mot “Amazing!”

Faisons quelque chose de plus intéressant. Je veux afficher les nombres de 1 à 10. Ok, nous savons comment créer une variable et comment l'incrémenter, mais comment l'afficher dans la fenêtre de sortie ? Fournir simplement MyVar à Debug.Print donnera une erreur parce que Debug.Print n'accepte que les chaines de caractères, pas les entiers. Comment convertir un entier en chaine ? Très simple : utiliser MyVar.ToString(). Facile !

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         int MyVar;
         MyVar = 0;
         while(MyVar<10)
         {
            MyVar++;
            Debug.Print(MyVar.ToString());
         }
      }

   }
}
```


Dernière chose que nous allons ajouter : nous voulons que le programme écrive :

Comptage: 1

Comptage: 2

...

...

Comptage: 9

Comptage:10

Cela peut être fait très facilement en ajoutant des chaines. Les chaines sont ajoutées en utilisant le symbole +, comme si on ajoutait des nombres entre eux.

Essayez le code suivant :

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         int MyVar;
         MyVar = 0;
         while(MyVar<10)
         {
            MyVar++;
            Debug.Print("Comptage: " + MyVar.ToString());
         }
      }

   }
}
```

Assemblies

Les Assemblies sont des fichiers contenant du code compilé (assemblé). Cela permet au développeur d'utiliser le code tout en n'ayant aucun accès à ce code source. Nous avons déjà utilisé Debug.Print, par ex. Mais qui a créé la classe Debug et qui a écrit la méthode Print à l'intérieur ? Ces appels ont été créés par l'équipe NETMF de Microsoft. Ils compilent le code et mettent à votre disposition une Assembly pour l'utiliser. Ainsi, vous n'avez pas à vous préoccuper des détails internes pour l'utiliser.

Au début du code utilisé précédemment, nous pouvons voir “using Microsoft.SPOT;"

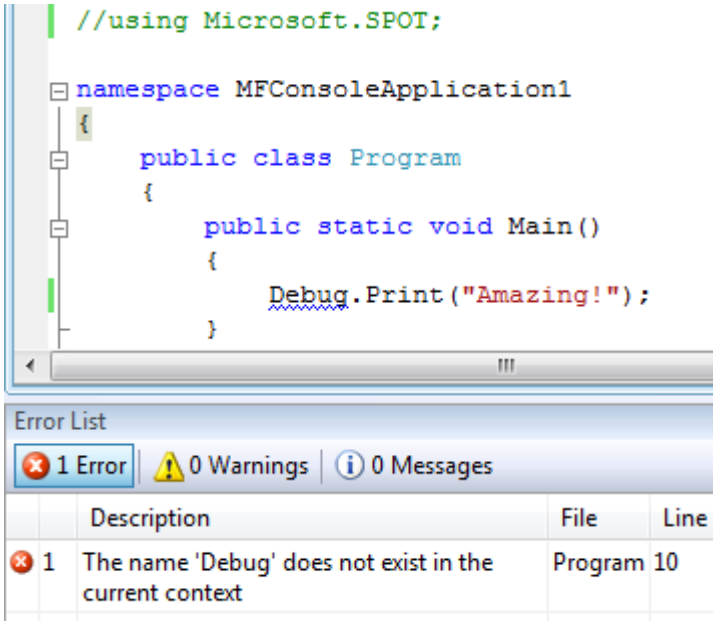
Cela indique à C# que nous voulons utiliser l'espace de nom Microsoft.SPOT. Euh... C'est quoi un “espace de noms” ? Les programmes sont divisés en régions “espaces”. C'est très important quand les programmes deviennent longs. Chaque morceau de code ou de librairie se voit assigner un “nom” dans son “espace”. Les programmes avec le même espace de nom peuvent se voir entre eux mais s'il est différent alors nous pouvons dire à C# d'utiliser (use) l'autre espace de nom.

Le “nom” pour “l'espace” de notre programme est namespace MFConsoleApplication1 Pour utiliser un autre espace nom comme "Microsoft.SPOT" vous devez ajouter "using Microsoft.SPOT;"

C'est quoi SPOT, d'ailleurs ? Voici pour la petite histoire : il y a quelques années, Microsoft a démarré un projet en interne appelé SPOT. Ils ont réalisé que ce projet était une bonne idée et ont voulu le proposer aux autres développeurs. Ils ont alors décidé de changer le nom du produit en .NET Micro Framework mais ont conservé le même code pour des raisons de compatibilité avec l'existant. Donc en gros, SPOT c'est NETMF !

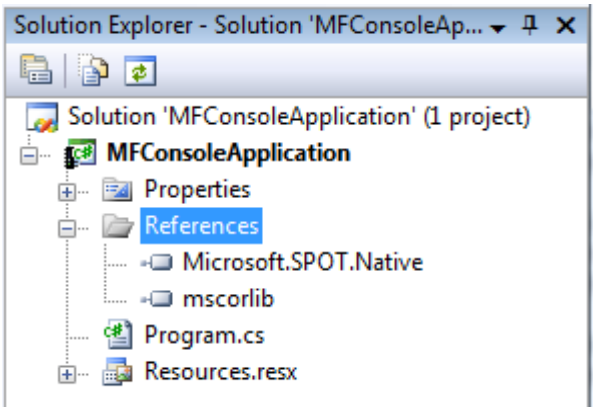
Retour au code. Essayez de supprimer (ou commentez) using Microsoft.SPOT; et votre code ne fonctionnera plus.

Voici le message d'erreur reçu après avoir commenté la ligne using Microsoft.SPOT;

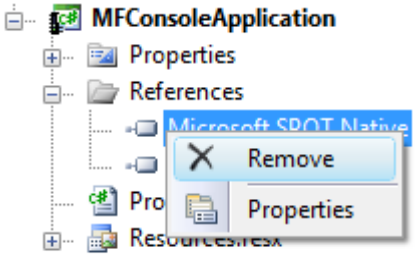


Nous avons utilisé les assemblies mais où ont-elles été ajoutées ?

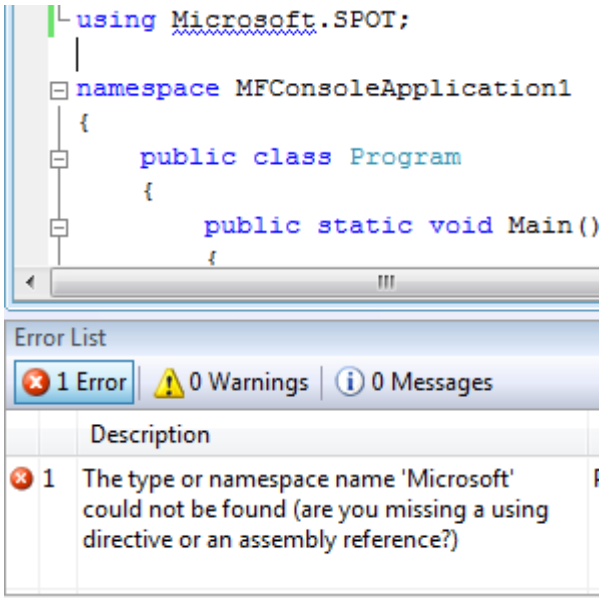
Otez le commentaire et assurez que le programme fonctionne à nouveau. Maintenant, regarder la fenêtre “Explorateur de solution” et cliquez sur le signe + devant le mot “Références”. Vous devriez voir 2 assemblies.



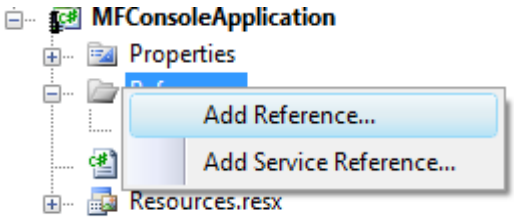
Maintenant, faites un clic-droit sur “Microsoft.SPOT.Native” puis sur “Supprimer”



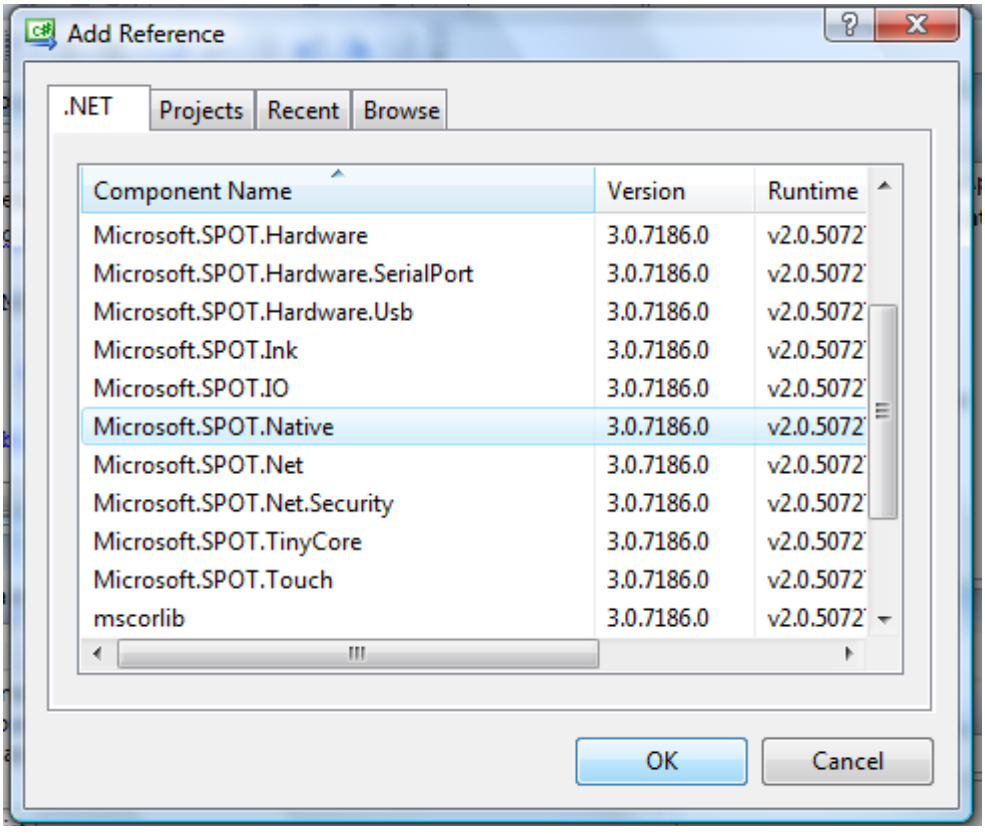
Notre programme est strictement le même qu'avant sauf qu'il lui manque maintenant une assembly très importante. Essayez de le lancer et vous obtiendrez quelque chose comme ceci :



Retournons ajouter cette assembly et vérifions que le programme fonctionne de nouveau. Faites un clic-droit sur le mot “References” et sélectionnez “Ajouter une référence...”



Dans la fenêtre qui vient de s'afficher, sélectionnez l'onglet .NET et choisissez “Microsoft.SPOT.Native”, puis cliquez sur OK.



Essayez le programme, il devrait fonctionner. Si vous avez des erreurs, revenez en arrière dans la lecture et assurez vous d'avoir bien respecté les étapes mentionnées avant de poursuivre plus loin.

Quelles Assemblies ajouter ?

Dans ce livre, je fournis plusieurs exemples mais je ne vous dis pas quelles assemblies

j'utilise. C'est vraiment facile à trouver depuis la doc mais vous pouvez trouver ça difficile, parfois. Pourquoi ne pas tout ajouter d'un coup ? Comme vous débutez, vos applications seront très petites et vous aurez quand même beaucoup de mémoire disponible si vous ajoutez toutes les assemblies, même si vous ne les utilisez pas toutes.

Les assemblies ci-dessous sont les plus utilisées. Ajoutez-les toutes dans vos projets pour l'instant. Une fois que vous saurez quoi prendre et où, vous pourrez supprimer celles dont vous n'avez pas besoin.

GHIElectronics.NETMF.Hardware

GHIElectronics.NETMF.IO

GHIElectronics.NETMF.System

Microsoft.SPOT.Hardware

Microsoft.SPOT.Native

Microsoft.SPOT.Hardware.SeriaPort

Microsoft.SPOT.IO

mscorlib

System

System.IO

N'oubliez pas d'utiliser une des suivantes selon le matériel dont vous disposez. Elles contiennent les assignations des broches du processeur.

FEZMini_GHIElectronics.NETMF.FEZ

FEZDomino_GHIElectronics.NETMF.FEZ.

Multi-Tâches

C'est un sujet très difficile. Seules les informations de base seront couvertes ici.

Les processeurs/programmes n'exécutent qu'une seule instruction à la fois. Vous vous souvenez comment nous allions pas-à-pas dans le code ? Une seule instruction était exécutée et le flux passait à la suivante ensuite. Alors comment est-il possible que votre PC puisse faire tourner plusieurs programmes en même temps ? En fait, le PC n'en fait même pas tourner une ! Il exécute chaque programme durant un court laps de temps, le stoppe et exécute le programme suivant un court laps de temps.

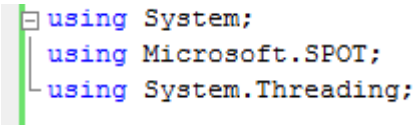
En général, le multi-tâches n'est pas recommandé pour les débutants, mais certaines choses sont plus faciles à faire en utilisant les threads. Par ex, vous voulez faire clignoter une LED.

Ca serait bien que ça se fasse dans un autre thread et que vous n'ayez pas à vous en soucier dans votre programme principal.

En plus, pour ajouter des des délais dans votre programme vous avez besoin de l'espace de nom “Threading”. Vous comprendrez mieux dans les exemples qui suivent. Au passage, LED veut dire Light Emitting Diodes. Il y a des LED partout autour de vous. Prenez n'importe quelle TV, DVD ou appareil électronique et vous verrez une petite lampe rouge (ou autre). Ce sont des LEDs.

FEZ fournit une librairies LED dédiée pour simplifier ça au maximum. Ce livre explique comment contrôler directement les broches et le matériel.

Ajoutez “using System.Threading” à votre programme :



C'est tout ce dont nous avons besoin pour utiliser les threads ! Il est important de savoir que notre programme lui-même est un thread. Au démarrage de l'exécution, C# va chercher Main et la lancer dans un thread. Nous allons ajouter un délai dans notre thread (notre programme) pour qu'il écrive “Amazing!” toutes les secondes. Pour ralentir un thread, on le fait “dormir” (Sleep). Notez que ce repos (Sleep) ne s'applique pas à tout le système mais uniquement au thread concerné.

Ajoutez “Thread.Sleep(1000);

La méthode “Sleep” utilise un temps en in millisecondes. Donc pour 1 seconde, on aura 1000 millisecondes.

```
using System;
using Microsoft.SPOT;
using System.Threading;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         while (true)
         {
            Debug.Print("Amazing!");
            Thread.Sleep(1000);
         }
      }

   }
}
```

Lancez le programme et regardez la fenêtre de sortie. Si vous avez essayé dans l'émulateur et que ce n'était pas tout à fait 1 sec, ne vous inquiétez pas. Essayez sur le vrai matériel (FEZ) et vous serez très proches de 1 sec.

Créons un deuxième thread (le premier était créé automatiquement, vous vous souvenez ?) Nous allons devoir créer un nouveau gestionnaire de thread et lui donner un nom sympa, comme MyThreadHandler, puis créer une nouvelle méthode locale et l'appeler MyThread et enfin lancer le nouveau thread.

Nous n'allons plus utiliser le thread “principal” (Main) alors on le fera dormir à l'infini.

Voici le code. Si vous ne comprenez pas tout, pas d'inquiétude. Le but ici est de savoir comment faire “dormir” (Sleep) un thread.

```
using System;
using Microsoft.SPOT;
using System.Threading;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void MyThread()
      {
         while (true)
         {
            Debug.Print("Amazing!");
            // sleep this thread for 1 seond
            Thread.Sleep(1000);
         }
      }
      public static void Main()
      {
         // create a thread handler
         Thread MyThreadHandler;
         // create a new thread object
         // and assing to my handler
         MyThreadHandler = new Thread(MyThread);
         // start my new thread
         MyThreadHandler.Start();

         ////////////////////////////////
         // Do anythign else you like now here
         Thread.Sleep(Timeout.Infinite);
      }

   }
}
```

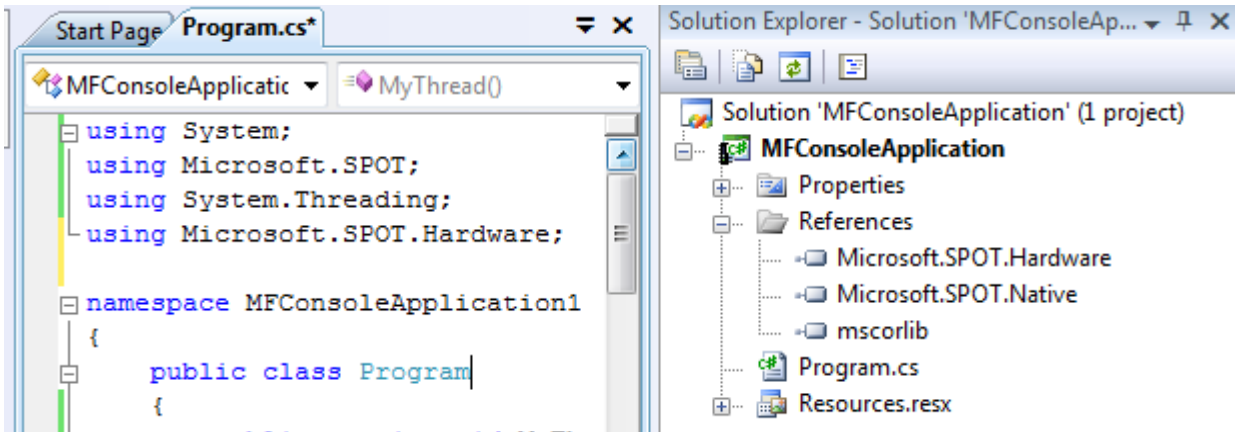
9. Entrées & Sorties numériques

Sur les processeurs, il y a plusieurs broches “numériques” qui peuvent être utilisées en entrée ou en sortie. Quand on dit “numérique”, cela veut dire que l'état de la broche peut être 0 ou 1.

Note importante: L'électricité statique (même celle du corps humain) peut endommager le processeur. Vous avez déjà ressenti une telle décharge en touchant quelqu'un, parfois. Cette petite décharge est assez puissante pour griller des circuits électroniques. Les professionnels utilisent des équipements spéciaux et adaptés pour ne pas émettre ces décharges. Vous n'avez peut-être pas cet équipement, alors évitez de toucher au(x) circuit(s) si vous n'avez pas à le faire. Vous pouvez également utiliser des bracelets anti-statiques.

NETMF fournit les E/S numériques à travers l'assembly et l'espace de nom “Microsoft.SPOT.Hardware”.

Allons-y et ajoutons cette assembly et l'espace de nom comme on l'a vu tout à l'heure.



Nous sommes maintenant prêts à utiliser les broches numériques.

9.1. Sorties numériques

Nous savons qu'une sortie numérique peut être mise à 0 ou 1. Notez qu'il ne s'agit pas de 1 volt mais plutôt que la broche fourni une tension. Si le processeur est alimenté en 3.3V alors l'état 1 sur une broche indique qu'il y a une tension de 3,3V sur cette broche. Ca ne sera pas exactement 3,3V mais très proche. Quand l'état de la broche est mis à 0, alors sa tension sera très proche de 0V.

Ces sorties numériques sont très faibles ! Elles ne peuvent pas alimenter des appareils qui nécessitent beaucoup de puissance.. Par exemple, un moteur peut très bien être alimenté en

3,3V mais vous ne pouvez pas le connecter directement sur la sortie numérique du processeur. Parce que le processeur fournit 3,3V mais avec très très peu de puissance. Le mieux que vous puissiez faire c'est alimenter une LED ou signaler un “1” ou “0” à une autre broche.

Toutes les cartes FEZ ont une LED connectée à une sortie numérique. Nous allons faire clignoter cette LED.

Les sorties numériques sont contrôlées par un objet OutputPort. Nous créons d'abord le gestionnaire de l'objet (référence) puis nous créons un nouvel objet OutputPort et l'assignons à notre gestionnaire. Quand vous créez un nouvel objet OutputPort, vous devez spécifier l'état initial de la broche : 0 ou 1, respectivement bas/haut ou également **true** (haut) et **false** (bas). Nous allons mettre la broche à vrai (état haut) dans cet exemple pour allumer la LED au démarrage.

Voici le code, qui utilise la broche 4, reliée à la LED sur la carte FEZ Domino.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         LED = new OutputPort((Cpu.Pin)4, true);

         Thread.Sleep(Timeout.Infinite);
      }

   }
}
```

Le SDK FEZ est fourni avec les assemblies“FEZMini_GHIElectronics.NETMF.FEZ” et “FEZDomino_GHIElectronics.NETMF.FEZ”. Ajoutez celle dont vous avez besoin à votre programme et ajoutez également “FEZ_GHIElectronics.NETMF.System”.

Maintenant, ajoutez “using GHIElectronics.NETMF.FEZ” au début de votre code.

Voici le code, cette fois en utilisant l'énumération des broches présente dans l'assembly

```
using System;
using Microsoft.SPOT;
using System.Threading;
```

```
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);

         Thread.Sleep(Timeout.Infinite);
      }
   }
}
```

Vous voyez comme c'est plus facile ? Nous n'avons pas à nous préoccuper de savoir sur quelle broche est connectée la LED..

Lancez le programme et regardez la LED. Elle doit être allumée. Ca devient un peu plus excitant !

Faire clignoter une LED

Pour faire clignoter une LED, nous devons mettre la broche à l'état haut, inclure un délai, la mettre à l'état bas et inclure un autre délai. C'est important d'inclure ce dernier délai. Pourquoi ? Parce que nos yeux ne sont pas assez rapides pour détecter le changement si la LED est allumée et éteinte aussitôt.

De quoi avons-nosu encore besoin ? ... nous avons appris la boucle while, comment insérer un délai, il ne nous reste plus qu'à savoir comment mettre la broche à l'état haut/bas. Cela se fait en appelant la méthode Write de l'objet OutputPort. Notez que vous ne pouvez pas utiliser directement “OutputPort.Write” car à cet instant, on ne sait pas de quel port il s'agit. A la place, il faut utiliser LED.Write, ce qui a plus de sens, finalement.

Voici le code pour faire clignoter la LED de la carte FEZ Domino/Mini

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
```

```
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            while (true)
            {
                LED.Write(!LED.Read());

                Thread.Sleep(200);
            }
        }
    }
}
```

Il y a une autre manière, plus simple :

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            while (true)
            {
                LED.Write(true);
                Thread.Sleep(200);

                LED.Write(false);
                Thread.Sleep(200);
            }
        }
    }
}
```

Essayez de changer la valeur du délai pour faire clignoter la LED plus ou moins rapidement. Essayez aussi des valeurs différentes pour l'état haut ou bas.

Note importante: Ne connectez jamais deux sorties ensemble. Si elles sont connectées et que l'une est à l'état haut et l'autre à l'état bas, vous endommagerez le processeur. Connectez toujours une sortie sur une entrée, pour piloter un circuit ou une LED.

9.2. Entrées numériques

Les entrées numérique détectent si l'état d'une broche est haut ou bas. Il y a quelques limitations sur ces Entrées. Par ex, la tension minimale est 0V. Une tension négative peut endommager la broche du processeur. Egalement, la tension maximale doit être inférieure à l'alimentation du processeur. Toutes les cartes GHI Electronics utilisent des processeurs alimentés en 3,3V donc la tension maximale sur une broche devrait être 3,3V. Ceci est vrai pour ChipworkX mais pour Embedded Master et USBizi, les processeurs tolèrent 5V. Cela signifie que même si le processeur est alimenté en 3,3V, il est capable d'encaisser 5V sur ses entrées. La plupart des circuits numériques que vous pouvez utiliser sont en 5V. Tolérer les 5V permet donc d'utiliser tous ces circuits avec notre processeur.

Note: FEZ est basé sur USBizi et est donc compatible 5V.

Note importante: Compatible 5V ne veut pas dire que le processeur peut être alimenté en 5V. Alimentez-le toujours en 3,3V. Seules les broches d'entrées tolèrent 5V.

L'objet InputPort est utilisé pour gérer les entrées numériques. Toutes les broches sur les processeurs GHI peuvent être des entrées ou des sorties, mais bien sûr pas en même temps ! Les entrées non connectées sont dites “flottantes”. Vous pourriez penser qu'une entrée non connectée est à l'état bas mais ce n'est pas le cas. Quand une entrée n'est pas connectée, elle est ouverte à tout bruit ambiant qui peut la rendre haute/basse. Pour pallier à ça, les processeurs modernes incluent en interne des résistances pull-up ou pull-down qui sont généralement contrôlée par le logiciel. Activer la résistance pull-up va tirer la broche vers l'état haut. Notez que ça ne met pas la broche à l'état haut mais plutôt la force dans un état proche de haut. Si rien n'est connecté dessus, l'état lu sera haut.

Il y a plein d'utilisation pour les ports d'entrée mais la plus classique est d'y connecter un bouton ou un interrupteur. FEZ inclut déjà un bouton connecté à la broche “Loader”. Cette broche est utilisée à la mise sous tension pour activer le téléchargement au démarrage mais nous pouvons utiliser cette broche librement ensuite. Le bouton s'appelle “LDR” ou “Loader”.

Le bouton connectera la masse à la broche d'entrée. Nous allons également connecter la résistance pull-up. Cela signifie que la broche sera “haut” (pull-up) quand le bouton sera relâché et “bas” (connecté à la masse) quand le bouton sera pressé.

Nous allons lire l'état du bouton et le passer à la LED. La LED s'éteindra quand le bouton sera pressé.

Le code:

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
```

```
public class Program
{
   public static void Main()
   {
      OutputPort LED;
      InputPort Button;
      LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
      Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
      while (true)
      {
         LED.Write(Button.Read());
         Thread.Sleep(10);
      }
   }
}
}
```

Créer un port d'entrée nécessite un paramètre supplémentaire pour le filtre anti-rebonds. Ceci sera expliqué ultérieurement. Egalement, la manière dont nous avons passé l'état d'un port d'entrée pour activer un port de sortie peut paraitre surprenante. Nous y reviendrons dans la prochaine section.

9.3. Port d'interruption

Si nous voulons connaitre l'état d'une broche, nous devons vérifier son état périodiquement. Cela gâche du temps processeur pour une tâche qui n'est pas importante. Vous allez vérifier une broche peut-être un million de fois avant que le bouton ne soit pressé ! Les ports d'interruptions nous permettent d'écrire une méthode qui sera appelée uniquement quand le bouton sera pressé (broche “bas”, par exemple).

Nous pouvons déclencher l'interruption sur différents changements d'états de la broche, quand elle est “bas” ou “haut” par exemple. L'utilisation courante est “on change” (à chaque changement). Le changement de bas à haut ou haut à bas crée un flanc sur le signal. Le flanc haut apparait quand le signal passe de bas à haut et le flanc bas quand le signal passe de haut à bas.

Dans l'exemple ci-dessous, j'utilise les deux flancs, ainsi notre méthode “IntButton_OnInterrupt” sera appelée dès que l'état de la broche change.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
```

```
    {
        static OutputPort LED; // déplacé ici pour pouvoir être utilisé par d'autres méthodes

        public static void Main()
        {
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            // la broche génèrera une interruption à chaque flanc haut et bas
            InterruptPort IntButton = new InterruptPort((Cpu.Pin)FEZ_Pin.Interrupt.LDR, true,
                Port.ResistorMode.PullUp, Port.InterruptMode.InterruptEdgeBoth);

            // ajoute un gestionnaire d'interruption à la broche
            IntButton.OnInterrupt += new NativeEventHandler(IntButton_OnInterrupt);

            //faites ce que vous voulez ici
            Thread.Sleep(Timeout.Infinite);
        }

        static void IntButton_OnInterrupt(uint port, uint state, DateTime time)
        {
            // met la LED dans l'état de la broche
            LED.Write(state == 0);
        }
    }
}
```

Note: la plupart des broches du processeur supporte les interruptions, mais pas toutes. Pour une meilleure identification des broches concernées, utilisez l'énumération “Interrupt” au lieu de “Digital” comme montré dans un des codes précédents.

9.4. Ports Triple-état

Comment faire si nous voulons une broche qui soit à la fois entrée et sortie ? Une broche ne peut jamais être les deux simultanément mais nous pouvons la rendre “sortie” pour faire quelque chose et la rendre “Entrée” pour lire une réponse. Une façon de faire serait de “Disposer” la broche : on la désigne “Sortie”, on l'utilise et on la dispose. Là on peut la désigner “Entrée” et la lire.

NETMF propose de meilleures options pour ça, à travers les ports “Triple-état” (tristate ports). Les trois états possibles sont : entrée, sortie basse et sortie haute. Le seul souci avec ces broches triple-état est que si une broche est “Sortie” et que vous la désigner à nouveau comme “Sortie”, alors vous recevrez une exception. Une possiblité pour éviter cela est de vérifier la direction de la broche avant de la changer. La direction se trouve dans sa propriété “Active” ou false signifie Entrée et true Sortie. Personnellement, je ne recommande pas l'utilisation de tels ports à moins d'absolue nécessité.

```
using System;
using Microsoft.SPOT;
using System.Threading;
```

```
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      static void MakePinOutput(TristatePort port)
      {
         if (port.Active == false)
            port.Active = true;
      }
      static void MakePinInput(TristatePort port)
      {
         if (port.Active == true)
            port.Active = false;
      }
      public static void Main()
      {
           TristatePort TriPin = new TristatePort((Cpu.Pin)FEZ_Pin.Interrupt.LDR, false, false,
Port.ResistorMode.PullUp);
         MakePinOutput(TriPin);// Broche en sortie
         TriPin.Write(true);
         MakePinInput(TriPin);// Broche en entrée
         Debug.Print(TriPin.Read().ToString());
      }
   }
}
```

Note: par conception, les ports Triple-état ne fonctionnent qu'avec des broches générant des interruptions.

Note importante : attention de ne pas avoir la broche connectée à un interrupteur et la passer en sortie haute. Cela endommagera le processeur. Je dirais que pour les applications de débutants vous n'avez pas besoin des ports triple-état. Donc ne les utilisez pas tant que vous n'êtes pas au point avec les circuits numériques de base.

10. C# Niveau 2

10.1. Variables booléennes

Nous avons appris comment des variables integer pouvaient contenir des nombres. Les variables booléennes, elles, ne peuvent contenir que true ou false (vrai ou faux). Une lampe peut être allumée ou éteinte et représenter cet état avec un entier n'aurait guère de sens alors qu'en utilisant un booléen on peut dire que true égale allumé et false égale éteint. Nous avons d'ailleurs déjà utilisé ce type de variable pour changer l'état de broches digitales en haut ou bas : LED.Write(true);

Pour stocker la valeur d'un bouton dans une varibale, on utilise :

```
bool button_state;
button_state = Button.Read();
```

Nous avons aussi utilisé des boucles while et demandé qu'elles bouclent à l'infini quand nous avons utilisé "true" dans la condition.

```
while (true)
{
   // votre code ici
}
```

Prenez le dernier code que nous avons écrit et modifiez-le en utilisant un booléan pour qu'il soit plus facile à lire. Plutôt que de passer l'état du bouton directement à la LED, nous allons stocker l'état du bouton dans une variable booléenne et passer ensuite cette variable à la LED.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
```

```
        InputPort Button;
        bool button_state;
        LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
        Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
        while (true)
        {
            button_state = Button.Read();
            LED.Write(button_state);
            Thread.Sleep(10);
        }
    }
  }
}
```

Pouvez-vous faire clignoter une LED tant qu'on appuie sur le bouton ?

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
  public class Program
  {
    public static void Main()
    {
      OutputPort LED;
      InputPort Button;

      LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
      Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
      while (true)
      {
        while (Button.Read() == false)// L'état du bouton est false quand il est appuyé
        {
          LED.Write(true);
          Thread.Sleep(300);
          LED.Write(false);
          Thread.Sleep(300);
        }
      }
    }
  }
}
```

Note importante: Le signe "==" est utilisé en C# pour tester une égalité alors que le signe "=" est utilisé pour assigner une valeur.

10.2. Mot-clé if

Une part importante de la programme consiste à vérifier des états et agir en conséquence. Par ex, “si la température dépasse 35°, alors met le ventilateur en marche”.

Pour essayer le mot-clé if avec notre petit programme, nous allons allumer la LED “si” le bouton est appuyé. Notez que c'est l'inverse de tout à l'heure. Car dans le programme, l'état du bouton est bas quand il est appuyé. Donc, pour arriver à cela, nous allons inverser l'état de la LED par rapport à celui du bouton. Si le bouton est pressé (bas) alors on allume la LED (high). Cela doit être vérifié régulièrement, donc nous allons le faire toutes les 10ms.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         InputPort Button;
         bool button_state;
         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
         Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
         while (true)
         {
            button_state = Button.Read();

            if (button_state == true)
            {
               LED.Write(false);
            }

            if (button_state == false)
            {
               LED.Write(true);
            }

            Thread.Sleep(10);
         }
      }
   }
}
```

10.3. Mots-clés if et else

Nous venons de voir comment fonctionne un “if”. Maintenant, nous allons utiliser le mot-clé “else”. Simplement, si un “if” est vrai alors le code associé est exécuté ou “sinon” (else) le code de la partie “else” est exécuté. Avec ce nouveau mot-clé, nous pouvons optimiser notre code :

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         InputPort Button;
         bool button_state;
         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
         Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
         while (true)
         {
            button_state = Button.Read();

            if (button_state == true)
            {
               LED.Write(false);
            }
            else
            {
               LED.Write(true);
            }

            Thread.Sleep(10);
         }
      }
   }
}
```

Je vais vous dire un secret ! Nous avons utilisé les mots-clé if et else ici juste pour la démonstration. Car nous pouvons écrire le même code de cette manière :

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;
```



```
namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         InputPort Button;

         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
         Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
         while (true)
         {
            LED.Write(Button.Read() == false);
            Thread.Sleep(10);
         }
      }
   }
}
```

Ou même comme ceci :

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         OutputPort LED;
         InputPort Button;

         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
         Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false, Port.ResistorMode.PullUp);
         while (true)
         {
            LED.Write(!Button.Read());
            Thread.Sleep(10);
         }
      }
   }
}
```

En général, il y a plusieurs façon d'écrire un code. Utilisez ce qui vous parait le plus facile au départ, et avec l'expérience vous apprendrez à optimiser votre code.

10.4. Méthodes et Arguments

Les méthodes sont des actions exécutées par un objet. On les appelle également des “fonctions” d'un objet. Nous avons déjà vu et utilisé des méthodes. Rappelez-vous l'objet Debug avec sa méthode Print. Nous avons déjà souvent utilisé Debug.Print, à laquelle nous avons donné une chaine à afficher. Cette “chaine” que nous avons passé s'appelle un “argument”

Les méthodes peuvent avoir un ou plusieurs arguments optionnels mais ne peuvent renvoyer qu'une seule valeur optionnelle.

La méthode Print de l'objet Debug prend un seul argument. D'autres méthodes n'ont pas besoin d'arguments ou en prenne plusieurs. Par ex, une méthode qui dessinerait un cercle peut prendre 4 arguments : Cercle(CentreX, CentreY, Diametre, Couleur). Un exemple de méthode renvoyant une valeur serait une méthode qui fournit une température.

Jusqu'ici, nous avons appris 3 types de variables : int, string et bool. Nous verrons d'autres types plus tard, mais souvenez-vous que tout ce que nous disons ici s'applique aussi à ces autres types.

La valeur renvoyée est optionnelle. Si aucune valeur n'est renvoyée alors on utilise le type “void”. Le mot-clé “return” est utilisé pour renvoyer des valeurs à la fin d'une méthode.

Voici une méthode simple qui renvoie la somme de deux entiers :

```
int Add(int var1, int var2)
{
   int var3;
   var3 = var1 + var2;
   return var3;
}
```

Nous commençons par indiquer le type de la valeur renvoyée (int) suivi du nom de la méthode. Puis nous avons la liste des arguments. Les arguments sont groupés entre parenthèses et séparés par une virgule.

A l'intérieur de la méthode Add, une variable entière locale a été déclarée : var3. Les variables locales n'existent que dans la méthode dans laquelle elles sont déclarées. Ensuite nous ajoutons nos deux arguments et enfin renvoyons le résultat.

Comment faire pour renvoyer une chaine de caractère représentant la somme de 2 nombres ? Souvenez-vous qu'une chaine contenant “123” n'est pas la même chose qu'un entier valant 123. Un integer est un nombre et une chaine de caractères représente un texte. Pour les humains c'est la même chose alors que pour un ordinateur ce sont deux choses totalement différentes

Voici le code pour renvoyer une chaine de caractères :


```
string Add(int var1, int var2)
{
   int var3;
   var3 = var1 + var2;

   string MyString;
   MyString = var3.ToString();

   return MyString;
}
```

Nous pouvons voir comment le type renvoyé change pour une chaine. Nous ne pouvions pas renvoyer directement var3 car c'est un entier et donc nous avons dû le convertir en chaine. Pour faire cela, nous avons créé un nouvel objet de type chaine appelé MyString, puis converti var3 en chaine ("ToString()") et placé ce résultat dans MyString.

La question qu'on peut se poser est comment est-ce possible d'appeler une méthode "ToString()" sur une variable entière ? En fait, dans C#, tout est représenté par un objet, même les types de variables. Ce livre ne rentre pas dans ces détails car il est destiné avant tout aux débutants.

Tout ce qui a été écrit avant a été détaillé pour l'explication, mais on peut en fait compacter l'écriture pour obtenir le même résultat :

```
string Add(int var1, int var2)
{
   return (var1+var2).ToString();
}
```

Je ne vous conseille pas d'écrire du code si compact tant que vous ne serez pas familiarisé avec le langage. Et quand bien même, il y a certaines limites à respecter si on veut que le code reste lisible ou qu'il soit ré-utilisé par quelqu'un d'autre facilement.

10.5. Les classes

Tous les objets dont nous avons parlé jusqu'ici sont en faites des “classes” en C#. Dans les langages modernes orientés objets, tout est un objet et les méthodes appartiennent toutes à un objet. Cela permet d'avoir des méthodes aux noms identiques mais dont les actions sont complètement différentes. Un humain peut “marcher” et un chat peut aussi “marcher”, sauf qu'il ne le fait pas de la même manière. Si vous écrivez une méthode “Marcher” en C# alors ce n'est pas très clair s'il s'agit de l'humain ou du chat. Sauf si vous utilisez : Humain.Marcher ou Chat.Marcher.

La création de classes n'est pas l'objectif de ce livre, mais voici quand même comment déclarer une classe simple :


```
class MaClasse
{
   int Add(int a, int b)
   {
      return a + b;
   }
}
```

10.6. Public / Private

Les méthodes peuvent être dédiée à une classe (private) ou accessibles publiquement (public). Ceci est utile pour renforcer la programmation et éviter les confusions. Si vous créeez un objet avec des méthodes que vous ne désirez pas que le programmeur utilise, alors il vous suffit d'ajouter le mot-clé “private” devant le type de la méthode. Sinon, mettez le mot-clé “public”.

Voici un exemple :

```
class MyClass
{
   public int Add(int a, int b)
   {
      // L'objet peut utiliser des méthodes "private"
      // uniquement à l'intérieur de la classe
      DoSomething();
      return a + b;
   }
   private void DoSomething()
   {
   }
}
```

10.7. Static et non-static

Certains objets dans la vie ont plusieurs instances alors que d'autres n'existent qu'une seule fois. Par ex, un objet représentant un humain ne veut pas dire grand'chose. Vous devrez créer une “instance” de cet objet pour représenter un humain. Ca ressemblerait à

human Mike;

Nous avons maintenant une “référence” appelée Mike, de type Humain. Il est important de noter à ce stade que cette référence ne représente rien en soit (pas d'instance créée), donc

elle est NULL.

Pour créer réellement l'instance, on utilise le mot-clé “new” sur la référence en spécifiant la classe de l'objet :

Mike = new human();

Nous pouvons maintenant utiliser les méthodes “humaines” sur l'instance Mike :

Mike.Run(distance);

Mike.Eat();

bool hungry = Mike.IsHungry();

Nous avons déjà utilisé de telles méthodes non statiques quand nous avons contrôlé les broches d'E/S.

Quand nous créons un nouvel objet non statique, le mot-clé “new” est utilisé pour appeler le constructeur de l'objet. Le constructeur est un type spécial de méthode qui ne renvoie aucune valeur et qui utilisé exclusivement lors de la construction/création de l'objet.

Les méthodes statiques sont plus facile à gérer car elles font référence à un seul objet sans en créer d'instance. L'exemple typique est notre objet Debug : il y a un seul objet Debug dans NETMF. Donc pour utiliser ses méthodes, on les appelle directement depuis l'objet :

Debug.Print(“chaine”);

Je n'ai peut-être pas utilisé les définitions exactes pour “static” et “instances” mais je voulais faire une description la plus simple possible.

10.8. Constantes

Certaines variables peuvent avoir des valeurs qui ne changent jamais. Pour se protéger d'une éventuelle erreur de programmation qui changerait ces valeurs, il suffit d'ajouter le mot-clé “const” devant le type de la variable, lors de sa déclaration :

```
const int heures_dans_un_jour = 24;
```

10.9. Enumérations

Les énumérations sont similaires aux constantes. Admettons que nous ayons un appareil qui accepte 4 commandes : MOVE, STOP, LEFT, RIGHT. Cet appareil n'est pas un humain, donc ces commandes sont en fait des nombres. Nous pouvons créer des constantes pour ces 4 commandes :


```
const int MOVE = 1;
const int STOP = 2;
const int RIGHT = 3;
const int LEFT = 4;
// maintenant nous pouvons envoyer une commande...
SendCommand(MOVE);
SendCommand(STOP);
```

Les noms des constantes sont en majuscules par convention. Tout programmeur voyant une variable en majuscules sait qu'il a à faire à une constante.

Le code ci-dessus est bien et fonctionne, mais il serait encore mieux si nous regroupions ces commandes :

```
enum Command
{
   MOVE = 1,
   STOP = 2,
   RIGHT = 3,
   LEFT = 4,
}
// maintenant nous pouvons envoyer une commande...
SendCommand(Command.LEFT);
SendCommand(Command.STOP);
```

Avec cette nouvelle approche, il n'y a plus besoin de se souvenir des commandes existantes et que ce sont des nombres. Dès que le mot “Command” est tapé, Visual Studio vous proposera la liste des commandes disponibles.

C# est également assez sympa pour incrémenter lui-même le nombre à chaque nouvelle commande. Donc le code ci-dessous fonctionne exactement de la même manière

```
enum Command
{
   MOVE = 1,
   STOP ,
   RIGHT,
   LEFT ,
}
// maintenant nous pouvons envoyer une commande...
SendCommand(Command.LEFT);
SendCommand(Command.STOP);
```


11. Correspondance Assembly/Firmware

Les appareils NETMF incluent généralement des fonctions étendues. Ces fonctions étendues nécessitent une assembly/librairie supplémentaire pour qu'un projet C# puisse les utiliser. Par exemple, NETMF ne gère pas les broches analogiques, mais FEZ et d'autres matériels de GHI le font. Pour utiliser ces broches analogiques, vous devez donc ajouter une assembly fournie par GHI et avoir ainsi accès à ces nouvelles possibilités.

Note importante: Le firmware ne fonctionnera pas si l'assembly/librairie ne correspond pas à la version du firmware installé.

C'est un problème très courant que l'utilisateur rencontre quand il met à jour son firmware et que l'application ne fonctionne plus.

Voici ce qui se passe :

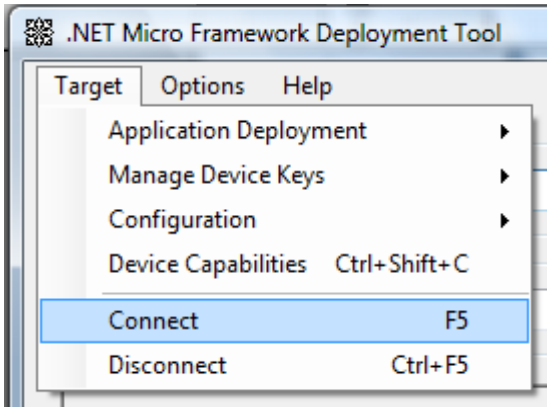
Scenario #1: Un développeur reçoit un nouveau matériel. Celui-ci a le firmware version 1.0. Le développeur va ensuite sur le site GHI et charge le dernier SDK, qui a la version 1.1. Quand vous allez essayer de charge votre programme, VS2008 ne pourra pas se connecter au matériel et ne dira pas pourquoi ! Le développeur pensera alors que son matériel ne fonctionne pas. En fait, son produit fonctionne bien, mais dans cette exemple, le firmware est 1.0 et l'assembly 1.1, donc ça ne vas pas. Pour résoudre ce problème, il faut donc mettre à jour le firmware.

Scenario #2: Un développeur a un matériel qui fonctionne bien, avec un firmware 2.1. Un nouveau SDK 2.2 sort, et le développeur met à jour le firmware en conséquence. Quand il redémarre son appareil, ça ne fonctionne pas ! Parce que le programme dans le processeur est toujours à la version 2.1. Pour résoudre ce problème, ouvrez le projet ocncerné et retirez les assemblies spécifiques au matériel puis ajoutez-les de nouveau. Ainsi le nouveau SDK sera pris en compte lors de la compilation.

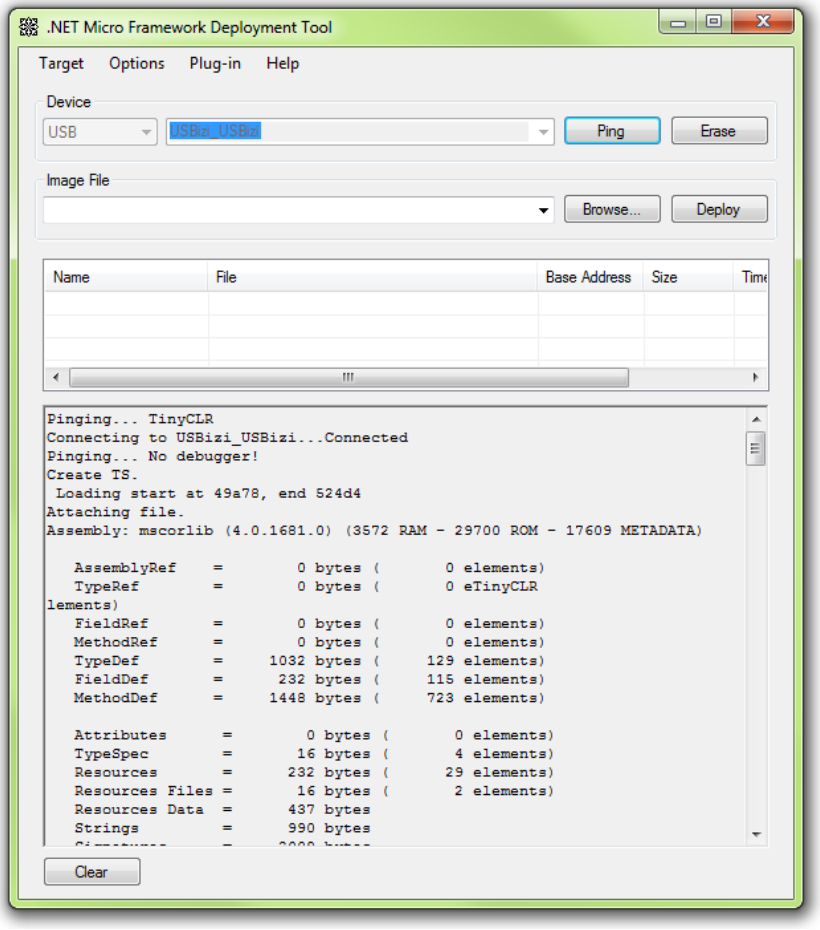
Messages de démarrage

Nous pouvons facilement voir pourquoi le système ne fonctionne pas en utilisant MFDeploy. NETMF affiche beaucoup de messages très utiles au démarrage. Que le système se bloque, ne fonctione pas ou pour tout autre souci de débogage, nous pouvons utiliser MFDeploy pour visualiser ces messages de démarrage. Egalement, tous les messages envoyés par “Debug.Print” et que nous voyons dans la fenêtre de sortie sont également visibles par MFDeploy.

Pour afficher ces messages, cliquez sur “Target->Connect” depuis le menu et appuyez sur le bouton reset de la carte. Juste après **cliquez sur “ping”**. MFDeploy se bloquera un instant et affichera une longue liste de messages.



Notez que pour FEZ Domino le bouton reset est sur la carte. Pour la FEZ Mini, vous devez connecter un bouton à la broche Reset. Si vous avez le startkit ou le kit robot, alors vous pouvez utiliser le bouton sur ces cartes.



12. Pulse Width Modulation

En français, mais peu utilisé : Modulation de la Largeur d'Impusion. PWM est un moyen de contrôler la puissance fournie à un appareil. Atténuer une LED ou ralentir un moteur se fait plus facilement en utilisant un signal PWM. Si nous appliquons une tension à une LED, elle s'allume complètement et si nous coupons l'alimentation elle s'éteint complètement. Mais que se passerait-il si nous l'allumions pendant 1ms et la coupions pendant également 1ms ? Elle clignoterait tellement vite que notre oeil ne verrait pas le clignotement mais constaterait que la lumière émise est plus faible.

Nous pouvons calculer ceci très facilement : (on/(off+on))x100= 50%. On voit donc que la LED obtient seulement la moitié de la puissance.

Et si nous l'allumons pendant 1ms et l'éteignons pendant 9ms, alors elle recevrait (1/(1+9))x100=10% de la puissance et paraitrait très faible.

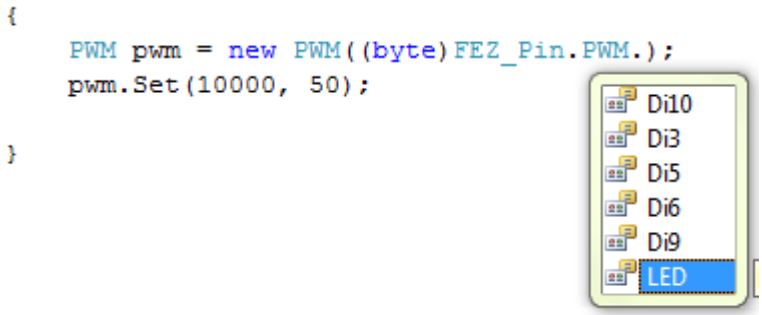
Un signal PWM est très simple à générer mais si nous devions changer l'état d'une broche plusieurs centaines ou milliers de fois par seconde, ça ferait perdre un temps fou au processeur. Il y a des cartes qui génèrent des signaux PWM plus efficacement et beaucoup de processeurs incluent des circuits capables de générer de tels signaux PWM au niveau du matériel. Cela signifie que nous pouvons configurer notre carte pour qu'elle génère automatiquement un signal PWM sans que nous ayons à nous en préoccuper ensuite.

FEZ propose plusieurs broches pouvant servir à générer un signal PWM. La librairie FEZ inclue tout ce qui est nécessaire pour activer le PWM sur ces broches.

Voici un exemple qui crée un objet PWM avec une fréquence de 10KHz (10 000 impulsions/sec) et un cycle de service (duty cycle) de 50 (50% de puissance)

```
PWM pwm = new PWM((PWM.Pin) FEZ_Pin.PWM.LED);
pwm.Set(10000, 50);
```

FEZ propose une énumération pour connaitre quelles broches peuvent générer un signal PWM. En utilisant cette énumération, Visual Studio vous donne la liste pendant que vous tapez cotre code, comme montré ci-dessous :



La liste ci-dessous montre que la diode sur la carte est connectée à une broche PWM. Et si au lieu de la faire clignoter nous la faisions faiblir/augmenter en luminosité ? Ca serait sympa ! Voici le code :

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         sbyte dirr = 1;
         byte duty = 40;
         PWM pwm = new PWM((PWM.Pin)FEZ_Pin.PWM.LED);
         while (true)
         {
            pwm.Set(10000, duty);
            duty = (byte)(duty + dirr);
            if (duty > 90 || duty < 10)
            {
               dirr *= -1;
            }

            Thread.Sleep(10);
         }

      }
   }
}
```

Ce code augmente la puissance fournie à la LED jusqu'à 90% puis la diminue jusqu'à 10% et reprend dans le sens inverse.

Important: Un signal PWM est bien adapté pour contrôler la vitesse de déplacement d'un robot ou la rotation d'un ventilateur. Une broche PWM peut aisément allumer une LED mais ce n'est pas le cas si on veut contrôler un appareil gourmand en puissance comme un moteur ou une ampoule normale. Dans ce cas, on utilisera le signal PWM pour commander un autre circuit, qui lui commandera la puissance fournie à l'appareil. Par exemple, les pont en H sont très souvent utilisés pour contrôler la direction et la vitesses de moteurs.

Note importante: Tous les signaux PWM matériels partagent la même horloge interne. Le changement de fréquence sur une broche changera la fréquence sur toutes les autres. Par contre, le cycle de service est indépendant.

Simuler un signal PWM

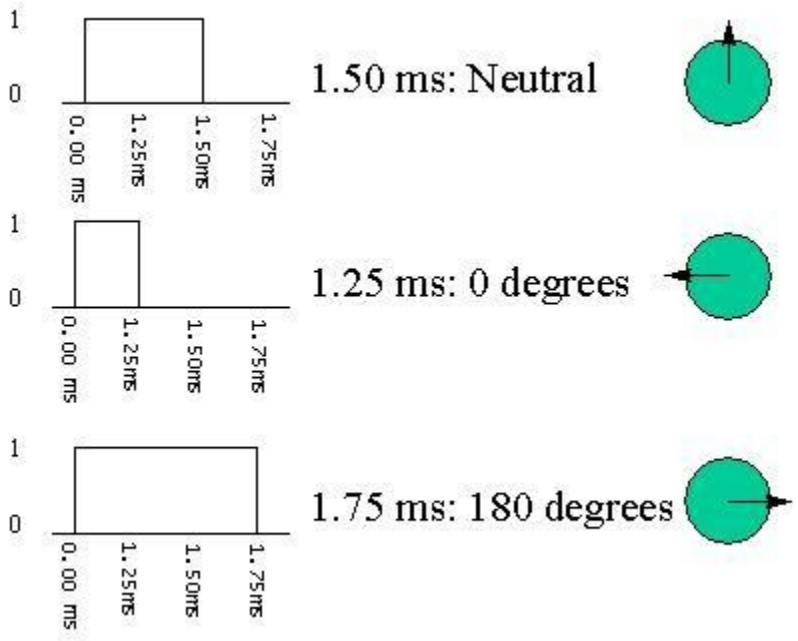
Les broches PWM sont contrôlées à l'intérieur du processeur par des circuits spécialisés. Cela signifie que ces circuits basculeront les broches, pas vous. Le processeur n'a besoin que de quelques registres afin qu'aucune interaction ne soit nécessaire pour générer le signal PWM.
Les cartes GHI's NETMF proposent la classe OutputCompare. A travers cette classe, l'utilisateur peut générer à peu près n'importe quel signal (PWM inclus) Notez que cela est fait de manière logicielle, donc il est plus judicieux d'utiliser la classe PWM si possible. Pourquoi ? Quand nous utilisons la classe PWM, le signal est généré par le matériel et le processeur n'a rien à faire. D'un autre côté, quand vous utilisez OutputCompare pour générer le PWM, le processeur doit conserver une trace des temps passés et changer l'état des broches à intervalles constants.

L'exemple servo-moteur fourni sur www.TinyCLR.com utilise OutputCompare. L'avantage ici est que vous pouvez utiliser n'importe quelle broche pour contrôler le servo. C'est mieux d'utiliser PWM mais dans ce cas vous êtes limités aux broches PWM.

Contrôler un servo et régler le PWM

Nous avons vu comment contrôler la puissance avec PWM. Il y a d'autres bonnes utilisations pour PWM, comme contrôler des servos. La class PWM fournit 2 méthodes Set et SetPulse. Avant, nous avons utilisé PWM.Set qui est très bien pour spécifier la puissance. Pour contrôler des servos, SetPulse est plus adapté.

Comment fonctionnent les servos ? Voyez ici : www.pc-control.co.uk



Vous pouvez contrôler la position d'un servo en lui envoyant une impulsion. Si l'impulsion dure 1,25ms alors le servo est à 0°. Si on augmente à 1,50ms le servo sera à 90° (position neutre). Une impulsion de 1,75ms l'enverra à 180°. Bon, il est facile de générer le signal PWM mais il nous manque encore une info : l'image nous montre la durée d'une impulsion “haute” mais qu'en est-il de l'impulsion “basse” ? Les servos attendent une impulsion toutes les 20 à 30ms. Maintenant, nous avons tout ce qu'il nous faut. Avant de commencer, je vais expliquer ce qu'est la “période” : c'est la durée totale haut+bas. Maintenant on est prêt pour coder.

La méthode PWM.SetPulse a besoin de la durée haute et de la période du signal. Comme la durée basse n'est pas critique (entre 20 et 30ms), je prendrai 20ms. Ce qui est important, c'est que la durée haute soit entre 1,25ms et 1,75ms.

Une dernière chose : SetPulse accepte des valeurs en nanosecondes mais nous avons parlé de millisecondes. 1 ms = 1 000 000 ns.

Voici enfin le code :

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;
class Program
{
   public static void Main()
   {
      PWM servo = new PWM((PWM.Pin)FEZ_Pin.PWM.Di5);
      while (true)
      {
         // 0 degrés. Période 20ms et impulsion haute 1,25ms
         servo.SetPulse(20 * 1000 * 1000, 1250 * 1000);
         Thread.Sleep(1000);//wait for a second

         // 90 degrés. Période 20ms et impulsion haute 1,50ms
         servo.SetPulse(20 * 1000 * 1000, 1500 * 1000);
         Thread.Sleep(1000);//wait for a second

         // 180 degrés. Période 20ms et impulsion haute 1,75ms
         servo.SetPulse(20 * 1000 * 1000, 1750 * 1000);
         Thread.Sleep(1000);//wait for a second
      }
      Thread.Sleep(-1);
   }
}
```

12.1. Piezo

Les haut-parleurs piézo electriques sont un moyen bon marché pour ajouter un peu de son à vos appareils. Ils émettent un son dès qu'on leur applique une fréquence donnée. Un signal PWM est parfait pour générer une telle fréquence et peut donc facilement activer un HP piézo. Nous utiliserons un duty-cycle de 50% mais nous changerons la fréquence pour avoir différentes tonalités.

Quand vous connectez un piézo, faites attention à la polarité : + va vers PWM et – vers la masse.

Regardez le composant HP FEZ Piezo sur www.TinyCLR.com

C'est un projet qui décode un fichier MIDI à partir d'une carte SD et qui joue la partie principale sur le piézo.

http://www.microframeworkprojects.com/index.php?title=PWM_MIDI_Player

13. Filtre Anti-rebonds

Quand nous avons utilisé le bouton tout à l'heure, nous l'avons mis à l'état haut de sorte qu'une fois pressé la broche devenait basse. Quand vous actionnez un interrupteur ou appuyez sur un bouton, celui-ci peut rebondir. En d'autres termes, le bouton génère quelques états on/off avant de se stabiliser. On dirait donc que le bouton a été appuyé plusieurs fois. Ces quelques rebondissements se produisent en un temps très court. Pour éliminer ceci au niveau du matériel, nous pouvons mettre un condensateur entre la broche et la masse. Pour le gérer côté logiciel, on peut mesurer le temps entre deux appuis et s'il est très court, alors c'est un rebond (“glitch”) et doit être ignoré. Un utilisateur peut appuyer sur un bouton toutes les 200ms environ. Donc si l'écart est de 10ms entre deux appuis, on sait que ce n'est pas un humain qui l'a fait et donc on peut le filtrer.

Heureusement, NETMF inclut en interne un tel filtre pour qu'on n'ait pas à s'en soucier. Quand nous créons une broche d'entrée, nous avons la possibilité d'activer le filtre anti-rebonds. Le deuxième argument de la création d'une broche d'entrée est un booléen qui indique si le filtre doit être activé (true) ou pas (false). Quand on utilise des boutons/interrupteurs, mieux vaut l'activer.

```
Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, true, Port.ResistorMode.PullUp);
```

On peut changer la sensibilité du filtre anti-rebonds de cette manière :

```
TimeSpan ts = new TimeSpan(0, 0, 0, 0, 200);//200ms
Microsoft.SPOT.Hardware.Cpu.GlitchFilterTime = ts;
```

Note important: Le filtre anti-rebonds ne fonctionne que sur des broches capables de générer des interruptions. Si vous essayez d'utiliser un port d'entrée (InputPort) avec un filtre anti-rebonds à True et que vous obtenez une exception, alors il y a de fortes chances pour que la broche choisie ne soit pas capable de générer des interruptions.

14. Entrées/Sorties analogiques

Les broches analogiques sont généralement multiplexées avec les broches numériques. Certaines broches peuvent être l'une ou l'autre mais pas les deux à la fois.

14.1. Entrées analogiques

Les entrées numériques peuvent seulement lire des états hauts ou bas (0 ou 1) mais les broches analogiques peuvent mesurer une tension. Il y a des limitations quant aux tensions applicables aux entrées analogiques. Par ex, les entrées analogiques FEZ peuvent lire une tension entre 0 et 3,3V. Quand une broche est numérique, elle tolère 5V mais si elle est configurée en analogique, alors 3,3V est le maximum. Cette limitation n'est pas réellement un problème car la plupart des signaux analogiques tiennent compte de ce comportement. Un diviseur de tension ou un circuit ampli-op peut être utilisé pour obtenir une partie du signal, le mettre à l'échelle. Par ex, si nous voulons mesurer la tension d'une batterie, c'est à dire 6V, alors nous allons diviser par deux cette tension avec des résistances par ex, pour que la broche ne voit que la moitié de la tension, soit 3V max. Dans le logiciel, comme nous savons que la tension lu est divisée par 2, il suffira de la multiplier par 2 pour avoir la tension véritablement mesurée.

Heureusement, l'implémentation de GHI pour les entrées analogiques tient compte de cette mise à l'échelle. Quand on crée un nouvel objet Entrée Analogique, vous pouvez préciser l'échelle, en option.

La référence interne est de 0V à 3,3V, donc chaque mesure que vous faites doit en tenir compte.. Le plus facile est de mettre l'échelle de 0 à 3300. On peut ainsi penser en millivolts : is on lit 1000, alors la tension à l'entrée est de 1V.

```
AnalogIn BatteryVoltage = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An0);
BatteryVoltage.SetLinearScale(0, 3300);
int voltage = BatteryVoltage.Read();
```

Pensez à utiliser l'énumération pour voir quelles broches peuvent être analogiques.

Pour afficher la tension lue en volts, il faut convertir les millivolts puis convertir en chaine.

```
AnalogIn BatteryVoltage = new AnalogIn((AnalogIn.Pin) FEZ_Pin.AnalogIn.An0);
BatteryVoltage.SetLinearScale(0, 3300);
int voltage = BatteryVoltage.Read();
Debug.Print("Voltage = " + (voltage / 1000).ToString() + "." + (voltage % 1000).ToString());
```

On divise par 1000 pour avoir la tension et modulo pour la partie après la virgule.

14.2. Sorties analogiques

Les sorties analogiques ressemblent aux sorties numériques en ce sens qu'elles ont également des limites quant à la puissance qu'elle peuvent fournir. Elle est même plus faible que pour les sorties numériques. Elles ne sont capable que de fournir un petit signal pour commander un autre circuit ou éventuellement un ampli de puissance.

Les sorties numériques peuvent être à l'état haut ou bas alors que les sorties analogiques peuvent fournir une tension entre 0 et 3,3V. L'implémentation par GHI des sorties analogiques permet une mise à l'échelle automatique : vous donnez le minimum, le maximum et la valeur de la tension.

Un simple test serait de mettre le mini à 0 le le max à 330V (3,3Vx100) et la valeur à 100 (1Vx100). Cela donnera 1V sur la broche. Nous allons connecter cette broche à une entrée analogique pour vérifier cela. Cela ne sera pas exactement 1V mais très proche.

La sortie analogique est multiplexée avec l'entrée analogique 3. Quand on utilise la sortie analogique sur cette broche, l'entrée analogique ne peut plus être utilisée, mais on peut toujours utiliser les autres entrées analogiques.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         AnalogOut VoltLevel = new AnalogOut((AnalogOut.Pin)FEZ_Pin.AnalogOut.An3);
         VoltLevel.SetLinearScale(0, 3300);
         VoltLevel.Set(1000);

         AnalogIn PinVoltage = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An0);
         PinVoltage.SetLinearScale(0, 3300);

         while (true)
         {
            int voltage = PinVoltage.Read();
            Debug.Print("Tension = " + (voltage / 1000).ToString() + "." + (voltage % 1000).ToString());

            Thread.Sleep(200);
         }
      }
   }
}
```

Connectez un fil entre An0 et AOUT(An3) et lancer ce programme. Notez que si aucun fil n'est connecté alors la broche est “flottante” et montre des valeurs aléatoires. Essayez de touche An0 et voyez comment la valeur change puis connectez-là à Aout pour voir que vous reviendrez à 1V. Cela ne sera pas tout à fait 1V mais sera très proche.

15. Le ramasse-miettes

En programmant avec des langages plus anciens comme C ou C++, les programmeurs devait conserver une trace des objets qu'ils créaient et les libérer quand nécessaire. Si un objet est créé et pas libéré, alors il utilise des ressources systèmes qui ne seront jamais disponibles. Le symptôme le plus courant est la “fuite de mémoire”. Un programme qui a une fuite de mémoire va utiliser de plus en plus de mémoire jusqu'à ce que le système en manque et plante. Ces bugs sont très difficiles à localiser dans un code.

Les langages modernes ont un “ramasse-miettes” (Garbage Collector) qui garde une trace des objets utilisés. Quand le système vient à manquer de mémoire, le ramasse-miettes s'active et cherche parmi tous les objets et libère ceux qui ne sont plus référencés. Vous vous souvenez quand nous avons créé des objets avec le mot-clé “new” puis que nous avons assigné une référence à l'objet ? Un objet peut être référencé plusieurs fois et le ramasse-miettes ne le supprimera pas tant qu'il restera une référence.

```
// Nouvel objet
OutputPort Ref1 = new OutputPort(FEZ_Pin.Digital.LED, true);
// deuxième référence pour le même objet
OutputPort Ref2 = Ref1;
// on perd la première référence
Ref1 = null;
// Notre objet est toujours référencé
// Il ne sera donc pas supprimé pour l'instant
// maintenant on enlève la seconde référence
Ref2 = null;
// à partir de maintenant, l'objet est susceptible d'être
// supprimé par le ramasse-miettes
```

Notez que l'objet n'est pas supprimé immédiatement. Au besoin le ramasse-miettes s'activera et le supprimera. Cela peut éventuellement poser un problème parfois car le ramasse-miettes a besoin d'un peu de temps pour chercher et supprimer les objets. Cela ne durera que quelques millisecondes mais comment faire si votre application ne peut pas se le permettre ? Dans ce cas, vous pouvez forcer l'exécution du ramasse-miettes quand vous le désirez.

```
// Force l'usage du ramasse-miettes
Debug.GC(true);
```


15.1. Perte de ressources

Le ramasse-miettes facilite l'allocation d'objets mais peut aussi poser des problèmes si nous ne sommes pas prudents. Voici un exemple typique avec une sortie numérique. Supposons que nous ayons besoin d'une broche à l'état haut. Nous créons un objet OutputPort et mettons la broche à haut.. Plus tard, nous perdons la référence à cette objet pour une raison X. La broche sera toujours à l'état haut. Jusque là, pas de souci. Après quelques minutes, le ramasse-miettes entre en piste et trouve cet objet non référencé, donc il le supprime. Normal. Sauf que libérer un port en sortie le change en port d'entrée ! Du coup, la broche n'est plus à l'état haut !

```
// Allume la LED
OutputPort LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
LED = null;
// On a perdu la référence mais la LED est toujours allumée
//On force le ramasse-miettes
Debug.GC(true);
// La LED est éteinte !
```

Une chose importante à noter est que si nous créons une référence à un objet à l'intérieur d'une méthode, la référence sera perdue à la sortie de la méthode. Voici un exemple :

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
   public class Program
   {
      static void TurnLEDOn()
      {
         // Allume la LED
         OutputPort LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
      }
      public static void Main()
      {
         TurnLEDOn();
         // on pourrait croire que tout va bien, ce qui n'est pas le cas
         // Force le ramasse-miettes
         Debug.GC(true);
         // La LED est-elle encore allumée ?
      }

   }
}
```

Pour pallier à ça, nous avons besoin d'une référence qui soit toujours accessible. Voici le code correct :

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
   public class Program
   {
      static OutputPort LED;
      static void TurnLEDOn()
      {
         // Allume la LED
         LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
      }
      public static void Main()
      {
         TurnLEDOn();
         // Lance le ramasse-miettes
         Debug.GC(true);
         // La LED est-elle toujours allumée ?
      }

   }
}
```

Un autre bon exemple concerne les timers. NETMF propose des timers qui permettent de gérer certaines actions à des intervalles réguliers. Si la référence au timer est perdue et que le ramasse-miettes est passé par là, alors le timer est perdu et ne fonctionnera plus comme prévu. Les timers seront expliqués un peu plus tard.

15.2. Dispose

Le ramasse-miettes libèrera les objets à certains moments, mais comment faire quand on veut libérer un objet immédiatement ? La plupart des objets ont une méthode Dispose(). Si un objet a besoin d'être libéré à n'importe quel moment, on peut le “Disposer”.

Disposer les objets est très important avec NETMF. Quand nous créons un objet InputPort, la broche est réservée. Comment faire si on veut l'utiliser ensuite comme ue sortie ? Ou même comme une entrée analogique ? Nous devrons d'abord libérer la broche et ensuite seulement créer le nouvel objet :

```
OutputPort OutPin = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.Di5, true);
OutPin.Dispose();
InputPort InPort = new InputPort((Cpu.Pin)FEZ_Pin.Digital.Di5, true, Port.ResistorMode.PullUp);
```


16. C# Niveau 3

Cette section va couvrir les infos C# que nous voulions absolument inclure dans ce livre. Un bon livre gratuit pour continuer à apprendre C# est disponible ici :

http://www.programmersheaven.com/2/CSharpBook

16.1. Byte (Octet)

Nous avons appris comment les int étaient pratiques pour stocker des nombres. Ils peuvent stocker des nombres très grands mais chaque entier consomme 4 octets de mémoire. Vous pouvez penser l'octet comme une case de mémoire. Un octet peut contenir une valeur entre 0 et 255. Ca parait peu mais c'est pourtant suffisant pour beaucoup de choses. En C#, les octets sont déclarés en utilisant le mot-clé “byte” :

```
byte b = 10;
byte bb = 1000;// Ca ne marche pas !!!
```

La valeur d'un octet est 255 au maximum, alors que se passe-t-il quand on l'incrémente au-delà ? Et bien sa valeur retourne à 0, tout simplement.

Vous allez probablement vouloir utiliser des entiers pour la plupart de vos variables, mais nous apprendrons plus tard que les octets sont très importants dès qu'on commence à utiliser les tableaux (array).

16.2. Char (Caractère)

Pour représenter un langage comme l'anglais, nous avons besoin de 26 valeurs pour les minuscules et 26 pour les majuscules, puis 10 pour les chiffres et peut-être encore 10 pour les symboles. En additionnant le tout, on arrive à moins de 255, donc un octet nous suffira. Si nous créons une tables contenant les lettres, les chiffres et les symboles, nous pouvons tout représenter par une valeur numérique. En fait, cette table existe déjà et est appelée Table ASCII.

Un octet est suffisant pour stocker tous les “caractères” que nous employons en anglais. Les ordinateurs modernes ont été étendus pour inclure d'autres langages, certains avec des caractères “non latins” très complexes. Ces nouveaux caractères sont appelés Unicode. Ceux-ci peuvent prendre des valeurs supérieures à 255 donc un octet ne suffit pas. Mais un entier (4 octets) reste trop long. Nous avons donc besoin d'un type qui utilise 2 octets de mémoire. 2 octets sont bien pour stocker des valeurs entre 0 et 65535. Ce type s'appelle

“short” mais nous ne l'utilisons pas dans ce livre.

Les systèmes peuvent donc représenter les caractères en utilisant un ou deux octets. Les programmeurs ont décidé de créer un nouveau type appelé “char” qui peut être 1 ou 2 octets, selon le système concerné. Comme NETMF est fait pour les petits systèmes, la taille d'un caractère est 1 octet. Ce qui n'est pas le cas sur un PC où on a 2 octets !

Ne vous inquiétez pas trop à ce sujet : n'utilisez pas le type char si vous n'en avez pas besoin et si vous l'utilisez, souvenez-vous qu'il occupe 1 octet sur NETMF.

16.3. Array (Tableau)

Si nous lisons 100 fois une entrée analogique et que nous voulons passer les valeurs lues à une méthode, ce n'est pas pratique d'utiliser 100 variables dans 100 arguments ! A la place, on crée un tableau du type de notre variable. Vous pouvez créer des tableaux de n'importe quel objet. Nous allons principalement utiliser des tableaux d'octets. Quand vous allez interfacer des matériels ou accéder à des fichiers, vous allez toujours utiliser des tableaux d'octets.

La déclaration est similaire à celle des objets :

```
byte[] MyArray;
```

Le code ci-dessus crée une “référence” à un objet de type “tableau d'octets”. C'est pour l'instant une simple référence et il ne contient aucun objet, il est “null”. Si vous avez oublié ce qu'est une référence alors revenez en arrière dans le chapitre “C# Niveau2”.

Pour créer l'objet, nosu utilisons le mot-clé “new” et devons préciser la taille de notre tableau. La taille est le nombre d'éléments que nous aurons dans le tableau. Ici, le type est “octet” et donc le nombre représentera la taille mémoire allouée en mémoire.

```
byte[] MyArray;
MyArray = new byte[10];
```

Nous avons créé un tableau d'octets avec 10 éléments dedans. Ce tableau est référencé par “MyArray”.

Nous pouvons maintenant lire/écrire n'importe laquelle des 10 valeurs dans le tableau simplement en indiquant l'index de la valeur désirée.

```
byte[] MyArray;
MyArray = new byte[10];
MyArray[0] = 123;// first index
MyArray[9] = 99;// last index
MyArray[10] = 1;// This is BAD...ERROR!!
```


Un point très important à noter ici est que les index débutent à 0. Donc pour un tableau de 10 éléments, les index vont de 0 à 9. L'accès à l'index 10 ne fonctionnera pas et génèrera une exception.

Nous pouvons assigner des valeurs aux éléments du tableau lors de sa déclaration. L'exemple suivant va stocker les nombre de 1 à 10 dans les index de 0 à 9 :

```
byte[] MyArray = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Pour copier un tableau, utilisez la classe Array comme suit :

```
byte[] MyArray1 = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
byte[] MyArray2 = new byte[10];
Array.Copy(MyArray1, MyArray2, 5);// Copie de 5 éléments seulement
```

Une propriété importante et très utilie d'un tableau est sa propriété Length. Nous pouvons l'utiliser pour déterminer le nombre d'éléments qu'il contient.

```
byte[] MyArray1 = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
byte[] MyArray2 = new byte[10];
Array.Copy(MyArray1, MyArray2, MyArray1.Length);//Copie tout le tableau
```

16.4. String

Nous avons déjà utilisé des chaines en différents endroits. Nous allons revoir un peu tout cela et ajouter quelques détails.

Les programmes ont généralement besoin d'écrire des messages. Ceux-ci doivent être lisibles par un homme. Parce que c'est utile et très souvent utilisé dans les programmes, C# supporte les chaines nativement. C# sait que le texte dans un programme est une chaine parce qu'il est entre guillemets.

Voici un exemple de chaines :

```
string MyString = "Une chaine";
string UneAutreChaine = "UneAutreChaine";
```

Tout ce qui est entre les guillemets est coloré en rouge et considéré comme une chaine. Notez que sur la deuxième ligne j'ai fait exprès de placer le même texte entre guillemets. C# ne compile pas ce qui se trouve entre les guillemets (texte rouge) mais le considère simplement comme une chaine.

Vous pouvez peut-être encore être troublé par la différence entre une variable entière qui contient la valeur 5 et une chaine qui contient le chiffre 5. Voici un exemple :


```
string MyString = "5" + "5";
int MyInteger = 5 + 5;
```

Quelles sont les valeurs des variables ? Pour l'entier, c'est 10 car 5+5=10. Mais pour la chaine, ce n'est pas le cas. Les chaines ne s'occupent pas de la signification de leur contenu, il n'y a pas de différences entre un chiffre et une lettre. Quand on ajoute une chaine à une autre, la deuxième est simplement “collée” à la première. Et donc “5”+”5”+ = “55” et non 10 comme pour un entier.

Quasiment tous les objets ont une méthode ToString() qui convertit les informations en texte lisible. Voici un exemple :

```
int MyInteger = 5 + 5;
string MyString = "La valeur de MyInteger est: " + MyInteger.ToString();
Debug.Print(MyString);
```

Ce code écrira :

La valeur de MyInteger est: 10

Les chaines peuvent être converties en tableaux d'octets si nécessaire. C'est important si nous devons utiliser une méthode qui n'accepte que des octets et que nous voulons lui passer notre chaine. En faisant cela, tous les caractères de notre chaine seront convertis en leurs équivalents en octets et stockés dans le tableau.

```
using System.Text;
.....
.....
byte[] buffer = Encoding.UTF8.GetBytes("Example String");
```

16.5. Boucle For

Les boucles while suffisent généralement à nos besoins mais parfois les boucles “for” sont plus facile à utiliser. L'exemple le plus simple consiste à compter de 1 à 10. Ou encore faire clignoter 10 fois une LED. Une boucle For prend 3 arguments sur une variable. Il faut la valeur initiale, comment terminer la boucle et quoi faire dans chaque boucle.

```
int i;
for (i = 0; i < 10; i++)
{
   // Fait quelque chose
}
```


Au départ, nous devons déclarer la variable à utiliser. Puis, dans la boucle For, nous devons fournir les 3 argument (initial, règle, action). Dans la toute première boucle, nous demandons de mettre la variable i à 0. Puis la boucle continuera de tourner tant que i sera inférieur à 10. Enfin, la boucle incrémentera la variable i à chaque itération. Testons cela :

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         int i;
         for (i = 0; i < 10; i++)
         {
            Debug.Print("i= " + i.ToString());
         }
      }

   }
}
```

Si nous lançons le programme, nous verrons qu'il affiche des valeurs 0 à 9 mais pas 10. Mais nous voulions de 1 à 10, pas de 0 à 9 ! Pour commencer à 1 et non 0, nous devons initialiser i à 1 dans la boucle initiale. Egalement, pour aller jusqu'à 10, nous devons dire à la boucle d'aller jusqu'à 10 et pas plus de 10, donc nous changerons la condition de “<” à “<=”

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         int i;
         for (i = 1; i <= 10; i++)
         {
            Debug.Print("i= " + i.ToString());
         }
      }

   }
}
```


Peut-on faire une boucle qui ne compte que les nombres pairs (incrémenter de 2, en fait) ?

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         int i;
         for (i = 2; i <= 10; i = 1 + 2)
         {
            Debug.Print("i= " + i.ToString());
         }
      }
   }
}
```

La meilleure façon de comprendre les boucles For est d'aller pas-à-pas dans le code et regarder comment C# les exécute.

16.6. Mot-clé Switch

Pour débuter, vous n'allez probablement pas utiliser “switch” mais ensuite vous verrez que c'est très utile pour de gros programmes, particulièrement quand il faut tester différents états de matériels. Le mot-clé switch compare une variable à une liste de constantes (uniquement) et exécute une action selon le cas. Dans cet exemple, nous lisons le jour courant (“DayOfWeek”) et selon sa valeur nous écrirons le jour de la semaine sous forme de chaine. Nous pourrions le faire avec un if mais voyons comment simplifier la chose avec switch :

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         DateTime currentTime = DateTime.Now;
         int day = (int)currentTime.DayOfWeek;
         switch (day)
         {
```

```
            case 0:
               Debug.Print("Dimanche");
               break;
            case 1:
               Debug.Print("Lundi");
               break;
            case 2:
               Debug.Print("Mardi");
               break;
            case 3:
               Debug.Print("Mercredi");
               break;
            case 4:
               Debug.Print("Jeudi");
               break;
            case 5:
               Debug.Print("Vendredi");
               break;
            case 6:
               Debug.Print("Samedi");
               break;
            default:
               Debug.Print("Vous ne devriez jamais voir ce message");
               break;
         }
      }
   }
}
```

Le point important est que la variable est comparée à des constantes. Après chaque cas (“case”) nous devons avoir une constante et pas une variable.

Nous pouvons modifier ce code pour utiliser l'énumération des jours de la semaine :

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         DateTime currentTime = DateTime.Now;
         switch (currentTime.DayOfWeek)
         {
            case DayOfWeek.Sunday:
               Debug.Print("Dimanche");
               break;
```

```
            case DayOfWeek.Monday:
               Debug.Print("Lundi");
               break;
            case DayOfWeek.Tuesday:
               Debug.Print("Mardi");
               break;
            case DayOfWeek.Wednesday:
               Debug.Print("Mercredi");
               break;
            case DayOfWeek.Thursday:
               Debug.Print("Jeudi");
               break;
            case DayOfWeek.Friday:
               Debug.Print("Vendredi");
               break;
            case DayOfWeek.Saturday:
               Debug.Print("Samedi");
               break;
            default:
               Debug.Print("Il n'y a pas d'autre jour !");
               break;
         }
      }
   }
}
```

Allez pas-à-pas dans ce code et voyez le fonctionnement dans le détail.

17. Interfaces série

Il y a beaucoup d'interfaces série disponibles pour le transfert de données entre processeurs. Chacune a ses avantages et inconvénients. Je vais essayer de les détailler suffisamment pour que vous puissiez les utiliser avec NETMF.

Bien qu'il y ait plusieurs interfaces série, quand nous parlons de série nous parlons de UART ou RS232. D'autres interfaces, come CAN ou SPI, transmettent les données en série mais ne sont pas des ports série !!

Pour plus de détails, consultez le web et plus particulièrement http://www.wikipedia.org/ .

17.1. UART

UART est l'une des plus anciennes et plus répandues des interfaces. Les données sont expédiées sur une broche TXD avec une séquence et une vitesse pré-définies. Quand l'émetteur envoie des 0 et 1, le récepteur vérifie les données entrantes sur la broche RXD à la même vitesse que l'émetteur envoie. Les données sont envoyés un octet à la fois. Ceci représente une seule direction pour les données. Pour en transférer dans l'autre sens, un circuit similaire existe sur le périphérique distant. Emission et réception sont deux circuits complètement séparés et peuvent fonctionner ensemble ou séparément. Chaque partie peut envoyer ou recevoir des données à n'importe quel moment.

La vitesse pour envoyer/recevoir des données s'appelle baud rate. Baud rate représente combien d'octets seront envoyés en une seconde. Généralement, une des vitesses standard est utilisée, comme 9600, 119200, 115200, par exemple.

A travers UART, deux processeurs peuvent s'interconnecter en branchant la broche TXD d'un côté sur la broche RXD de l'autre et vice-versa. Comme le signal est numérique, les tensions sur les broches UART TXD/RXD iront de 0V (bas) à 3,3V ou 5V (haut).

Dans l'industrie ou quand de longs câbles sont utilisés, 3,3V ou même 5V ne suffisent pas. Des standards existent pour définir comment transformer ce signal avec une tension supérieure pour autoriser des communications plus fiables. Le plus répandu est RS232. Quasiment tous les ordinateurs ont un port RS232. Avec RS232, les données sont les mêmes que pour UART mais les tensions sont converties des niveaux TTL (0 à 3,3V) vers les niveaux RS232 (--12V à +12V). Un point très important est que les tensions sont inversées par rapport à ce que nous pourrions penser. Quand le signal est logiquement “bas”, la tension est +12V et quand il est “haut” la tension est -12V. Il y a beaucoup de petits circuits qui convertissent les niveaux TTL en niveaux RS232, comme MAX232 ou MAX3232.

Quand nous avons besoin d'interfacer un processeur qui utilise UART avec un PC qui utilise RS232, nous devons donc utiliser un convertisseur de signaux. Voici un exemple de tels convertisseurs :

http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html

Dans le monde PC, les port USB sont plus courants que les ports série. Les ordinateurs récents, et plus particulièrement les portables, n'ont plus de port série mais ont plusieurs ports USB. Pour cette raison, les fabricants on créé des convertisseurs USB-RS232. Un produit intéressant de chez FTDI est un câble USB avec une interface UART. Notez qu'il s'agit d'UART TTL, pas RS232, ce qui signifie qu'on peut le connecter directement au port UART du processeur. Ce câble porte la référence “TTL-232R-3V3”.

Pour résumer tout cela, vous pouvez interconnecter deux processeurs en branchant les broches UART directement. Pour interfacer un processeur à un PC, vous devrez utiliser un convertisseur UART/RS232 ou RS232/USB en utilisant des circuits comme MAX232 pour le RS232 et FT232 pour l'USB.

NETMF supporte les ports séries UART de la même manière que le .Net Framewrok complet sur PC. Pour utiliser un port série, ajoutez l'assembly “Microsoft.SPOT.Hardware.SerialPort” et “using System.IO.Ports” au début de votre code.

Les ports série sur PC et NETMF sont appelés COM et commencent à COM1. Il n'y a pas de COM0 sur les ordinateurs. Cela peut être perturbant quand on veut faire correspondre un port COM à un port UART sur le processeur car sur ce dernier, cela commence en général à UART0 et non UART1... Donc : COM1 = UART0 et COM2 = UART1...etc.

Les PC disposent souvent d'un programme “Terminal” qui ouvre les ports série pour envoyer/recevoir des données et les afficher à l'écran (ou les envoyer). Un exemple d'un tel programme est : teraterm.

Le programme qui suit envoie la valeur d'un compteur 10 fois par seconde. Les données sont envoyées à 115200 baud, donc accurez vous que le terminal est configuré à cette vitesse. Le programme envoie les données sur le port COM1 du périphérique NETMF. Ce port COM n'a rien à voir avec de votre PC. Par ex, vous pourriez très bien utiliser un convertisseur USB/Série qui aura créé un port COM8. Donc vous devriez ouvrir le port COM8 sur votre PC et non COM1.

```
using System.Threading;
using System.IO.Ports;
using System.Text;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         SerialPort UART = new SerialPort("COM1", 115200);
         int counter=0;
         UART.Open();
         while (true)
         {
```

```
            // Crée une chaine
            string counter_string = "Compte : " + counter.ToString() + "\r\n";
            // Convertit la chaine en octets
            byte[] buffer = Encoding.UTF8.GetBytes(counter_string);
            // Envoie les octets sur le port série
            UART.Write(buffer, 0, buffer.Length);
            // Incrémente le compteur
            counter++;
            //Attends 1/10 de secondes (1000ms/100)
            Thread.Sleep(100);
        }
    }
  }
}
```

Notez que nous avons terminé notre chaine avec “\r\n”. “\r” dit au terminal de faire un retour en début de ligne et “\n” signifie “ajouter une nouvelle ligne”.

Quand les données sont reçues sur l'UART, elles sont mises en mémoire pour ne pas en perdre tant qu'elles ne sont pas lues.. Notez qu'il y a des limites à la quantité de données qui peuvent être mise en mémoire, donc si vous déboguez ou si vous ne lisez pas les données reçues alors le système ralentira sensiblement et l'exécution risque d'être aléatoire. Idéalement, les “évènements” seront utilisés pour gérer automatiquement la réception de données. Nous verrons les évènements un peu plus tard.

Cet exemple va attendre de recevoir des octets et affichera ce que vous avez saisi (transmis, en fait)

```
using System.Threading;
using System.IO.Ports;
using System.Text;

namespace MFConsoleApplication1
{
  public class Program
  {
    public static void Main()
    {
      SerialPort UART = new SerialPort("COM1", 115200);
      int read_count = 0;
      byte[] rx_byte = new byte[1];

      UART.Open();
      while (true)
      {

        // Lit un octet
        read_count = UART.Read(rx_byte, 0, 1);
        if (read_count > 0)// Y a-t-il vraiment une donnée ?
        {
          // Crée une chaine
```

```
                string counter_string = "Vous avez écrit : " + rx_byte[0].ToString() + "\r\n";
                // Convertit la chaine en octets
                byte[] buffer = Encoding.UTF8.GetBytes(counter_string);
                // Envoie les octets sur le port série
                UART.Write(buffer, 0, buffer.Length);
                // Attends un peu
                Thread.Sleep(10);
            }
        }
    }
}
```

Le dernier exemple est une boucle locale. Connectez un fil de la broche TX à la broche RX de votre carte et le programme enverra et testera une chaine pour vérifier la connexion.

```
using System.Threading;
using System.IO.Ports;
using System.Text;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         SerialPort UART = new SerialPort("COM1", 115200);
         int read_count = 0;
         byte[] tx_data;
         byte[] rx_data = new byte[10];
         tx_data = Encoding.UTF8.GetBytes("FEZ");
         UART.Open();
         while (true)
         {
            // Assure que tout est transmis
            UART.Flush();
            // Envoie des données
            UART.Write(tx_data, 0, tx_data.Length);
            // Délai pour être certain que tout est transmis
            Thread.Sleep(100);
            // Lit les données
            read_count = UART.Read(rx_data, 0, rx_data.Length);
            if (read_count != 3)
            {
               // Nous avons envoyé 3 caractères donc nous devons en recevoir 3
               Debug.Print("Nbre reçu incorrect : " + read_count.ToString());
            }
            else
            {
```

```
                // On a bien reçu le nombre attendu donc on vérifie le contenu
                // Je le fais de façon à ce que le code soit lisible
                if (tx_data[0] == rx_data[0])
                {
                    if (tx_data[1] == rx_data[1])
                    {
                        if (tx_data[1] == rx_data[1])
                        {
                            Debug.Print("Perfect data!");
                        }
                    }
                }
            }
            Thread.Sleep(100);
        }
    }
  }
}
```

17.2. SPI

SPI utilise 3 ou 4 fils pour transmettre des données. En UART, les deux côtés doit utiliser une vitesse prédéfinie. Pour SPI, c'est différent car un des noeuds envoie un signal d'horloge aux autre en même temps que les données. Ce signal d'horloge sert au récepteur pour qu'il sache à quelle vitesse il doit lire les données. Si vous avez quelques notions d'électronique, il s'agit d'un registre à décalage. Le signal d'horloge est toujours transmis depuis le périphérique maitre. L'autre périphérique est un esclave qui n'envoie aucun singal d'horloge mais reçoie celui du maitre.

Donc, le maitre transmet son signal d'horloge sur la broche SCK(“serial clock”) et transmet simultanément les données sur la broche MOSI (Master Out Slave In). L'esclave lira le signal d'horloge sur sa broche SCK et en même temps lira les données sur sa broche MOSI. Jusque là, c'est une communication dans un seul sens. Pendant que des données sont transmises dans une direction en utilisant MOSI, un autre paquet de de données est transmis sur la broche MISO (Master In Slave Out). Tout cela est simultané : horloge, envoi et réception. Avec SPI, il n'est pas possible de seulement envoyer ou recevoir. Vous recevrez toujours un octet pour un octet envoyé. Différentes tailles de données sont possibles, mais la plus courante reste l'octet. NETMF support des transferts en 8 octets (byte) et 16 octets (short).

Du fait de ce schéma maitre/esclave, on peut ajouter plusieurs esclaves sur le même bus et le maitre choisit avec quel esclave il échange des données. J'utilise le terme échanger car ce n'est pas envoyer ou recevoir mais envoyer et recevoir (échanger) des données. Le maitre sélectionne l'escalve en utilisant sa broche SSEL (Slave Select) ou CS (Chip Select). En théorie, un maitre peut avoir un nombre illimité d'esclaves mais il ne peut en sélectionner qu'un à la fois. Le maitre n'a besoin que de 3 connexions (SCK, MISO, MOSI) pour se

connecter à tous les escalves, mais a besoin d'une broche SSEL dédiée à chaque esclave.

Certains matériels SPI (esclaves) peuvent avoir plus d'une broche CS, comme le décodeur MP3 VS1053 qui utilise une broche pour les données et une autre pour les commandes, mais les deux partagent les mêmes broches de transfert (SCK, MOSI, MISO).

SPI a besoin de plus de câbles que d'autres bus similaires mais permet des transferts très rapides. Une horloge à 50MHz est possible sur un bus SPI, c'est à dire 50 millions de bits par seconde.

Les périphériques NETMF sont toujours des SPI maitres. Avant de créer un objet SPI, nous avons besoin de créer un objet “Configuration SPI”. L'objet configuration est utilisé pour paramétrer l'état des broches et certains paramètres de temps. Dans la plupart des cas, vous aurez l'horloge à l'état bas avec une impulsion sur l'état haut et 0 pour “select setup” et “hold time”. La seule chose que vous devriez avoir à changer est la fréquence d'horloge. Certains périphériques peuvent acepter des fréquences élevées mais pas tous. En la mettant à 1000KHz (1MHz), vous devriez être bon avec la plupart des périphériques.

L'exemple qui suit envoie/reçoit (échange, plutôt) 10 octets sur le canal 1 du bus SPI. Notez que NETMF démarre les canaux SPI (modules) à 1 mais que sur le processeur ils démarrent à 0. SPI1 dans le code est donc SPI0 sur le processeur.

```
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         SPI.Configuration MyConfig =  new
SPI.Configuration((Cpu.Pin)FEZmini.Pin.Di2x,false,0,0,false,true,1000,SPI.SPI_module.SPI1);
         SPI MySPI = new SPI(MyConfig);

         byte[] tx_data= new byte[10];
         byte[] rx_data = new byte[10];

         MySPI.WriteRead(tx_data, rx_data);

         Thread.Sleep(100);
      }
   }
}
```

17.3. I2C

I2C a été développé par Philips pour permettre à de nombreux chipsets de communiquer sur un bus à 2 fils dans les matériels domestiques (TV, en général). Comme SPI, I2C a un maitre et plusieurs esclaves sur le même bus. Plutôt que de choisir l'esclave via une broche, I2C utilise l'adressage logiciel. Avant de transférer des données, le maitre envoie l'adresse sur 7 bits de l'esclave avec lequel il souhaite communiquer. Il envoie également un bit indiquant si le maitre veut envoyer ou recevoir des données. L'esclave qui repère son adresse sur le bus confirmera sa présence et le maitre pourra alors envoyer/recevoir des données.

Le maitre va débuter sa transaction avec “start “, avant d'envoyer quoi que ce soit d'autre, et terminera avec la condition “stop”.

Les pilotes I2C NETMF sont basés sur les transactions. Si nous voulons lire la valeur d'un registre sur un capteur, nous aurons besoin d'envoyer le numéro du registre, puis nous lirons le registre. Ce sont donc deux transactions : écrire puis lire.

Cet exemple montre une communication avec un périphérique I2C à l'adresse 0x38. Il écrira 2 (numéro du registre) et lira en retour la valeur du registre.

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
   public class Program
   {
      public static void Main()
      {
         //create I2C object
         I2CDevice.Configuration con = new I2CDevice.Configuration(0x38, 400);
         I2CDevice MyI2C = new I2CDevice(con);

         //create transactions (we need 2 in this example)
         I2CDevice.I2CTransaction[] xActions = new I2CDevice.I2CTransaction[2];

         // create write buffer (we need one byte)
         byte[] RegisterNum = new byte[1] { 2 };
         xActions[0] = MyI2C.CreateWriteTransaction(RegisterNum);
         // create read buffer to read the register
         byte[] RegisterValue = new byte[1];
         xActions[1] = MyI2C.CreateReadTransaction(RegisterValue);

         // Now we access the I2C bus and timeout in one second if no responce
         MyI2C.Execute(xActions, 1000);

         Debug.Print("Register value: " + RegisterValue[0].ToString());
      }
   }
}
```

17.4. One Wire

Une exclusivité de GHI est le support des périphériques 1-wire sur NETMF. Les semiconducteurs Dallas, comme les capteurs de température ou les EEPROMS, utilisent un seul fil pour le transfert de données. Plusieurs périphériques peuvent être connectés et contrôlés sur un seul fil. La classe “one wire” propose plusieurs méthodes pour lire et écrire des données d'un périphérique 1-wire. Elle inclue également une méthode de calcul du CRC.

Cet exemple va lire la température fournie par une sonde digitale 1-wire DS18B20. Notez qu'il s'agit d'une exclusivité GHI et donc vous devrez ajouter l'assembly GHI pour le faire fonctionner.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace Test
{
   public class Program
   {
      public static void Main()
      {
         // Changez selon la broche choisie
         Cpu.Pin myPin = (Cpu.Pin)FEZ_Pin.Digital.Di4;

         OneWire ow = new OneWire(myPin);
         ushort temperature;

         // Lecture toutes les secondes
         while (true)
         {
            if (ow.Reset())
            {
               ow.WriteByte(0xCC);     // Ignore la ROM
               ow.WriteByte(0x44);     // Démarre la conversion de température

               while (ow.ReadByte() == 0) ;   // Attente pendant que la sonde est occupée

               ow.Reset();
               ow.WriteByte(0xCC);     // Ignore la ROM
               ow.WriteByte(0xBE);     // Lecture de la donnée

               temperature = ow.ReadByte();       // LSB
               temperature |= (ushort)(ow.ReadByte() << 8); // MSB

               Debug.Print("Temperature: " + temperature / 16);
               Thread.Sleep(1000);
            }
```

```
            else
            {
               Debug.Print("La sonde n'a pas été détectée.");
            }
            Thread.Sleep(1000);
        }
    }
  }
}
```

17.5. CAN

Controller Area Network est une interface très courante dans l'industrie et l'automatisme. CAN est très robuste et fonctionne très bien dans des environments perturbés à de hautes vitesses. Tous les traitements d'erreurs et de réparatsions sont fait au niveau du matériel. TD (Transmit Data) et RD (Receive Date) sont les deux seules broches nécessaires. Ces broches véhiculent le signal numérique destiné à être transformé en analogique avant d'être émis sur la ligne en utilisant la couche physique. Les couches physiques sont parfois appelées “émetteur-récepteur”

Il y a plusieurs sortes de couches physiques, mais la plus utilisée est HSDW (High Speed Dual-Wire) qui utilise une paire torsadée pour son immunité au “bruit”. Cette couche peut communique jusqu'à 1MBps et transférer des données sur de très longues distances si la vitesse utilisée est lente. Les données peuvent être transférées entre les noeuds sur le bus où tous les noeuds peuvent transférer à n'importe quel instant et tous les autres noeuds doivent être en mesure de recevoir ces données. Avec CAN, il n'y a pas de maitre/esclave. Egalement, tous les noeuds doivent avoir un critère de synchronisation prédéfini. C'est nettement plus compliqué que le simple baud-rate de UART. Pour cette raison, il existe de nombreux calculateurs (gratuits) de bit-rate pour CAN.

Le périphérique CAN utilisé sur la carte Embedded Master est identique au populaire SJA1000. Une petite recherche sur internet pour SJA1000 devrait vous donner plusieurs calculateurs.

Voici un exemple de calculateur dédié :

http://www.esacademy.com/en/library/calculators/sja1000-timing-calculator.html

GHI Electronics est en train de modifier le pilote CAN pour NETMF 4.0, donc l'interface est susceptible de changer. Vérifiez la documentation pour plus d'aide.

Voici le code avec des commentaires détaillés :

```
using System.Threading;
using Microsoft.SPOT;
```

```
using System;
using GHIElectronics.Hardware;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         // Ces nombres proviennt du calculateur mentionné ci-dessus
         ////////////////////////////////////////////////////////////////////////////////
         // Bitrate 250Kbps
         // CLK = 72 Mhz, with BRP = 12 -> 6Mhz CAN clock
         // 6Mhz/250Kbps = 24 TQ
         // T1 = 16 minus 1 for sync = 15
         // T2 = 8
         // 15 + 1 + 8 = 24 TQs which is what we need
         ////////////////////////////////////////////////////////////////////////////////
         const int BRP = 12;
         const int T1 = 15;
         const int T2 = 8;
         // For 500Kbps you can use BRP=6 and for 1Mbps you can use BRP=3 and for 125Kbps use
BRP=24...and so on
         // Keep T1 and T2 the same to keep the sampling pint the same (between 70% and 80%)

         // Initialize CAN channel, set bit rate
         CAN canChannel = new CAN(CAN.CANChannel.Channel_1,
                                  ((T2 - 1) << 20) | ((T1 - 1) << 16) | ((BRP - 1) << 0));
         // make new CAN message
         CAN.CANMessage message = new CAN.CANMessage();
         // make a message of 8 bytes
         for (int i = 0; i < 8; i++)
            message.data[i] = (byte)i;
         message.DLC = 8; // 8 bytes
         message.ArbID = 0xAB; // ID
         message.isEID = false; // not extended ID
         message.isRTR = false; // not remote
         // send the message
         canChannel.PostMessage(message);
         // wait for a message and get it.
         while (canChannel.GetRxQueueCount() == 0) ;
         // get the message using the same message object
         canChannel.GetMessage(message);
         // Now "message" contains the data, ID, flags of the received message.
      }
   }
}
```

18. Output Compare

Cette fonctionnalité exclusive de GHI permet au développeur de générer n'importe quel signal sur n'importe quelle broche. Par exemple, OutputCompare peut être utilisée pour générer un signal UART ou un signal pour contrôler une télécommande infrarouge 38KHz pour simuler une télécommande de TV.

Un très bon exemple est le pilote pour l'afficheur LCD 2x16 série présent sur le site www.TinyCLR.com. L'afficheur est contrôlé par l'UART. Il ne renvoie aucune donnée. Tout ce dont vous avez besoin est d'envoyer en série quelques codes de contrôle. Du coup, on peut connecter l'afficheur sur une des broches série. Mais alors nous allons perdre le port au complet pour une chose toute simple, sans compter que UART a besoin de 2 broches (envoi/réception). Mais comme l'afficheur n'envoie pas de données, on ne perdra que la broche émission. La bonne façon de contrôler cet afficheur série est donc d'utiliser OutputCompare : vous n'aurez besoin que d'une broche et surtout n'importe laquelle !

Comment fonctionne OutputCompare ? A la base, vous fournissez à une méthode un tableau de temps entre chaque basculement de d'état de la broche. OutputCompare va parcourir ce tableau et générer le signal sur la broche choisie. Donc si nous voulons générer de l'UART, nous devons d'abord pré-calculer les valeurs représentant la donnée à transmettre et la donner comme paramètre à l'objet OutputCompare.

Ce pilote exemple vous explique comment faire. C'est une copie du pilote présent sur le site www.TinyCLR.com pour l'afficheur LCD série

```
using System;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
namespace GHIElectronics.NETMF.FEZ
{
   public static partial class FEZ_Components
   {
      public class SerialLCD : IDisposable
      {
         const byte DISP_ON = 0xC;    // Allume le LCD
         const byte CLR_DISP = 0x01;      //Efface l'écran
         const byte CUR_HOME = 2;     //Positionne le curseur au départ et efface l'écran
         const byte SET_CURSOR = 0x80;   //SET_CURSOR + X : Position curseur en X
         const byte Move_CURSOR_LEFT = 0x10;

         OutputCompare oc;
         const int MAX_TIMINGS_BUFFER_SIZE = 10;
         uint[] buffer = new uint[MAX_TIMINGS_BUFFER_SIZE];
         const int BAUD_RATE = 2400;
```

```
const int BIT_TIME_US = 1 * 1000 * 1000 / BAUD_RATE;
readonly int BYTE_TIME_MS;
public void Dispose()
{
   oc.Dispose();
   buffer = null;
}
private void SendByte(byte b)
{
   bool currentPinState;
   int currentBufferIndex = 0;
   uint currentStateTiming;
   // Bit de démarrage
   currentPinState = false;
   currentStateTiming = BIT_TIME_US;
   // Bits de données
   for (int i = 0; i < 8; i++)
   {
      bool neededState = (b & (1 << i)) != 0;

      if (neededState != currentPinState)
      {
         buffer[currentBufferIndex] = currentStateTiming;
         currentStateTiming = BIT_TIME_US;
         currentPinState = neededState;
         currentBufferIndex++;
      }
      else
      {
         currentStateTiming += BIT_TIME_US;
      }
   }
   // Bit d'arrêt
   if (currentPinState != true)
   {
      buffer[currentBufferIndex] = currentStateTiming;
      currentBufferIndex++;
   }
   oc.Set(false, buffer, 0, currentBufferIndex, false);\
   // attends que les données soient transmises
   Thread.Sleep(BYTE_TIME_MS);
}
public void PutC(char c)
{
   SendByte((byte)c);
}

private void SendCommand(byte cmd)
{
   SendByte(0xFE);
   SendByte(cmd);
}
```

```
        public void Print(string str)
        {
            for (int i = 0; i < str.Length; i++)
                PutC(str[i]);
        }
        public void ClearScreen()
        {
            SendCommand(CLR_DISP);
        }
        public void CursorHome()
        {
            SendCommand(CUR_HOME);
        }
        public void SetCursor(byte row, byte col)
        {
            SendCommand((byte)(SET_CURSOR | row << 6 | col));
        }
        public void MoveLeft()
        {
            SendCommand(Move_CURSOR_LEFT);
        }
        public SerialLCD(FEZ_Pin.Digital pin)
        {
            BYTE_TIME_MS = (int)Math.Ceiling((double)BIT_TIME_US *
MAX_TIMINGS_BUFFER_SIZE / 1000);
            oc = new OutputCompare((Cpu.Pin)pin, true, MAX_TIMINGS_BUFFER_SIZE);
            // Initialise le LCD

            SendCommand(DISP_ON);
            SendCommand(CLR_DISP);
        }
    }
  }
}
```

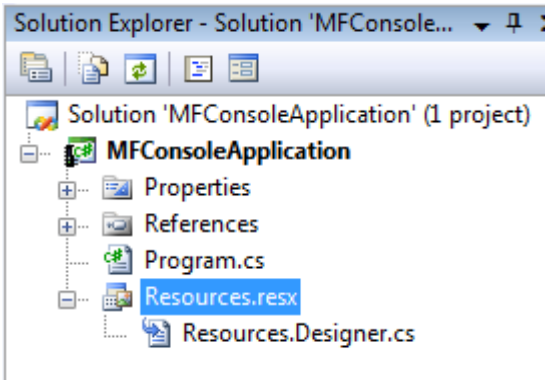
19. Charger des ressources

Une ressource est un fichier qui est inclus avec l'image de l'application. Si vos application dépend d'un fichier particulier (image, icône, son) alors on ajoute ce fichier aux ressources de l'application. Une application pourrait lire ce fichier depuis le système de fichier, mais elle deviendrait alors dépendante du système de fichiers utilisé.

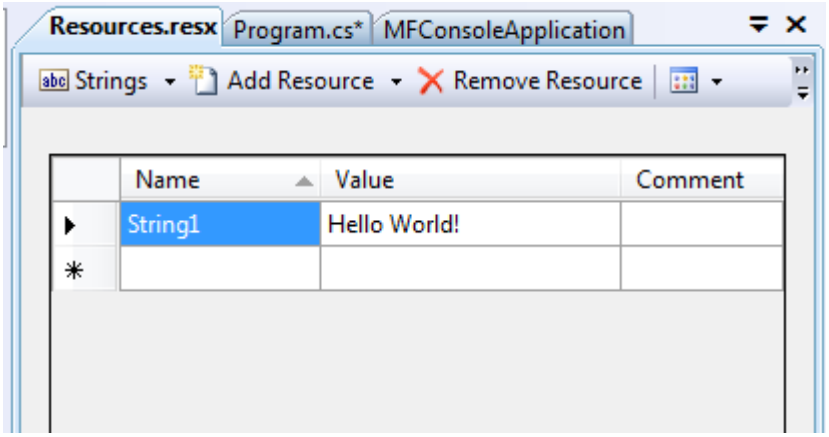
Une ressource peut également être un texte, comme une mention de copyright. Ainsi, si la mention de copyright doit être modifiée, il n'y aurait que le fichier de ressources à modifier et rien dans le code complet du programme.

Attention : l'ajout de gros fichiers résultera en une erreur de déploiement et VS2008 ne vous dira pas que c'est parce que le fichier est trop volumineux.

En regardant dans la fenêtre “Explorateur de solution”, on peut voir “Resource.resx”, qui est étendu en “Resources.Designer.cs”

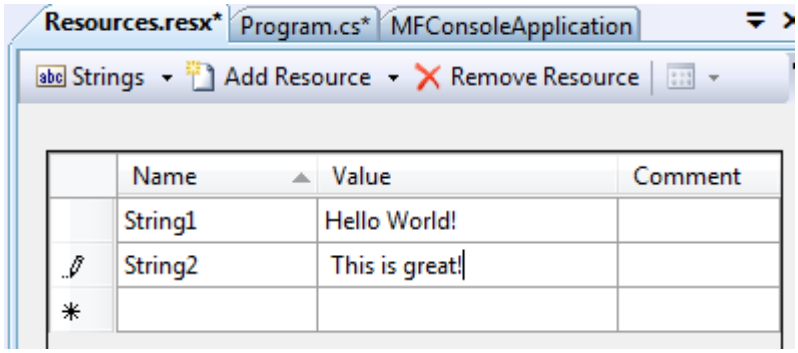


Visual Studio crée automatiquement un fichier “resources designer”. Ne modifiez jamais vous-même ce fichier. A la place, double-cliquez sur “Resources.resx” pour ouvrir une boite d'outil dédiée.



Dans cette fenêtre, la première liste déroulante à gauche représente le type de ressource que nous voulons modifier/ajouter.

Cliquez juste en-dessous de String1 et ajoutez une nouvelle ressource chaine comme ci-dessous.

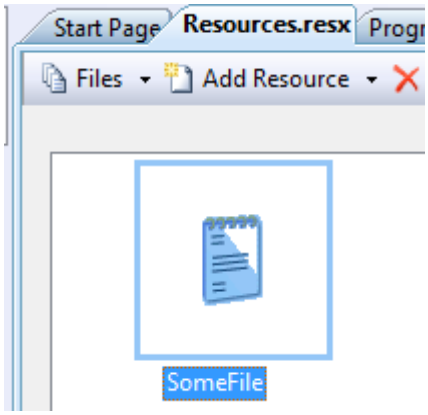


Vous avez maintenant deux ressources chaines. Utilisons-les :

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Debug.Print(Resources.GetString(Resources.StringResources.String1));
         Debug.Print(Resources.GetString(Resources.StringResources.String2));
      }
   }
}
```

Essayez de modifier le texte de la ressource et observez les changements.

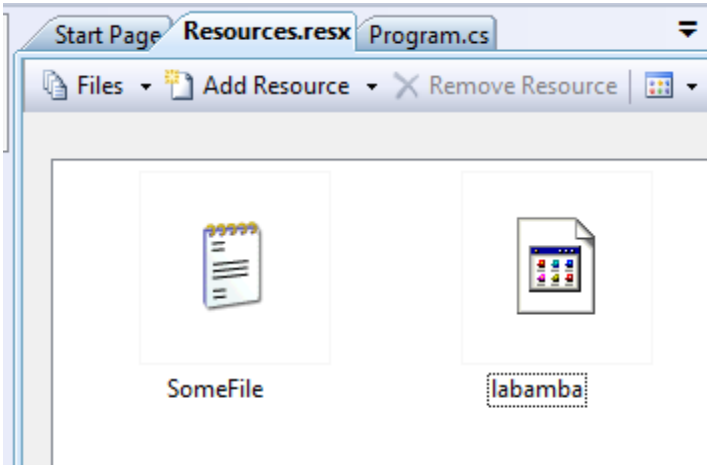
Ajoutons maintenant un fichier. Créez un fichier texte sur le bureau et écrivez quelques mots dedans. Choisissez “Fichier” depuis la liste déroulante puis cliquez sur “Ajouter ressource”



Comme c'est un fichier texte, Visual Studio l'a ajouté de la même manière qu'une chaine de caractères. Nous pouvons donc y accéder de la même façon :

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Debug.Print(Resources.GetString(Resources.StringResources.SomeFile));
      }
   }
}
```

Dans cet exemple, j'ajoute un fichier midi. Plus tard dans ce livre nous verrons comment ajouter un décodeur et jouer ce fichier. Choisissez un fichier midi quelconque et ajoutez-le.



Dans cet exemple, la taille du fichier est à peu près 33Ko. Ce qui est peu pour un PC mais gros pour un système embarqué. L'exemple ci-dessous fonctionnera sur les cartes disposant de beaucoup de RAM, comme ChipworkX et Embedded Master. Sur USBizi et FEZ, ça peut ne pas fonctionner. Nous pouvons bien sûr lire de gros fichiers avec USBizi ou FEZ, mais plutôt que le lire en une seule fois (en entier), nous le lirons par morceaux, décoderons ce morceaux et reprendrons la lecture d'un autre morceau. Nous verrons cela en détail plus tard.


```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         byte [] file_data = Resources.GetBytes(Resources.BinaryResources.labamba);
         Debug.Print("La taille du fichier est " + file_data.Length.ToString());
      }
   }
}
```


20. Afficheurs

20.1. Afficheurs caractères (LCD)

La plupart des afficheurs LCD utilisent la même interface. En général, il y 2 lignes de 16 caractères, communément appelés 2x16 LCD. L'affichage est contrôlé avec une interface 8 bits ou 4 bits. L'option 4 bits est mieux car elle requiert moins de broches.

L'interface utilise RS (Data/Instruction), RW(Read/Write), E (Enable) et 4 bits de données. La doc de l'afficheur est la meilleure source d'infos, mais voici un petit exemple pour démarrer rapidement.

GHI offre une alternative à cet afficheur. Le SerialLCD proposé sur www.TinyCLR.com fonctionne sur une seule broche de la carte FEZ. Le pilote inclus facilite encore plus son utilisation.

Voici :

```
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         FEZ_Components.SerialLCD LCD =
            new FEZ_Components.SerialLCD(FEZ_Pin.Digital.Di5);

         LCD.ClearScreen();
         LCD.CursorHome();
         LCD.Print("FEZ ça pousse !");
      }
   }
}
```

20.2. Afficheurs graphiques

Support natif

NETMF, avec sa classe bitmap, gère très bien les graphiques. Cette classe peut utiliser des images de type BMP, JPG et GIF. Les images peuvent être obtenues à partir du système de fichiers mais il est plus facile de les inclure dans une ressource.

L'objet bitmap peut être utilisé pour dessiner des images, des formes ou du texte (en utilisant une police de caractères). NETMF supporte la création de polices avec l'outil TFConvert.

Quand nous dessinons en utilisant l'objet bitmap, nous dessinons en fait dans l'objet (en mémoire) et rien n'est visible à l'écran. Pour transférer l'image de la mémoire à l'écran, nous devons utiliser la méthode flush(). Un point important ici est que flush() ne fonctionne que si la taille du bitmap est identique à celle de l'écran.

Si nous avons une image de 128x128 et voulons l'afficher sur l'écran, nous devons d'abord créer un bitmap de la taille de l'écran puis un deuxième de la taille de l'image. Dessinez le bitmap de l'image sur celui de l'écran et utilisez flush() sur celui de l'écran.Pour éviter les confusions, j'ai toujours un objet bitmap appelé LCD et tout est dessiné sur ce bitmap.

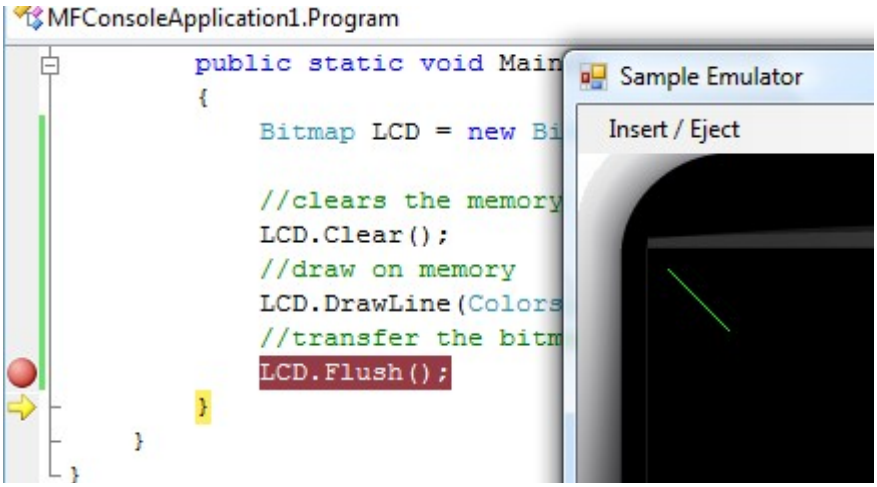
Nous allons faire nos tests sur l'émulateur car votre matériel ne supporte peut-être pas les graphiques nativement.

```
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth, SystemMetrics.ScreenHeight);

         //Efface la mémoire (pas l'écran)
         LCD.Clear();
         //Dessine en mémoire
         LCD.DrawLine(Colors.Green, 1, 10, 10, 40, 40);
         //Transfère le dessin en mémoire vers l'écran
         LCD.Flush();
      }
   }
}
```

Il faut ajouter l'assembly Microsoft.SPOT.TinyCore au programme et faire une référence à

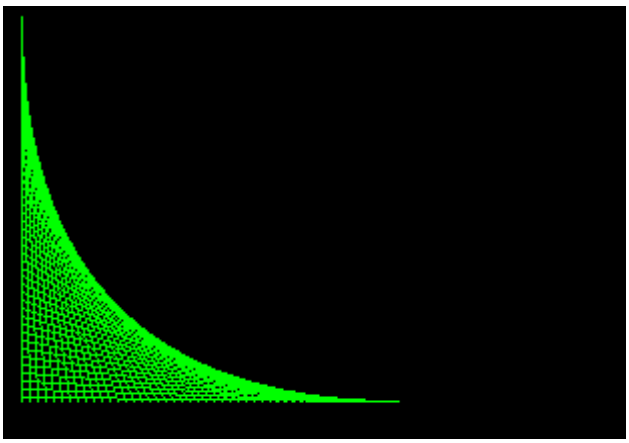
l'espace de nom “Présentation” ppour pouvoir utiliser “SystemMetrics”.

Lancez le code et vous obtiendrez une ligne verte à l'écran.



Essayez d'utiliser ce que vous connaissez des boucles pour dessiner plusieurs lignes :

```
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth, SystemMetrics.ScreenHeight);

         //Efface la mémoire (pas l'écran)
         LCD.Clear();
         int i;
         for (i = 10; i < 200; i += 4)
         {
            //Dessine en mémoire
            LCD.DrawLine(Colors.Green, 1, 10, i, i, 200);
         }
         // Transfère l'image en mémoire vers l'écran
         LCD.Flush();
      }
   }
}
```



Pour écrire un texte, nous devons d'abord avoir une ressource “police de caractères”. Ajoutez une nouvelle ressource au projet. Vous pouvez utiliser un des fichiers livré avec les exemples du SDK NETMF.

Ils se trouvent ici \Mes Documents\Microsoft .NET Micro Framework 4.0\Samples\

J'utilise “NinaB.tinyfnt”. Ajoutez le fichier à vos ressources comme vu précédemmet. Nous pouvons maintenant exécuter ce programme pour écrire sur l'écran LCD.

```
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth, SystemMetrics.ScreenHeight);
         Font MyFont = Resources.GetFont(Resources.FontResources.NinaB);

         //Efface la mémoire mais pas l'écran
         LCD.Clear();
         int i;
         for (i = 10; i < 200; i += 4)
         {
            // Dessine en mémoire
            LCD.DrawLine(Colors.Green, 1, 10, i, i, 200);
         }
         // Affiche un texte sur l'écran
         LCD.DrawText("Still Amazing!", MyFont, Colors.Red, 100, 20);
         // Transfère la mémoire graphique vers l'écran
```

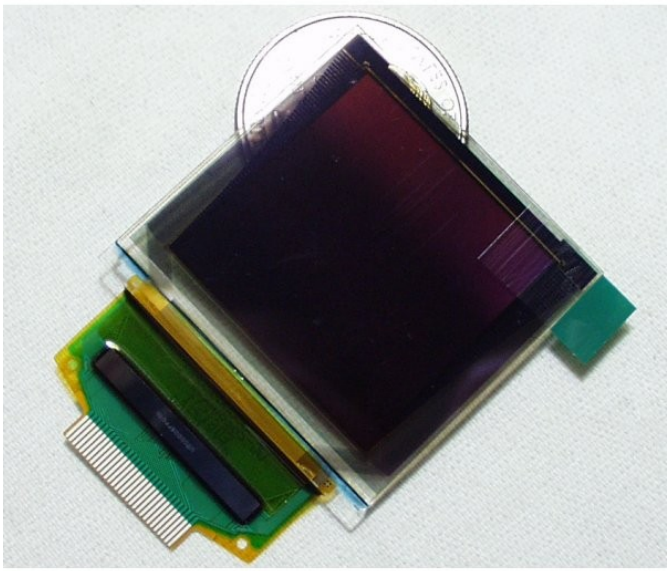
```
            LCD.Flush();
        }
    }
}
```

Support non natif

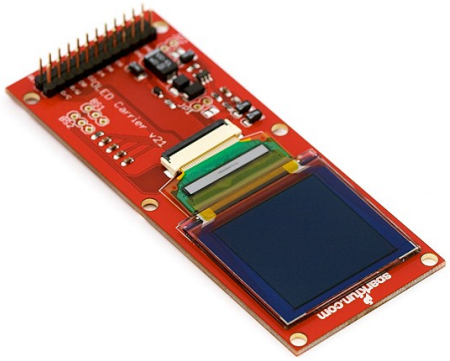
Beaucoup de petits écrans graphiques utilisent SPI pour recevoir des images de l'hôte. Les cartes NETMF comme Emmbedded Master et ChipworkX supportent généralement des grands écrans TFT qui utilisent un bus spécial. Mais, même si le système ne supporte pas de tels écran, comme USBizi et FEZ, un utilisateur peut y connecter un écran utilisant SPI et afficher des graphiques de cette manière. Il est aussi possible d'avoir 2 écrans sur les systèmes qui supportent nativement l'interface TFT. Un écran large tournera sur l'interface TFT et un plus petit utilisera le bus SPI.

Bien que SPI soit très rapide, les écrans contiennent des millions de pixels, il est donc préférable de choisir des modèles pourvus d'un accélérateur graphique. Ces écrans avec accélérateur disposent de commandes spéciales pour certaines tâches. Afficher un cercle peut se faire en envoyant une seule commande plutôt qu'en calculant chaque point et en l'envoyant à l'écran.

Pour la démonstration, j'ai choisi un écran OLED couleur 128x128 proposé sur www.sparkfun.com, sku: LCD-00712.



Sparkfun propose également une carte support pour celui-ci. sku: LCD-00763



Le projet complet est disponible ici :

http://www.microframeworkprojects.com/index.php?title=SimpleGraphicsSupport

Une meilleure option est d'utiliser la classe bitmap pour dessiner le texte et les formes, puis envoyer le bitmap à l'écran. Vous ne pouvez faire ça que le graphique (Bitmap) est supporté par votre carte. USBizi ne les supporte pas.

Ce code affichera quelques données sur l'écran F-51852 128x64 présent sur l'ancienne carte (non-TFT) Embedded Master development system.

```
using System;
using System.Text;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.System;

   public class Program
   {
      static SPI.Configuration conf = new SPI.Configuration((Cpu.Pin)33, false, 0, 0, false, true, 1000,
SPI.SPI_module.SPI2);
      static SPI SPI_port = new SPI(conf);
      static OutputPort RST = new OutputPort((Cpu.Pin)9, true);
      static OutputPort DC = new OutputPort((Cpu.Pin)15, true);
      static byte[] _ba = new byte[1];
      static void OPTREX_SSD_WriteCmdByte(byte b)
      {
         DC.Write(false);
```

```
    Thread.Sleep(1);

    _ba[0] = b;
    SPI_port.Write(_ba);
}

static void OPTREX_SSD_WriteByte(byte b)
{
    DC.Write(true);

    Thread.Sleep(1);

    _ba[0] = b;
    SPI_port.Write(_ba);
}
static void OPTREX_Locate(int x, int y)
{
    if (y > 7)
        y = 7;

    if (x > 127)
        x = 127;

    OPTREX_SSD_WriteCmdByte((byte)(0X10 | (x >> 4)));//col up
    OPTREX_SSD_WriteCmdByte((byte)(0X00 | (x & 0xF)));//col down

    OPTREX_SSD_WriteCmdByte((byte)(0XB0 | y));//page addr

}
public static void Main()
{

    OPTREX_SSD_WriteCmdByte(0XA2);//bias
    OPTREX_SSD_WriteCmdByte(0XA1);//adc   inverse
    OPTREX_SSD_WriteCmdByte(0Xc0);//common dir...normal
    OPTREX_SSD_WriteCmdByte(0X40);//initial line
    OPTREX_SSD_WriteCmdByte(0X81);//evr set
    OPTREX_SSD_WriteCmdByte(0X20);
    OPTREX_SSD_WriteCmdByte(0X29);//2B we have -10V.....wait for stable voltage
    Thread.Sleep(10);
    OPTREX_SSD_WriteCmdByte(0XA4);//turn all on
    OPTREX_SSD_WriteCmdByte(0XE7);//driver
    OPTREX_SSD_WriteCmdByte(0XAF);//lcd on

    //OPTREX_SSD_WriteCmdByte(0XA7);//inverse
    OPTREX_SSD_WriteCmdByte(0XA6);//no inverse


    OPTREX_SSD_WriteCmdByte(0XB0);//page addr
    OPTREX_SSD_WriteCmdByte(0X10);//col
    OPTREX_SSD_WriteCmdByte(0X00);//col
```

```
        int x = 20;
        int y = 50;
        int dx = -2;
        int dy = -3;

        Bitmap bb = new Bitmap(128, 64);
        byte[] bitmapbytes;
        Font fnt = Resources.GetFont(Resources.FontResources.small);
        byte[] vram = new byte[128 * 64 / 8];
        byte[] singleline = new byte[128];
        while (true)
        {
            bb.Clear();
            bb.SetPixel(0, 0, Color.White);
            bb.SetPixel(0, 2, Color.White);
            bb.SetPixel(2, 0, Color.White);
            bb.SetPixel(2, 2, Color.White);
            bb.DrawText("Rocks!", fnt, Color.White, 20, 45);
            bb.DrawEllipse(Color.White, x, y, 5, 3);

            x += dx;
            y += dy;
            if (x < 0 || x > 128)
                dx = -dx;
            if (y < 0 || y > 64)
                dy = -dy;

            bitmapbytes = bb.GetBitmap();
            Util.BitmapConvertBPP(bitmapbytes, vram, Util.BPP_Type.BPP1_x128);

            for (int l = 0; l < 8; l++)
            {
                OPTREX_Locate(0, l);
                DC.Write(true);
                Array.Copy(vram, l * 128, singleline, 0, 128);
                SPI_port.Write(singleline);
            }
            Thread.Sleep(1);
        }
    }
}
```

21. Services de temps

Sur un ordinateur, le temps désigne deux choses. Le temps système, qui représente les impulsions processeur utilisées pour gérer les délais et toutes autres choses liées à la gestion du temps processeur. Et d'un autre côté, on trouve l'horloge temps réel (RTC) qui est utilisée pour correspondre au temps “humain” comme les heures, minutes ou dates, par ex.

21.1. Real Time Clock (RTC)

Toutes les cartes NETMF de GHI ont une RTC incorporée. Cette RTX maintient l'heure et la date. Même si le système est éteint, la RTC sera alimentée par une batterie. Pour que la RTC fonctionne quand le système est éteint, il faudra disposer d'un oscillateur à 32.768Khz et une pile bouton 3V sur une broche spécifique, généralement appelée VBAT. Toutes les cartes et modules GHI à part la FEZ Mini disposent d'un tel crystal. Seule la pile bouton 3V est donc nécessaire.

Il est important de comprendre que les services de temps en NETMF ne sont pas connectés directement à la RTC. Par exemple, vous pouvez régler “l'heure système NETMF” et à partir de ce moment, les temps seront obtenus de manière fiable. Mais ça ne veut pas dire que la RTC matérielle a la bonne heure ! Egalement, un utilisateur pourrait lire l'heure sur un serveur de temps et préférer l'utiliser à celui de la RTC.

Voici un exemple qui règle d'abord l'heure puis l'affiche toutes les 100ms. Ceci règle le temps NETMF, pas la RTC matérielle. Une interruption d'alimentation laissera l'horloge dans un état aléatoire.

```
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using System;

namespace MFConsoleApplication1
{
   public class Program
   {
      public static void Main()
      {
         // On règle la date/heure à 09/09/2010  09:09:09
         DateTime time = new DateTime(2010, 9, 9, 9, 9, 9);
         Utility.SetLocalTime(time);
         while (true)
         {
            Debug.Print(DateTime.Now.ToString());
            Thread.Sleep(100);
```

```
        }

    }
  }
}
```

Pour utiliser la RTC matérielle, nous devons d'abord vérifier si elle contient l'heure correcte. Peut-être que la pile a été changée ou que l'horloge n'a jamais été reglée auparavant. Si la RTC a l'heure correcte, alors on peut la lire et utiliser la RTC (matériel) ensuite pour mettre à jour l'heure NETMF (logiciel). Si l'heure n'est pas correcte, alors il faut la régler.

```
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;

    public class Program
    {
        public static void Main()
        {
            // Pour gérer le temps dans votre applicatoin, régler l'heure au
début de votre programme d'après la RTC.

            // Si elle n'a JAMAIS été réglée auparavant qu'elle utilise une
batterie (chargée), alors on la met à l'heure
            if (RealTimeClock.IsTimeValid == false)
                RealTimeClock.SetTime(new DateTime(2010, 03, 01, 12, 0, 0, 0));

            Utility.SetLocalTime(RealTimeClock.GetTime());
        }
  }
```

21.2. Timers

Le Micro Framework inclue 2 classes de timers : Timer et ExtendedTimes. La classe Timer est identique à celle présente dans le framework complet alors que ExtendedTimer est spécifique à NETMF avec des fonctions supplémentaires. Pour des applications de débutant, je vous suggère de continuer à utiliser les threads avant de vous aventurer dans les timers. Je ne montrerai donc qu'un exemple ici. Ce programme va créer un timer qui démarrera au bout de 5 sec et qui ensuite s'activera toutes les secondes.

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
```

```
{
   public class Program
   {
      static void RunMe(object o)
      {
         Debug.Print("Je suis dans le timer !");
      }
      public static void Main()
      {
         Timer MyTimer = new Timer(new TimerCallback(RunMe), null, 5000, 1000);
         Debug.Print("Le timer démarrera dans 5 secondes et s'activera ensuite toutes les secondes.");
         Thread.Sleep(Timeout.Infinite);
      }
   }
}
```

22. Hôte USB

Il y a souvent une confusion entre “hôte USB” et “périphérique USB”. L'hôte USB est le matériel qui se connecte à plusieur autres matériels USB. Par exemple, le PC est un hôte USB car on peut lui connecter divers périphériques USB tels que souris, claviers ou disques dur. Implementer un périphérique USB est relativement simple alors qu'implémenter un hôte est beaucoup plus compliqué.

L'hôte USB est une fonctionnalité exclusive de GHI Electronics. Grâce à elle, vous pouvez connecter quasiment n'importe quel périphérique USB à votre carte NETMF GHI (UsBizi, Embedded Master, ChipworkX). Elle ouvre de nouveaux horizons pour un système embarqué. Votre produit peut maintenant se connecter à un clavier USB standard ou même accéder à une clé USB !

USB est un système “à chaud”, c'est à dire que les périphériques peuvent être connectés ou déconnectés à n'importe quel moment. Des évènements sont générés lors de ces connexions/déconnexions. Le développeur doit s'inscrire à ces évènement pour gérer les périphériques connectés. Comme il s'agit ici d'un livre pour débutants, je considèrerai que le périphérique est toujours connecté au système.

Un hub USB peut être connecté sur la prise USB Host pour pouvoir ensuite y brancher d'autres périphériques.

Premièrement, nous allons détecter quels périphériques sont conenctés. La première chose à faire est de démarrer le gestionnaire système pour qu'il nous donne la liste des périphériques disponibles. N'oubliez pas d'ajouter l'assembly correspondante à votre projet.

```
using System;
using System.Threading;
using Microsoft.SPOT;

using GHIElectronics.NETMF.USBHost;

namespace Test
{
   class Program
   {
      public static void Main()
      {
         // Inscription aux évènements USBHost
         USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;
         USBHostController.DeviceDisconnectedEvent += DeviceDisconnectedEvent;

         // Dodo !
         Thread.Sleep(Timeout.Infinite);
      }
```

```
        static void DeviceConnectedEvent(USBH_Device device)
        {
            Debug.Print("Périphérique connecté...");
            Debug.Print("ID: " + device.ID + ", Interface: " + device.INTERFACE_INDEX + ", Type: " +
device.TYPE);
        }

        static void DeviceDisconnectedEvent(USBH_Device device)
        {
            Debug.Print("Périphérique déconnecté...");
            Debug.Print("ID: " + device.ID + ", Interface: " + device.INTERFACE_INDEX + ", Type: " +
device.TYPE);
        }
    }
}
```

Dès qu'on détecte un périphérique, on peut communiquer avec lui directement. Cela requiert une certaine connaissances des périphériques USB. Heureusement, la plupart des périphériques font partie de classes standard et GHI fournit des pilotes pour celles-ci.

22.1. Périphériques HID

HID est l'acronyme de Human Interface Devices (périphérique d'interface homme-machine). Les périphériques HID comme les souris, claviers ou joysticks sont directement supportés. L'utilisation de HID se fait par les évènements. Les évènements sont des méthodes que vous attachez à des évènements physiques. Quand cet évènement physique survient, votre méthode est exécutée automatiquement.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
    public class Program
    {
        static USBH_Mouse mouse;
        public static void Main()
        {
            // On s'inscrit aux évènements USBHost
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Dodo
            Thread.Sleep(Timeout.Infinite);
        }
```

```
        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.Mouse)
            {
                Debug.Print("Souris connectée");
                mouse = new USBH_Mouse(device);
                mouse.MouseMove += MouseMove;
                mouse.MouseDown += MouseDown;
            }
        }

        static void MouseMove(USBH_Mouse sender, USBH_MouseEventArgs args)
        {
            Debug.Print("(x, y) = (" + sender.Cursor.X + ", " + sender.Cursor.Y + ")");
        }

        static void MouseDown(USBH_Mouse sender, USBH_MouseEventArgs args)
        {
            Debug.Print("Appui sur bouton : " + args.ChangedButton);
        }
    }
}
```

C'est quasiment la même chose pour un joystick. Voici le même exemple, mais modifié pour fonctionner avec un joystick.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.USBHost;
namespace Test
{
    public class Program
    {
        static USBH_Joystick j;
        public static void Main()
        {
            // On s'inscrit aux évènements USBHost
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Ronflette
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.Joystick)
            {
                Debug.Print("Joystick Connecté");
                j = new USBH_Joystick(device);
```

```
            j.JoystickXYMove += JoystickXYMove;
            j.JoystickButtonDown += JoystickButtonDown;
        }
    }

    static void JoystickButtonDown(USBH_Joystick sender, USBH_JoystickEventArgs args)
    {
        Debug.Print("Appui sur bouton : " + args.ChangedButton);
    }

    static void JoystickXYMove(USBH_Joystick sender, USBH_JoystickEventArgs args)
    {
        Debug.Print("(x, y) = (" + sender.Cursor.X + ", " + sender.Cursor.Y + ")");
    }
  }
}
```

22.2. Périphériques série

L'interface série (UART) est très courante. Il y a plein de fabricants qui créent des circuits qui convertissent l'USB vers la série. GHI supporte les circuits produits par FTDI, Silabs et Prolific.

Egalement, il y a une classe USB standard définie pour les communications série : CDC (Communication Device Class). Cette classe est également supportée.

Notez ici que ces circuits USB sont prévus pour être plus ou moins personnalisés. Par exemple, une compagnie qui utilise un chipset FTDI dans ses produits pour qu'ils soient compatibles série changera certaines infos dans les descripteurs USB pour y mettre son nom et pas celui de FTDI. Ils peuvent également changer les infos “vendor ID” et “product ID”.

```
using System;
using System.Text;
using System.Threading;
using Microsoft.SPOT;

using GHIElectronics.NETMF.USBHost;

namespace Test
{
  class Program
  {
    static USBH_SerialUSB serialUSB;
    static Thread serialUSBThread; // Affiche les données toutes les secondes

    public static void Main()
    {
```

```
            // On s'inscrit aux évènements USBHost
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Dodo
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            Debug.Print("Device connected");

            switch (device.TYPE)
            {
                case USBH_DeviceType.Serial_FTDI: // FTDI reconnu
                    serialUSB = new USBH_SerialUSB(device, 9600, System.IO.Ports.Parity.None, 8,
System.IO.Ports.StopBits.One);
                    serialUSB.Open();
                    serialUSBThread = new Thread(SerialUSBThread);
                    serialUSBThread.Start();

                    break;

                case USBH_DeviceType.Unknown: // SiLabs mais pas reconnu
                    // On force SiLabs
                    USBH_Device silabs = new USBH_Device(device.ID, device.INTERFACE_INDEX,
USBH_DeviceType.Serial_SiLabs, device.VENDOR_ID, device.PRODUCT_ID,
device.PORT_NUMBER);
                    serialUSB = new USBH_SerialUSB(silabs, 9600, System.IO.Ports.Parity.None, 8,
System.IO.Ports.StopBits.One);
                    serialUSB.Open();
                    serialUSBThread = new Thread(SerialUSBThread);
                    serialUSBThread.Start();

                    break;
            }
        }

        static void SerialUSBThread()
        {
            // Affiche le célèbre "Hello World!" chaque seconde
            byte[] data = Encoding.UTF8.GetBytes("Hello World!\r\n");
            while (true)
            {
                Thread.Sleep(1000);

                serialUSB.Write(data, 0, data.Length);
            }
        }
    }
}
```

22.3. Stockage de masse

Les périphériques de stockage comme les disques durs USB ou les clés USB utilisent la même classe USB : MSC (Mass Storage Class). La librairie GHI supporte directement ces périphériques. USB définit uniquement comment écrire/lire les secteurs bruts sur le média. Le système doit alors gérer le système de fichiers ensuite. NETMF supporte FAT32 et FAT16. Pour accéder à des fichiers sur un média USB, nous devons d'abord le détecter et ensuite le “monter”.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
   class Program
   {
      public static void Main()
      {
         // On s'inscrit aux évènements RemovableMedia
         RemovableMedia.Insert += RemovableMedia_Insert;
         RemovableMedia.Eject += RemovableMedia_Eject;

         // On s'inscrit aux évènements USBHost
         USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

         // Dodo
         Thread.Sleep(Timeout.Infinite);
      }

      static void DeviceConnectedEvent(USBH_Device device)
      {
         if (device.TYPE == USBH_DeviceType.MassStorage)
         {
            Debug.Print("Périphérique de stockage de masse USB détecté...");
            //....
            //....
         }
      }
   }
}
```

The next section explains how to access the files on the USB memory.

23. Système de fichiers

Le système de fichiers est supporté par le NETMF 4.0. GHI ajoute plusieurs fonctionnalités au support standard. Par exemple, une carte SD peut être montée sur le système de fichiers ou sur le service USB MSC. Quand elle est montée sur le système de fichiers, les développeurs peuvent accéder aux fichiers. Si elle est montée sur le service MSC, alors un PC connecté au port USB de la carte verra un lecteur de cartes USB avec une carte insérée. C'est pratique pour créer un système de journal, par exemple. La carte acumulera les données sur la carte SD et quand elle sera branchée sur un PC, alors elle deviendra un lecteur de carte avec la même carte insérée. La classe GHI's persistent storage est utilisée pour gérer le montage sur le système de fichiers.

Cette section couvrira uniquement l'utilisation de périphériques de stockage sur le système de fichiers.

23.1. Cartes SD

Premièrement, nous devons détecter l'insertion de la carte SD. Les connecteurs de cartse SD ont généralement un petit interrupteur interne qui se ferme quand la carte est insérée. Dans cet exemple, je considèrerai que la carte est toujours insérée et qu'il n'y a pas besoin de de la détecter. Le programme listera tous les fichiers présent dans le répertoire racine.

```
using System;
using System.IO;
using System.Threading;

using Microsoft.SPOT;
using Microsoft.SPOT.IO;

using GHIElectronics.NETMF.IO;

namespace Test
{
   class Program
   {
      public static void Main()
      {
         // ...
         // La carte SD est insérée
         // Crée un périphérique de stockage
         PersistentStorage sdPS = new PersistentStorage("SD");

         // Monte le système de fichiers
         sdPS.MountFileSystem();
```

```
            // Considère qu'un périphérique est bien disponible et y accède via le Micro Framework
            // pour afficher la liste des fichiers et répertoires
            Debug.Print("Lecture des fichiers et répertoires:");
            if (VolumeInfo.GetVolumes()[0].IsFormatted)
            {
                string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
                string[] files = Directory.GetFiles(rootDirectory);
                string[] folders = Directory.GetDirectories(rootDirectory);

                Debug.Print("Fichiers sur " + rootDirectory + ":");
                for (int i = 0; i < files.Length; i++)
                    Debug.Print(files[i]);

                Debug.Print("Répertoires sur " + rootDirectory + ":");
                for (int i = 0; i < folders.Length; i++)
                    Debug.Print(folders[i]);
            }
            else
            {
                Debug.Print("La carte n'est pas formatée. Formatez-là sur un PC en FAT32/FAT16.");
            }

            // Démonte le système de fichiers
            sdPS.UnmountFileSystem();
        }
    }
}
```

Ili y a plusieurs manières d'ouvrir des fichiers. Je ne parlerai ici que de l'objet FileStream. Cet exemple ouvrira un fichier pour écrire une chaine à l'intérieur. Comme FileStream ne prend que des tableaux d'octets, nous allons devoir convertir notre chaine en un tableau d'octets.

```
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using System.IO;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;

namespace MFConsoleApplication1
{
    public class Program
    {
        static void Main()
        {
            // ... vérifie que la carte SD est insérée

            // La carte SD est insérée
```

```
            // Crée un périphérique de stockage
            PersistentStorage sdPS = new PersistentStorage("SD");

            // Monte le système de fichiers
            sdPS.MountFileSystem();

            // Considère qu'un périphérique de stockage est disponible et y accède à travers NETMF
            string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
            FileStream FileHandle = new FileStream(rootDirectory + @"\hello.txt", FileMode.Create);
            byte[] data = Encoding.UTF8.GetBytes("Cette chaine ira dans le fichier !");
            // Ecrit les données et ferme le fichier
            FileHandle.Write(data, 0, data.Length);
            FileHandle.Close();

            // On démonte le système de fichiers au besoin
            sdPS.UnmountFileSystem();

            // ...
            Thread.Sleep(Timeout.Infinite);

        }
    }
}
```

Vous pouvez vérifier avec un PC et un lecteur de cartes que le fichier est présent. Maintenant, on va ouvrir ce même fichier et lire la chaine écrite précédemment.

```
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using System.IO;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;

namespace MFConsoleApplication1
{
    public class Program
    {
        static void Main()
        {
            // ... vérifie que la carte SD est insérée

            // La carte SD est insérée
            // Crée un périphérique de stockage
            PersistentStorage sdPS = new PersistentStorage("SD");

            // Monte le système de fichiers
            sdPS.MountFileSystem();

            // Considère qu'un périphérique de stockage est disponible et y accède à travers NETMF
```

```
            string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
            FileStream FileHandle = new FileStream(rootDirectory + @"\hello.txt", FileMode.Open,
FileAccess.Read);
            byte[] data = new byte[100];
            // Lit les données et ferme le fichier
            int read_count = FileHandle.Read(data, 0, data.Length);
            FileHandle.Close();
            Debug.Print("Taille des données lues : " + read_count.ToString());
            Debug.Print("Données extraites du fichier :");
            Debug.Print(data.ToString());

            // On démonte le système de fichiers  au besoin
            sdPS.UnmountFileSystem();

            // ...
            Thread.Sleep(Timeout.Infinite);

        }
    }
}
```

23.2. Disques USB

Les fichiers sont traités exactement de la même manières sur des disques USB que sur des cartes SD. La seule différence réside dans la détection du périphérique et la façon de le monter. Pour les cartes SD, on pouvait utiliser une broche d'entrée pour les détecter. Avec un disque USB, nous utilisons les évènements pour détecter un nouveau média.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
    class Program
    {
        // Maintient une référence statique au cas-où le ramasse-miettes viendrait à le supprimer
automatiquement. Notez que nous ne supportons qu'un seul lecteur dans cet exemple.
        static PersistentStorage ps;

        public static void Main()
        {
            // Souscrit aux évènements RemovableMedia
            RemovableMedia.Insert += RemovableMedia_Insert;
            RemovableMedia.Eject += RemovableMedia_Eject;
```

```
        // Souscrit aux évènements USBHost
        USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

        // Au dodo
        Thread.Sleep(Timeout.Infinite);
    }

    static void DeviceConnectedEvent(USBH_Device device)
    {
        if (device.TYPE == USBH_DeviceType.MassStorage)
        {
            Debug.Print("Disque USB détecté...");
            ps = new PersistentStorage(device);
            ps.MountFileSystem();
        }
    }

    static void RemovableMedia_Insert(object sender, MediaEventArgs e)
    {
        Debug.Print("Média \"" + e.Volume.RootDirectory + "\" inséré.");
        Debug.Print("Récupération des fichiers et répertoires :");
        if (e.Volume.IsFormatted)
        {
            string[] files = Directory.GetFiles(e.Volume.RootDirectory);
            string[] folders = Directory.GetDirectories(e.Volume.RootDirectory);

            Debug.Print("Fichiers sur  " + e.Volume.RootDirectory + ":");
            for (int i = 0; i < files.Length; i++)
                Debug.Print(files[i]);

            Debug.Print("Répertoires sur " + e.Volume.RootDirectory + ":");
            for (int i = 0; i < folders.Length; i++)
                Debug.Print(folders[i]);
        }
        else
        {
            Debug.Print("Le disque n'est pas formaté. Formatez-le sur un PC en FAT32/FAT16, SVP..");
        }
    }

    static void RemovableMedia_Eject(object sender, MediaEventArgs e)
    {
        Debug.Print("Média \"" + e.Volume.RootDirectory + "\" éjecté.");
    }
  }
}
```

Nous pouvons donc voir qu'après avoir monté le disque sur le système de fichier que tout le reste fonctionne exactement pareil que pour une carte SD.

23.3. A propos du système de fichiers

Le support par NETMF 4.0 du système de fichier FAT ne gère que FAT32 et FAT16. Un média formaté en FAT12 ne fonctionnera pas.

Le système de fichiers garde en mémoire un maximum de données en interne pour accélérer les accès fichiers et prolonger la durée de vie de la mémoire flash du média. Quand vous écrivez des données dans un fichier, il n'est pas certain que ces données soient écrites sur la carte. Elles sont probablement quelque part en mémoire, dans les tampons internes. Pour vous assurer que les données sont bien écrites sur le média, vous devez vider ces buffers en utilisant la méthode Flush(). Vider les buffers ou fermer le fichier sont les seules façon de garantir que les données sont réellement inscrites sur le média.

Ceci se passait au niveau du fichier. Au niveau du média lui-même, il y a aussi des informations qui peuvent être retardées. Par exemple, si vous supprimez un fichier et que vous retirez la carte, rien n'indique que le fichier est réellement supprimé. Pour vous en assurer, vous devez utiliser la méthode VolumeInfo.FlushALL().

Idéalement, vous devrier démonter (Unmount) le média **avant** de l'enlever du système. Cela n'est pas forcément possible, donc un Flush() ou FlushAll() vous garantira que vos données ont bien été écrites et qu'aucune perte de données n'est à crainte en cas de retrait intempestif du média.

24. Réseau

Les réseaux représent une partie essentielle de notre travail ou notre vie. Presque tous les foyers sont connectés à un réseau (internet) et la plupart des entreprises ne peuvent fonctionner sans un réseau (LAN ou WiFi) qui soit connecté à un réseau externe (internet). Tous ces réseaux communiquent d'une façon standard, en utilisant le protocole TCP/IP. Il y a en fait plusieurs protocoles qui prennent en charge différentes tâches dans un réseau : DNS, DHCP, IP, ICMP, TCP, UDP, PPP...et encore plein d'autres !

NETMF supporte les réseaux TCP/IP à travers les sockets standard .NET. Une socket est une connexion virtuelle entre deux périphériques sur un réseau.

GHI a étendu le support TCP/IP pour couvrir PPP et WiFi. A travers PPP, deux périphériques peuvent se connecter via une connexion série. La connexion série peut être un modem analogique ou un modem 3G/GPRS. Avec PPP, les périphériques NETMF peuvent se connecter à internet en utilisant la connexion 3G d'un téléphone portable. On peut également connecter deux périphériques NETMF à travers une connexion filaire ou sans fil (Xbee/Bluetooth/others).

Egalement, avec le support du WiFi, les périphériques NETMF peuvent se connecter à des réseaux sans fils sécurisés ou non.

NETMF supporte également les connexions sécurisées SSL.

Le support réseau est implémenté en standard et de façon complète (HTTP, SSL, Sockets...etc.) sur les cartes EMX, Embedded Master et ChipworkX.

24.1. Support réseau avec USBizi (FEZ)

Pour gérer un réseau gourmand en mémoire, USBizi a besoin d'un support externe. Par exemple, USBizi gère le réseau filaire à travers le chipset Wiznet W5100. Quand USBizi veut créer une connexion réseau, la seule chose à faire est d'envoyer la requête au Wiznet 5100 et la puce s'occupera du reste. Idem pour les transferts de données, où USBizi donnera les données au W5100 qui se chargera de les envoyer.

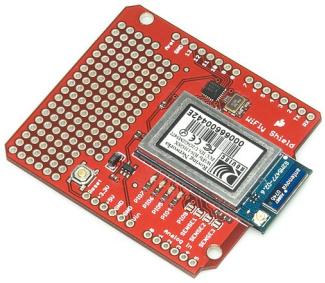
La puce Wiznet W5100 fonctionne sur le bus SPI. Le pilote actuel fourni par GHI permet d'utiliser le W5100 directement depuis C#. Ce pilote est une béta-version, pour des tests uniquement. Un développement est en cours pour fournir le support du W5100 directement dans le firmware USBizi. GHI fourni ce pilote pour plus de commodité et optimise son fonctionnement. Tant que le pilote GHI n'est pas disponible, chacun ayant un peu d'expérience peut implémenter un pilote W5100 selon ses besoins.

24.2. TCP/IP brut ou Sockets ?

C'est un domaine où la plupart des designers peuvent se tromper. Quand on sélectionne un module WiFi ou GPRS/3G, il y a deux catégories de modules à choisir. Soit des muldes “Sockets”, soit des modules “raw TCP/IP”. Les modules avec sockets gèrent tout le travail TCPIP en interne et vous donnent un accès de haut niveau sur les sockets. Cela signifie que le travail le plus pénible est fait en interne dans le module. Une fois que vous avez assigné une adresse et un port, vous n'avez plus qu'à envoyer/recevoir des données depuis le module. Problème : c'est très limité. Par exemple en nombre de sockets. Même si votre système est très puissant, a plusieurs MB de RAM, vous êtes limité par les fonctionnalités du module. Pour cette raison néanmoins, l'utilisation de tels modules “haut niveau” est idéale pour des petits systèmes.

Prenons un exemple : Roving Networks propose un module appelé WiFly. Ce module contient une pile TCP/IP et une seule socket. L'utilisation de ce module est très simple puisqu'il vous suffit de le connecter à un port série de votre carte, puis, avec de simples commandes série, vous pouvez lire/écrire des données depuis une des sockets disponibles. Cela suffit pour faire un petit serveur web, une connexion Telnet ou du transfert de données. C'est parfait pour les petits systèmes limités en mémoire et en ressources comme USBizi (FEZ). Le module fait tout pour vous, vous n'avez qu'à envoyer et recevoir des données série.

Si vous implémentez un serveur web qui fournit certaines valeurs comme la température ou l'humidité, alors c'est tout ce dont vous avez besoin et ça peut être implémenté facilement et pour pas cher avec USBizi (FEZ). Un prototype simple peut se faire en connectant l'extension WiFly à la carte FEZ Domino. Sur l'image de droite, on voit la carte WiFly de SparkFun.



Le côté négatif de ces modules simples est que vous êtes très limité en sockets et en nombre de connexions. Comment faire si vous avez besoin de plus ? Comment faire si vous voulez implémenter une connexion SSL ? Pour tout cela, vous aurez besoin d'un module qui ne gère pas TCP/IP en interne. Le travail TCP/devra être fait à l'extérieur du module. C'est ici que les cartes comme EMX and ChipworkX entrent en jeu. Ces cartes sont puissantes, avec beaucoup de ressources. Elles ont une pile TCP/IP en interne avec support de SSL/HTTP/DHCP...etc. Connecter un module comme le ZeroG à une carte EMX ou ChipworkX décuplera la puissance de votre carte, avec en plus des connexion Wifi sécurisées.

Et en ce qui conerne les modems GPRS et 3G ? C'est la même chose. Certains ont un support natif des sockets, comme le SM5100B alors que d'autres fonctionnent à travers PPP comme un modem de PC (module Telit, par ex). Si vous avez besoin d'une connexion réseau avec une pile TCP/IP complète, alors vous devrez vous orienter vers EMX ou ChipworkX avec des modems PPP standard, exactement comme un PC se connecterait à internet avec un modem.

Si vous avez besoin d'une connexion simple et bon marché, alors USBizi (FEZ) peut être utilisé avec le SM5100B. L'extension SpoarkFun Cellular Shield montrée à droite se monte directement sur la carte FEZ Domino/

24.3. Sockets standards .NET

Le support des sockets sur NETMF est très similaire à celui présent dans le .Net Framework complet. Le SDK NETMF inclut plusieurs exemple d'utilisation des sockets, clients et serveurs. Il y a également plusieurs projets montrant les différentes possibilités

Client Twitter :

http://www.microframeworkprojects.com/index.php?title=MFTwitter

Google maps:

http://www.microframeworkprojects.com/index.php?title=GoogleMaps

Client RSS :

http://www.microframeworkprojects.com/index.php?title=RSS_n_Weather

Radio internet MP3 :

http://www.microframeworkprojects.com/index.php?title=ShoutcastClient

Serveur Web :

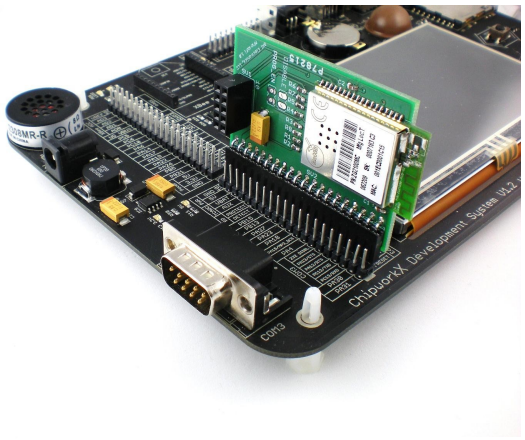
http://www.microframeworkprojects.com/index.php?title=WebServer

24.4. Wi-Fi (802.11)

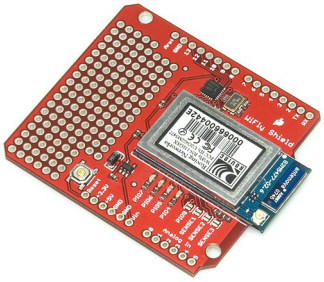
Le WiFi est très répandu dans les réseaux. Il permet des transferts sécurisés entre de multiples connexions. Il permet également des communications entre deux noeuds (réseau Ad-Hoc). La façon la plus utilisée consiste à connecter plusieurs ordinateurs à un point d'accès. Le Wifi n'est pas simple et surtout pas prévu pour être embarqué simplement. GHI Electronics est la seule entreprise qui offre l'option Wifi pour ses cartes NETMF.

WiFi est prévu pour fonctionner avec des piles TCP/IP (sockets réseau sur NETMF) et ne peut donc être utilisé que sur des systèmes qui supportent déjà TCPIP, comme EMX et ChipworkX.

Le Wifi supporté par GHI utilise les puces ZG2100 et ZG2101 (antenne interne et externe, respectivement) de ZeroG. Pour la réalisation de prototypes, l'extension WiFi est un bon départ.



Comme on l'a vu précédemment, USBizi (FEZ) peut être utilisé avec des modules qui contiennent en interne une pile TCP/IP et le support des sockets, comme l'extension de SparkFun, qui contient le module WiFly de Roving Networks.



24.5. Réseaux GPRS et 3G Mobile

EMX et ChipworkX ont un support TCP/IP et PPP en interne. Vous pouvez y connecter un modem standard et l'utiliser en quelques instants.

Concernant USBizi, une connexion à un réseau mobile peut être établie en utilisant un modem implementant TCP/IP comme le SM5100B. Ci-dessous l'extension SparkFun Cellular shield (avec un SM5100B)

25. Cryptographie

La cryptographie a représenté une part importante de la technologie depuis de nombreuses années. Les algorithmes modernes de cryptographie peuvent être très gourmands en ressources. Comme les cartes NETMF sont prévues pour des petits appareils, l'équipe NETMF a dû bien choisir quels alorithmes supporter. Ces algorithmes sont XTEA et RSA.

25.1. XTEA

XTEA, avec sa clé 16 octets (128 bits) est considérée comme bien sécurisée tout en ne nécessitant pas beaucoup de puissance processeur. A l'origine, XTEA a été pensé pour travailler sur des “morceaux” de 8 octets seulement. Cela peut poser problème pour encrypter une donnée dont la taille n'est pas un multiple de 8. L'implémentation de XTEA sur NETMF autorise l'encryptage de donnes de n'importe quelle taille.

L'encryptage et le décryptage sont faciles à faire. Voici un exemple :

```
using System;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Cryptography;
public class Program
{
   public static void Main()
   {
      // 16-octets (clé 128-bits)
      byte[] XTEA_key = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
      Key_TinyEncryptionAlgorithm xtea = new Key_TinyEncryptionAlgorithm(XTEA_key);
      // Les données que nous voulons encrypter
      string original_string = "FEZ is so easy!";//doit être plus grand que 8 octets
      // on les convertit en tableau d'octets
      byte[] original_data = UTF8Encoding.UTF8.GetBytes(original_string);

      //Encryptage des données
      byte[] encrypted_bytes = xtea.Encrypt(original_data, 0, original_data.Length, null);
      //Décryptage des données
      byte[] decrypted_bytes = xtea.Decrypt(encrypted_bytes, 0, encrypted_bytes.Length, null);
      // Affiche les données décryptées
      string decrypted_string = new string(Encoding.UTF8.GetChars(decrypted_bytes));
      Debug.Print(original_string);
      Debug.Print(decrypted_string);
   }
}
```

Les données cryptées ont la même taille que les données brutes.

XTEA sur les PC

Maintenant, vous pouvez partager des données entre des périphériques NETMF de manière sécurisée en utilisant XTEA, mais comment faire pour échanger ces données avec un PC ou un autre système ? Le livre “Expert .NET Micro Framework Second Edition” de Jens Kuhner fournit des codes sources montrant comment implémenter XTEA sur un PC. En utilisant le code de Jens, vous pouvez encrypter des données et les envoyer à une carte NETMF et vice-versa. Si vous n'avez pas le livre, le code source est en ligne, chapitre 8 :

Vous pouvez le télécharger ici : http://apress.com/book/view/9781590599730

25.2. RSA

XTEA est bien sécurisé, mais il a une limitation importante : la clé doit être partagée. Pour pouvoir partager des données cryptées, les deux systèmes doivent d'abord connaitre la clé utilisée. Si un des système essaye d'envoyer la clé à l'autre, quelqu'un pourrait espionner la ligne et obtenir la clé et ainsi décrypter les données. Ce n'est plus franchement sécurisé !

RSA contourne ce problème en fournissant une combinaison clé publique/clé privée. Cela peut paraitre absurde, mais la clé utilisée pour encrypter ne peut pas être utilisée pour décrypter. Une clé différente doit être utilisée pour décrypter les données. Imaginons que le système 'A' doivent lire des données sécurisées du système 'B'. La première chose que 'A' fera est d'envoyer une clé publique au système 'B'. 'B' va alors encrypter les données avec la clé publique et envoyer les données au PC. Un hacker peut voir les données encryptées et la clé publique, mais sans la clé privée, le décryptage est quasiment impossible. Enfin, le système 'A' peut décrypter les données grâce à sa clé privée.

Au passage, les sites web sécurisés fonctionnent ainsi.

Les cartes NETMF ne peuvent pas générer de clés. Celles-ci sont générées sur un PC avec un programme appelé MetaDataProcessor. Ouvrez une fenêtre en ligne de commande et entrez ces instructions :

cd “C:\Program Files (x86)\Microsoft .NET Micro Framework\v4.0\Tools”

Ce répertoire dépend de votre installation de .NET Micro Framework.

Générez les clés en procédant ainsi :

MetadataProcessor.exe -create_key_pair c:\private.bin c:\public.bin

Les clés sont codées en binaire, mais cet outil peut les convertir en texte lisible :

MetadataProcessor.exe -dump_key c:\public.bin >> c:\public.txt

MetadataProcessor.exe -dump_key c:\private.bin >> c:\private.txt

Maintenant, copiez la clé dans notre programme exemple. Notez que la clé publique commence toujours par 1,0,1 et que le reste est composé de 0. Ca nous permet d'optimiser

un peu notre code.

```
using System;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Cryptography;
public class Program
{
   public static void Main()
   {
      //Ceci est partagé entre la clé publique et la clé privée
      byte[] module = new byte[] { 0x17, 0xe5, 0x27, 0x40, 0xa9, 0x15, 0xbd, 0xfa, 0xac, 0x45, 0xb1,
0xb8, 0xe1, 0x7d, 0xf7, 0x8b, 0x6c, 0xbd, 0xd5, 0xe3, 0xf4, 0x08, 0x83, 0xde, 0xd6, 0x4b, 0xd0, 0x6c,
0x76, 0x65, 0x17, 0x52, 0xaf, 0x1c, 0x50, 0xff, 0x10, 0xe8, 0x4f, 0x4f, 0x96, 0x99, 0x5e, 0x24, 0x4c,
0xfd, 0x60, 0xbe, 0x00, 0xdc, 0x07, 0x50, 0x6d, 0xfd, 0xcb, 0x70, 0x9c, 0xe6, 0xb1, 0xaf, 0xdb, 0xca,
0x7e, 0x91, 0x36, 0xd4, 0x2b, 0xf9, 0x51, 0x6c, 0x33, 0xcf, 0xbf, 0xdd, 0x69, 0xfc, 0x49, 0x87, 0x3e,
0x1f, 0x76, 0x20, 0x53, 0xc6, 0x2e, 0x37, 0xfa, 0x83, 0x3d, 0xf0, 0xdc, 0x16, 0x3f, 0x16, 0xe8, 0x0e,
0xa4, 0xcf, 0xcf, 0x2f, 0x77, 0x6c, 0x1b, 0xe1, 0x88, 0xbd, 0x32, 0xbf, 0x95, 0x2f, 0x86, 0xbb, 0xf9,
0xb4, 0x42, 0xcd, 0xae, 0x0b, 0x92, 0x6a, 0x74, 0xa0, 0xaf, 0x5a, 0xf9, 0xb3, 0x75, 0xa3 };
      //La clé privée, pour le décryptage
      byte[] private_key = new byte[] { 0xb9, 0x1c, 0x24, 0xca, 0xc8, 0xe8, 0x3d, 0x35, 0x60, 0xfc, 0x76,
0xb5, 0x71, 0x49, 0xa5, 0x0e, 0xdd, 0xc8, 0x6b, 0x34, 0x23, 0x94, 0x78, 0x65, 0x48, 0x5a, 0x54, 0x71,
0xd4, 0x1a, 0x35, 0x20, 0x00, 0xc6, 0x0c, 0x04, 0x7e, 0xf0, 0x34, 0x8f, 0x66, 0x7f, 0x8a, 0x29, 0x02,
0x5e, 0xe5, 0x39, 0x60, 0x15, 0x01, 0x58, 0x2b, 0xc0, 0x92, 0xcd, 0x41, 0x75, 0x1b, 0x33, 0x49, 0x78,
0x20, 0x51, 0x19, 0x3b, 0x26, 0xaf, 0x98, 0xa5, 0x4d, 0x14, 0xe7, 0x2f, 0x95, 0x36, 0xd4, 0x0a, 0x3b,
0xcf, 0x95, 0x25, 0xbb, 0x23, 0x43, 0x8f, 0x99, 0xed, 0xb8, 0x35, 0xe4, 0x86, 0x52, 0x95, 0x3a, 0xf5,
0x36, 0xba, 0x48, 0x3c, 0x35, 0x93, 0xac, 0xa8, 0xb0, 0xba, 0xb7, 0x93, 0xf2, 0xfd, 0x7b, 0xfa, 0xa5,
0x72, 0x57, 0x45, 0xc8, 0x45, 0xe7, 0x96, 0x55, 0xf9, 0x56, 0x4f, 0x1a, 0xea, 0x8f, 0x55 };
      // la clé publique commence toujours par 1,0,1 . Le reste est composé de 0
      byte[] public_key = new byte[128];
      public_key[0] = public_key[2] = 1;

      Key_RSA rsa_encrypt = new Key_RSA(module, public_key);
      Key_RSA rsa_decrypt = new Key_RSA(module, private_key);

      // La donnée à encrypter
      string original_string = "FEZ is so easy!";
      // On la convertit en tableau d'octets
      byte[] original_data = UTF8Encoding.UTF8.GetBytes(original_string);
      // Et on l'encrypte
      byte[] encrypted_bytes = rsa_encrypt.Encrypt(original_data, 0, original_data.Length, null);
      //Puis on la décrypte
      byte[] decrypted_bytes = rsa_decrypt.Decrypt(encrypted_bytes, 0, encrypted_bytes.Length, null);
      //Et on affiche le résultat
      string decrypted_string = new string(Encoding.UTF8.GetChars(decrypted_bytes));
      Debug.Print("Taille des données non cryptées = " + original_string.Length + " Données = " +
original_string);
      Debug.Print("Taille des données cryptées = " + encrypted_bytes.Length + " Données décryptées = "
+ decrypted_string);
   }
}
```


La taille des données encryptées avec RSA n'est pas la même que celle des données originales. Ayez ceci en tête quand vous prévoyer de transférer ou sauvegarder des données cryptées.

RSA nécessite beaucoup de plus de puissance processeur que XTEA, ce qui peut poser problème sur des petits système. Je vous suggère donc de commencer une session RSA pour échanger des clés XTEA et des infos de sécurité, puis de poursuivre avec XTEA.

26. XML

26.1. Théorie XML

eXtensible Markup Language (XML) est une langage permettant une organisation standard des données informatiques. Quand vous voulez transférer des données entre deux périphériques, vous pouvez décider de la façon dont sont organisées les données envoyées depuis le périphérique A. De l'autre côté, le périphérique B qui les reçoit saura comment elles sont organisées. Avant XML, cela pouvait poser quelques problèmes : que se passait-il quand vous vouliez envoyer des données à un système développé par une autre personne ? Vous deviez lui expliquer comment vous aviez organisé vos données pour qu'il puisse les lire correctement. Maintenant, les développeurs peuvent utiliser XML pour une organisation commune des données.

XML est très utilisé quotidiennement et de plusieurs manières. Par exemple, si un site web marchand veut connaitre le montant des frais d'expédition d'un panier, il formatera le contenu du panier en XML et l'enverra à FedEx (par ex). Le site web de FedEx saura interpréter le contenu du panier et enverra en retour les infos sur l'expédition, toujours au format XML.

L'efficacité de XML peut aussi être utile dans d'autres cas. Imaginons que vous programmiez un journal de données. Imaginons également que l'utilisateur final puisse configurer ce qu'il veut tracer. Vous avez donc besoin de sauvegarder quelque part ces infos. Vous pourriez utiliser votre propre format, mais cela nécessiterait plus de code et de débogage. Le mieux est donc d'utiliser XML, qui est un standard.

Toutes les cartes NETMF de GHI Electronics ont un support pour la lecture/écriture XML.

Voici un exemple de fichier XML qui nous sera utile pour notre journal. J'ai conservé ici les termes anglais pour la compréhension générale.

```
<?xml version="1.0" encoding="utf-8" ?>
<NETMF_DataLogger>
  <FileName>Data</FileName>
  <FileExt>txt</FileExt>
  <SampleFreq>10</SampleFreq>
</NETMF_DataLogger>
```

Cet exemple montre un élément père et 3 fils. J'ai choisi cette structure, mais vous auriez très bien pu ne mettre que des éléments père. XML est très arrangeant, voire trop, parfois !Retour à notre exemple. L'élément père "NETMF_DataLogger" contient 3 morceaux d'information qui sont importantes pour notre journal. Il s'agit du nom du fichier, de son extension et de la fréquence de sauvegarde. Cet exemple créera un fichier Data.txt et le remplira 10 fois par

seconde.

Un autre usage important pour nous, les “développeurs pour embarqué”, est le partage d'infos avec des plus gros système, comme les PC. Comme tous les PC gèrent d'une façon ou d'une autre le XML, vous pouvez enoyer/recevoir des données depuis le PC en utilisant XML.

Les espaces et la présentation ne sont pas significatifs en XML. Ces choses sont uniquement utiles à nous, humains, pour des raisons de lisibilité. Le même exemple ci-dessus aurait très bien pu être écrit de cette façon :

```
<?xml version="1.0" encoding="utf-8" ?><NETMF_DataLogger>  <FileName>Data</FileName>
<FileExt>txt</FileExt><SampleFreq>10</SampleFreq></NETMF_DataLogger>
```

Vous comprenez pourquoi la présentation est importante pour les humains ?

Vous pouvez aussi ajouter des commentaires dans un fichier XML. Les commentaires ne représentent rien du point de vue XML mais permettent une lecture plus facile du fichier par un humain.

```
<?xml version="1.0" encoding="utf-8" ?>
<!--This is just a comment-->
<NETMF_DataLogger>
  <FileName>Data</FileName>
  <FileExt>txt</FileExt>
  <SampleFreq>10</SampleFreq>
</NETMF_DataLogger>
```

Enfin, XML peut utiliser des attributs. Un attribut est une info supplémentaire donnée à un élément. Mais pourquoi utiliser un attribut si vous pouvez ajouter un nouvel élément décrivant cet attribut ? Vous n'avez donc pas spécialement besoin des attributs et je dirais même que si vous n'avez pas une bonne raison de les utiliser et bien ne le faites pas ! Je ne les expliquerai d'ailleurs pas dans ce livre.

26.2. Créer un fichier XML

Les cartes NETMF de GHI Electronics permettent la lecture et l'écriture au format XML. Ceci se fait à travers les “streams”, ce qui signifie que n'importe quel stream (standard ou personnalisé) peut fonctionner avec XML. Ici, nous allons utiliser les streams de base MemoryStream et FileStream mais vous pourriez utiliser votre propre stream, ce qui n'est pas décrit dans ce livre.

Ce code montre comment créer un document XML en mémoire. C'est une reprise de notre code précédent.


```
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
   public static void Main()
   {
      MemoryStream ms = new MemoryStream();

      XmlWriter xmlwrite = XmlWriter.Create(ms);

      xmlwrite.WriteProcessingInstruction("xml", "version=\"1.0\" encoding=\"utf-8\"");
      xmlwrite.WriteComment("This is just a comment");
      xmlwrite.WriteStartElement("NETMF_DataLogger");// Elément père
      xmlwrite.WriteStartElement("FileName");//Elément fils
      xmlwrite.WriteString("Data");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteStartElement("FileExt");
      xmlwrite.WriteString("txt");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteStartElement("SampleFeq");
      xmlwrite.WriteString("10");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteEndElement();//Fin de l'élément père

      xmlwrite.Flush();
      xmlwrite.Close();
      //////// Affiche les données XML ///////////
      byte[] byteArray = ms.ToArray();
      char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
      string str = new string(cc);
      Debug.Print(str);
   }
}
```

Note importante : avec NETMF, les librairies XML writer et XML reader sont deux librairies séparées. Le reader vient de l'assembly "System.Xml" et le writer de "MFDpwsExtensions" ! Si vous voulez savoir pourquoi, vous n'avez qu'à le demander à Microsoft! Le reader se trouve dans l'espace de nom "System.Xml" mais le writer est lui dans "System.Ext.Xml". Pour vous faciliter la vie, incluez les deux assemblies et espaces de noms à chaque fois, comme j'ai fait dans le code auparavant.

Note: Quand vous allez ajouter l'assembly, vous verrez qu'il en existe deux pour XML : "System.Xml" et "System.Xml.Legacy". N'utilisez jamais le pilote “legacy” car il est très lent et consomme beaucoup de mémoire. Elle existe uniquement pour les systèmes qui n'ont pas la gestion de XML en standard. Toutes les cartes NETMF de GHI Electronics ont un support natif (très rapide) de XML, donc vous devriez utiliser uniquement "System.Xml".

En exécutant le programme ci-dessus, vous verrez les données s'afficher à la fin. Ces données sont correctes mais pas forcément bien lisibles. Notez que nous écrivons/lisons des fichiers XML sur un très petit système et donc moins il y a d'info (espaces, formatage) mieux c'est. Donc c'est mieux de n'avoir aucun formatage, mais bon, pour que les choses soient quand même plus sympa, nous allons ajouter quelques lignes pour améliorer l'affichage

```
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
   public static void Main()
   {
      MemoryStream ms = new MemoryStream();

      XmlWriter xmlwrite = XmlWriter.Create(ms);

      xmlwrite.WriteProcessingInstruction("xml", "version=\"1.0\" encoding=\"utf-8\"");
      xmlwrite.WriteComment("This is just a comment");
      xmlwrite.WriteRaw("\r\n");
      xmlwrite.WriteStartElement("NETMF_DataLogger");// Elément père
      xmlwrite.WriteString("\r\n\t");
      xmlwrite.WriteStartElement("FileName");// Elément fils
      xmlwrite.WriteString("Data");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n\t");
      xmlwrite.WriteStartElement("FileExt");
      xmlwrite.WriteString("txt");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n\t");
      xmlwrite.WriteStartElement("SampleFeq");
      xmlwrite.WriteString("10");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n");
      xmlwrite.WriteEndElement();// Fin de l'élément père

      xmlwrite.Flush();
      xmlwrite.Close();
      //////// Affiche les données XML ///////////
      byte[] byteArray = ms.ToArray();
      char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
      string str = new string(cc);
      Debug.Print(str);
   }
}
```

26.3. Lire du XML

Créer un fichier XML est plus facile que le lire (analyser). Il y a plusieurs façons de lire un fichier XML, mais vous pouvez simplement le parcourir et lire les infos au fur et à mesure jusqu'à la fin du fichier. Cet exemple crée des données XML et les lit en retour.

```
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
   public static void Main()
   {
      MemoryStream ms = new MemoryStream();

      XmlWriter xmlwrite = XmlWriter.Create(ms);

      xmlwrite.WriteProcessingInstruction("xml", "version=\"1.0\" encoding=\"utf-8\"");
      xmlwrite.WriteComment("This is just a comment");
      xmlwrite.WriteRaw("\r\n");
      xmlwrite.WriteStartElement("NETMF_DataLogger");// Elément père
      xmlwrite.WriteString("\r\n\t");
      xmlwrite.WriteStartElement("FileName");// Elément fils
      xmlwrite.WriteString("Data");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n\t");
      xmlwrite.WriteStartElement("FileExt");
      xmlwrite.WriteString("txt");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n\t");
      xmlwrite.WriteStartElement("SampleFeq");
      xmlwrite.WriteString("10");
      xmlwrite.WriteEndElement();
      xmlwrite.WriteRaw("\r\n");
      xmlwrite.WriteEndElement();// Fin de l'élément père

      xmlwrite.Flush();
      xmlwrite.Close();
      //////// Affiche les données XML //////////
      byte[] byteArray = ms.ToArray();
      char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
      string str = new string(cc);
      Debug.Print(str);

      ////////// Lecture des données XML
      MemoryStream rms = new MemoryStream(byteArray);

      XmlReaderSettings ss = new XmlReaderSettings();
      ss.IgnoreWhitespace = true;
```

```
        ss.IgnoreComments = false;
        //XmlException.XmlExceptionErrorCode.
        XmlReader xmlr = XmlReader.Create(rms,ss);
        while (!xmlr.EOF)
        {
            xmlr.Read();
            switch (xmlr.NodeType)
            {
                case XmlNodeType.Element:
                    Debug.Print("Elément: " + xmlr.Name);
                    break;
                case XmlNodeType.Text:
                    Debug.Print("Texte: " + xmlr.Value);
                    break;
                case XmlNodeType.XmlDeclaration:
                    Debug.Print("Déclaration: " + xmlr.Name + ", " + xmlr.Value);
                    break;
                case XmlNodeType.Comment:
                    Debug.Print("commentaire : " +xmlr.Value);
                    break;
                case XmlNodeType.EndElement:
                    Debug.Print("Fin de l'élément");
                    break;
                case XmlNodeType.Whitespace:
                    Debug.Print("Espace");
                    break;
                case XmlNodeType.None:
                    Debug.Print("Vide");
                    break;
                default:
                    Debug.Print(xmlr.NodeType.ToString());
                    break;
            }
        }
    }
}
```

27. Ajouter des E/S

Une application peut nécessiter plus de broches numériques ou analogiques que disponibles sur le processeur. Il y a plusieurs façons d'augmenter ce nombre.

27.1. Numériques

Le moyen le plus facile d'augmenter le nombre de borches numériques consiste à utiliser un registre à décalage. Ce registre se connectera au bus SPI. Via SPI, nous pouvons envoyer/recevoir l'état des broches. Les registres à décalage peuvent être connectés en cascade, ce qui permet en théorie un nombre illimité de borches digitales.

Les registres à décalage possèdent en général 8 broches numériques. Si nous en connectons 3 sur une carte via SPI, nous aurons 24 nouvelles broches numériques disponibles, tout en utilisant que les broches SPI du processeur.

Matrice de boutons

Les appareils comme les fours à micro-ondes ont souvent plusieurs boutons en façade. Normalement, on n'appuie jamais sur 2 boutons en même temps, donc on peut les grouper en matrice. Si nous avons 12 boutons sur notre système, nous aurions besoin de 12 entrées numériques sur le processeur pour toutes les lire. Mais en connectant ces boutons avec une matrice 4x3, nous n'avons plus besoin que de 7 broches au lieu de 12. De nombreuses matrices de boutons existent et peuvent être intégrées dans votre produit de cette façon.

Pour connecter les boutons en matrice, nous câblerons notre circuit de façon à avoir des lignes et des colonnes. Chaque bouton se connectera à une ligne et une colonne. C'est tout pour la partie matériel ! Notez que si nous n'utilisons pas une matrice, le bouton sera connecté entre la broche et la masse..

Pour lire les boutons, assignez toutes les colonnes à des sorties numériques et toutes les lignes à des entrées numériques. Mettez une ligne (et une seule) à l'état haut et les autres à l'état bas. Nous venons de sélectionner la ligne de boutons à vérifier. Maintenant, en lisant les colonnes, lisez l'état de tous les boutons. Quand c'est fait, mettez cette ligne à l'état bas, passez à la suivante et mettez-la à l'état haut et recommencez à lire les colonnes. Faites cela pour que chaque ligne ait été à l'état haut une fois.

Cet exemple considère les connexions suivantes : 3 lignes connectées aux broches (1,2,3) et 3 colonnes connectées aux broches (4,5,6)

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;
```

```
namespace MFConsoleApplication1
{
   public class Program
   {
      static OutputPort[] Rows = new OutputPort[3];
      static InputPort[] Colms = new InputPort[3];

      static bool ReadMatrix(int row, int column)
      {
         bool col_state;

         //Sélectionne une ligne
         Rows[row].Write(true);
         // Lit la colonne
         col_state = Colms[column].Read();
         // Dé-séelctionne la ligne
         Rows[row].Write(false);

         return col_state;
      }

      static void Main()
      {
         // Initialise les lignes en sortie et état bas
         Rows[0] = new OutputPort((Cpu.Pin)1, false);
         Rows[1] = new OutputPort((Cpu.Pin)2, false);
         Rows[2] = new OutputPort((Cpu.Pin)3, false);

         // Initialise les entrées avec un pull-down
         Colms[0] = new InputPort((Cpu.Pin)4, true, Port.ResistorMode.PullDown);
         Colms[1] = new InputPort((Cpu.Pin)5, true, Port.ResistorMode.PullDown);
         Colms[2] = new InputPort((Cpu.Pin)6, true, Port.ResistorMode.PullDown);

         while (true)
         {
            bool state;

            // Lit le bouton de la première ligne, première colonne
            state = ReadMatrix(0, 0);// Ca commence à 0
            Debug.Print("Etat du bouton : " + state.ToString());

            // Lit le bouton de la 4ème ligne, 3ème colonne
            state = ReadMatrix(3, 2);// Ca commence à 0
            Debug.Print("Etat du bouton : " + state.ToString());

            Thread.Sleep(100);
         }
      }
   }
}
```

27.2. Analogiques

Il y a des centaines (milliers) de puces disponibles pour les bus SPI, I2C ou 1-Wire. Certaines lisent de 0 à 5V et d'autres de -10V à +10V. En fait, il y a énormément de solutions pour ajouter des entrées analogiques à votre circuit !

Certaines puces ont des fontions spécifiques. Si nous avons besoin de mesurer une température, nous pouvons connecter un capteur de température à une broche analogique et lire la valeur que nous convertirons en température. C'est une possibilité, mais une meilleure solution consiste à utiliser un capteur numérique de température qui fonctionnent sur I2C, 1-Wire ou SPI. Cela donnera une meilleure précision de la mesure et permettra d'économiser des broches analogiques pour une autre utilisation.

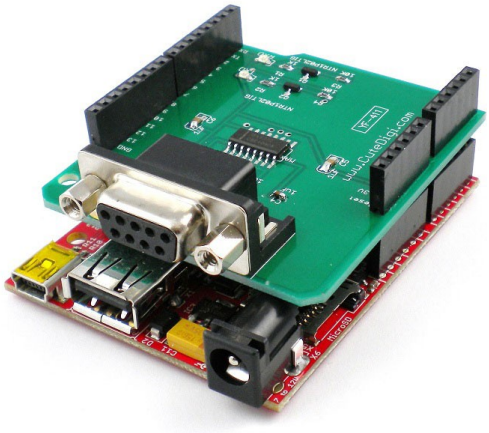
Boutons analogiques

Une astuce pour connecter beaucoup de boutons à une seule broche consiste à utiliser une entrée analogique. Chaque bouton sera connecté à une résistance différente et ainsi la tension à l'entrée de la broche analogique sera différente selon le bouton appuyé.

28. Client USB

Je voudrais préciser tout de suite que cette section est plutôt destinée aux utilisateurs avancés. Si vous venez de débuter avec NETMF, il est préférable d'y revenir plus tard.

28.1. Débogage série (COM)



Par défaut, tous les périphériques NETMF de GHI utilisent USB pour le débogage et le déploiement. Parfois, un développeur peut vouloir utiliser le port USB Client (pas le Host) pour autre chose que le débogage. C'est une possibilité offerte par NETMF à laquelle GHI ajoute d'autres fonctionnalités et le rend plus facile à paramétrer.

Note importante: Vous ne pouvez pas utiliser le port USB client pour à la fois déboguer et faire autre chose. Une fois que vous avez décidé d'utiliser le port USB client pour votre propre utilisation, vous ne pouvez plus l'utiliser pour le débogage. Au passage, GHI autorisait ce fonctionnement (à la fois débogage et utilisation “personnelle”) à travers des interfaces multiples sur le même port USB, mais cela causait une demande trop forte pour le support à cause de mauvaises manipulations et le support multiple a donc été abandonné.

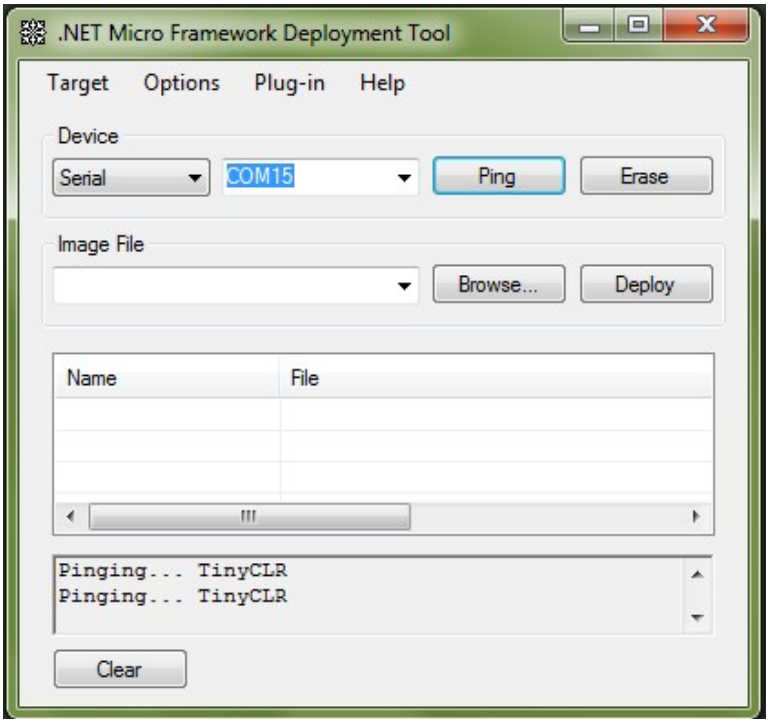
Voyons un exemple d'utilisation du port USB client. Supposons que vous construisiez un périphérique qui lit la température et le taux d'humidité et qui stocke les données lues sur la carte SD. Egalement, ce périphérique peut être configuré pour donner l'heure ou renvoyer des noms de fichiers, etc... Et vous voulez que ce périphérique soit configuré via USB. Donc quand votre périphérique est branché sur sur un port USB, vous allez faire en sorte qu'il se présente comme un port série virtuel. Ainsi, chacun pourra utiliser un programme type Terminal (comme TeraTerm) pour se connecter à votre périphérique et le configurer. C'est là que l'USB client devient pratique. Il ne reste plus qu'à ajouter soit un port série RS232 soit un convertisseur USB<->Série. Le port client USB peut être configuré pour agir en tant que CDC port série virtuel. Mais il y a un piège ! Le piège, c'est que vous devrez vous connecter à votre carte FEZ via le port série pour le débogage et le déploiement d'applications, car le port USB est utilisé par votre application embarquée. La bonne nouvelle malgré tout est que vous n'aurez besoin de cette liaison série que pendant la phase de développement : une fois le programme déployé sur la carte FEZ, le port série n'est plus nécessaire. Par exemple, vous pourriez utiliser l'interface optionnelle RS232 sur votre carte Domino pendant le développement de votre application et vous l'enlevez quand la carte est utilisée en production normalement.
Maintenant, vous avez le port COM1 connecté à votre périphérique et voulez l'utiliser pour

débuguer à la place de l'USB. L'étape suivante consiste donc à configurer votre périphérique pour qu'il le prenne en compte. Cette procédure est spécifique au produit, vous devrez donc vérifier la documentation pour plus de précisions. Par exemple, pour la carte FEZ Domino, il y a un cavalier appelé MODE que vous devez mettre en place pour autoriser le débogage série. Si vous utilisez la carte FEZ Cobra alors la broche LMODE doit être connectée au 3,3V pour obtenir le même résultat. Rappelez-vous également qu'une fois que vous avez connecté une telle broche pour cette utilisation, vous ne pourrez plus l'utiliser pour autres chose. Ainsi, la broche MODE de la FEZ Domino est la même broche qui sert pour la LED et le PWM. Une fois cette broche mise à la terre (via le cavalier) vous ne devriez plus essayer de l'utiliser ou tenter d'allumer la LED car vous risqueriez d'endommager votre carte !

28.2. Préparation

Bon allez, assez de bla-bla, passons à la configuration. J'utilise ici la FEZ Domino avec l'interface RS232 mais vous pouvez utiliser le périphérique de votre choix. Cette interface est connectée au PC via un convertisseur RS232<->USB (je n'ai pas de port série sur mon PC). J'ai placé le cavalier MODE et connecté le câble USB de la Domino sur le PC. Après avoir placé ce cavalier, Windows ne chargera aucun pilote quand vous brancherez le câble USB provenant de la FEZ Domino (vous n'entendrez pas le son classique “Périphérique détecté”). Enfin, nous devons nous assurer que nous pouvons communiquer avec la carte en utilisant MFDeploy. Nous l'avons déjà fait auparavant mais là nous choisirons une connexion Série (et non USB). Bien que le branchement au PC se fasse via USB, la communication se fait via un port série du fait du convertisseur. Pour être exact, il s'agit plutôt d'un port série virtuel.

Donc, ouvrez MFDeploy et choisissez COM et vous aurez une liste des ports disponibles. Sélectionnez celui qui est connecté à votre carte et cliquez sur Ping. Vous devriez recevoir le texte “TinyCLR”. Si ce n'est pas le cas, revenez en arrière pour vérifier votre configuration.



28.3. La souris : la bonne blague

Voici un plan d'enfer ! Faites cette blague à quelqu'un, filmez-le et envoyez-moi la vidéo. Voici comment faire : configurez votre FEZ pour émuler une souris USB et faites bouger cette souris en cercle toutes les X minutes. Je vous promets qu'on aura l'impression qu'il y a un fantôme dans votre machine ! Cerise sur le gâteau : Windows peut gérer plusieurs souris en même temps, donc la “souris” FEZ sera en cachée en arrière-plan.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

public class Program
{
    public static void Main()
    {
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.Di3);
        led.StartBlinking(100, 200);


        // Vérifie l'interface de débogage
        if (Configuration.DebugInterface.GetCurrent() ==
Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("L'interface de débogage en cours
est USB. Cela doit être changé avant de continuer. Consultez le manuel de votre
plateforme pour le changement d'interface de débogage.");

        // Démarre la souris
        USBC_Mouse mouse = USBClientController.StandardDevices.StartMouse();

        // Fait faire des cercles à la souris
        const int ANGLE_STEP_SIZE = 15;
        const int MIN_CIRCLE_DIAMETER = 50;
        const int MAX_CIRCLE_DIAMETER = 200;
        const int CIRCLE_DIAMETER_STEP_SIZE = 1;

        int diameter = MIN_CIRCLE_DIAMETER;
        int diameterIncrease = CIRCLE_DIAMETER_STEP_SIZE;
        int angle = 0;
        int factor;
        Random rnd = new Random();
        int i = 0;

        while (true)
        {
            // on choisit une fréquence aléatoire
            i = rnd.Next(5000) + 5000;//entre 5 et 10 secondes
```

```
            Debug.Print("Delai retenu : " + i + " ms");
            Thread.Sleep(i);
            i = rnd.Next(200) + 100;//on le fait pendant un petit laps de temps
            Debug.Print("Boucle " + i + " fois!");
            while (i-- > 0)
            {
                // Vérifie la connexion au PC
                if (USBClientController.GetState() ==
USBClientController.State.Running)
                {
                    // Note : les coordonnées (X,Y) de la souris sont renvoyées
en tant que modification de la position (mouvement relatif et non pas absolu)
                    factor = diameter * ANGLE_STEP_SIZE * (int)System.Math.PI /
180 / 2;
                    int dx = (-1 * factor * (int)Microsoft.SPOT.Math.Sin(angle) /
1000);
                    int dy = (factor * (int)Microsoft.SPOT.Math.Cos(angle) /
1000);

                    angle += ANGLE_STEP_SIZE;
                    diameter += diameterIncrease;

                    if (diameter >= MAX_CIRCLE_DIAMETER ||
                        diameter <= MIN_CIRCLE_DIAMETER
                        )
                        diameterIncrease *= -1;

                    // On envoie la position de la souris
                    mouse.SendData(dx, dy, 0, USBC_Mouse.Buttons.BUTTON_NONE);
                }

                Thread.Sleep(10);
            }
        }
    }
}
```

28.4. Le clavier

Emuler un clavier est aussi facile que pour une souris. L'exemple suivant créera un clavier USB et enverra “Hello world!” au PC toutes les secondes.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;
```

```
public class Program
{
    public static void Main()
    {
        // Vérifie l'interface de débogage
        if (Configuration.DebugInterface.GetCurrent() ==
Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("L'interface de débogage en cours
est USB. Cela doit être changé avant de continuer. Consultez le manuel de votre
plateforme pour le changement d'interface de débogage.");
        // Démarre le clavier
        USBC_Keyboard kb = USBClientController.StandardDevices.StartKeyboard();
        Debug.Print("Waiting to connect to PC...");
        // Envoie "Hello world!" chaque seconde
        while (true)
        {
            // Vérifie la connexion au PC
            if (USBClientController.GetState() ==
USBClientController.State.Running)
            {
                // On envoie Shift pour le H majuscule
                kb.KeyDown(USBC_Key.LeftShift);
                kb.KeyTap(USBC_Key.H);
                kb.KeyUp(USBC_Key.LeftShift);
                // Maintenant, on envoie "ello world"
                kb.KeyTap(USBC_Key.E);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.O);
                kb.KeyTap(USBC_Key.Space);
                kb.KeyTap(USBC_Key.W);
                kb.KeyTap(USBC_Key.O);
                kb.KeyTap(USBC_Key.R);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.D);
                // Le "!"
                kb.KeyDown(USBC_Key.LeftShift);
                kb.KeyTap(USBC_Key.D1);
                kb.KeyUp(USBC_Key.LeftShift);
                // On envoie la touche Entrée
                kb.KeyTap(USBC_Key.Enter);
            }
            Thread.Sleep(1000);
        }
    }
}
```


28.5. CDC – Port série virtuel

Les ports série sont les interfaces les plus commune, surtout dans le monde “embarqué”. C'est une solution idéale pour les périphériques pour transférer des données entre un PC et un périphérique embarqué (FEZ). Pour combiner la popularité et l'utilité de USB avec la facilité d'utilisation des ports série, nous avons des Ports série virtuels. Pour les applications Windows ou d'autres périphériques, un port série virtuel fonctionne exactement comme un port série classique sauf qu'en réalité il s'agit d'un port USB.

Une chose importante que je tiens à signaler ici est que généralement les pilotes CDC gèrent une transaction par trame. La taille maximum d'une trame en USB est 64 octets et il y a 1000 trames/seconde en USB. Cela veut donc dire que la vitesse maximum pour un driver CDC sera de 64Ko/sec. Je pense que Microsoft a compris les besoins en vitesse de transfert et a fait évoluer cette limitation. La dernière fois que j'ai testé la vitesse de transfert sur ma machine Windows 7, j'ai pu constater des débits d'environ 500Ko/sec.

Le programme suivant crée un un driver USB CDC et envoie “Hello world !” au PC toutes les secondes.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;

public class Program
{
    public static void Main()
    {
        // Vérifie l'interface de débogage
        if (Configuration.DebugInterface.GetCurrent() ==
Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("L'interface de débogage en cours
est USB. Cela doit être changé avant de continuer. Consultez le manuel de votre
plateforme pour le changement d'interface de débogage.");

        // Démarre le CDC
        USBC_CDC cdc = USBClientController.StandardDevices.StartCDC();

        // Envoie "Hello world!" au PC toutes les secondes. (Ajoute un caractère
NewLine au passage)
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes("Hello world!\r\n");
        while (true)
        {
            // Vérifie la connexion au PC
            if (USBClientController.GetState() !=
USBClientController.State.Running)
            {
```

```
                Debug.Print("Attends la connexion au PC...");
            }
            else
            {
                cdc.Write(bytes, 0, bytes.Length);
            }

            Thread.Sleep(1000);
        }
    }
}
```

28.6. Stockage de masse

Une des meilleures fonctionnalités uniques de GHI est le support des MSC (Mass Storage Class), en français : Périphériques de stockage de masse. Cette fonctionnalité autorise l'accès au média connecté depuis l'USB. Voyons cela à travers un exemple. Une application de collecte de données sauvegarde ces données sur une carte SD ou sur une mémoire USB. Quand la collecte est terminée, l'utilisateur peut connecter son périphérique à un PC et celui-ci le détectera comme un périphérique MSC. Il pourra alors transférer les données comme si elles provenaient d'un disque local. Vous pouvez comparer ce fonctionnement à celui d'un lecteur de cartes USB, en fait. On peut même améliorer la chose en le configurant d'abord en CDC pour le paramétrer, puis changer dynamiquement ensuite en MSC pour transférer des données.

Une des questions les plus fréquemment posées au support de GHI est la suivante : “Pourquoi ne puis-je pas accéder à mon média quand il est également utilisé en externe (depuis Windows) ?” On accède à un média à travers la classe PersistentStorage. L'accès à l'objet PersistentStorage peut se faire via le système de fichiers interne **ou** de manière externe via le MSC. D'accord, mais pourquoi pas les deux en même temps ? En fait, cela est dû au fait que les systèmes de fichiers mettent en cache beaucoup de données pour accélérer les temps d'accès. Un accès simultané depuis deux sources risque donc de corrompre le cache, d'où l'impossibilité de cet accès simultané. Notez cependant que vous pouvez très facilement basculer entre le système de fichiers interne et le MSC.

Cet exemple suppose qu'une carte SD est toujours présente dans le support. Il configure votre périphérique (ici une FEZ Domino) pour qu'il apparaise en tant que MSC, donc comme un lecteur de cartes.

```
using System;
using System.IO;
using System.Threading;
```

```
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.Hardware;

namespace USBClient_Example
{
    public class Program
    {
        public static void Main()
        {
            // Vérifie l'interface de dobogage
            if (Configuration.DebugInterface.GetCurrent() ==
Configuration.DebugInterface.Port.USB1)
                throw new InvalidOperationException("L'interface de débogage en
cours est USB. Cela doit être changé avant de continuer. Consultez le manuel de
votre plateforme pour le changement d'interface de débogage.");

            // Démarre le service de stockage de masse (MSC)
            USBC_MassStorage ms =
USBClientController.StandardDevices.StartMassStorage();

            // La carte SD est censée être présente
            PersistentStorage sd;
            try
            {
                sd = new PersistentStorage("SD");
            }
            catch
            {
                throw new Exception("Carte SD non détectée");
            }
            ms.AttachLun(0, sd, " ", " ");


            // Active l'accès hôte
            ms.EnableLun(0);

            Thread.Sleep(Timeout.Infinite);
        }
    }
}
```

28.7. Périphériques personnalisés

Le support par GHI du client USB vous permet même de personnaliser complètement votre périphérique, comme vous le désirez. Cette fonctionnalité requiert cependant une bonne connaissance de l'USB. Si vous ne savez pas ce qu'est un EndPoint ou un Pipe (termes anglais conservés pour la compréhension) alors évitez de créer vos propres périphériques. Il

est également très important que votre périphérique soit configurée correctement la première fois qu'il est connecté à Windows. En effet, ce dernier stockes énormément de données à ce sujet dans la base de registres. Donc si vous changez quoi que ce soit dans la configuration de votre périphérique ensuite, Windows ne pourra peut-être pas voir ces changements car il aura utilisé d'anciennes données puisées dans la base de registres.

Donc en gros : restez éloigné des périphériques personnalisés à moins d'avoir de bonnes raisons de vous y mettre et d'avoir les connaissances USB nécessaires, ainsi que celles utilisées pour le driver USB Windows associé.

Je n'irai pas plus loin sur ce sujet, je voulais juste clarifier pourquoi je ne le faisais pas. Les classes fournies en standard, expliquées auparavant, devraient être largement suffisantes pour couvrir la majorité des besoins de base.

29. Basse consommation

Les systèmes alimentés par batterie doivent limiter leur consommation autant que possible.

Il y a plusieurs façons de diminuer la consommation:

1. Réduire la fréquence du processeur
2. Arrêter le processeur quand le système est inactif (tout en gardant actifs les périphériques et les interruptions)
3. Arrêter certains périphériques
4. Mettre le système en hibernation

Un système NETMF peut supporter une ou plusieurs de ces méthodes. Consultez le manuel utilisateur pour voir ce qui est supporté par votre système. En général, toutes les cartes GHI arrêtent le processeur en cas d'inactivité. Sur la FEZ Rhino, la consommation tombe à 45mA, par exemple. En fait, vous n'avez rien à faire de spécial car dès que le système est inactif, le processeur est automatiquement suspendu/arrêté, tandis que les interruptions et les périphériques restent actifs.

Egalement, tous les périphériques tels que ADC, DAC, PWM, SPI, UART sont désactivés par défaut pour réduire la consommation. Ils sont automatiquement activés dès qu'on les utilise.

Si tout cela ne suffit pas, vous pouvez carrément mettre tout le système en hibernation. Souvenez-vous cependant qu'en hibernation, le système n'est plus actif et que les périphériques sont désactivés. Par exemple, des données arrivant sur l'UART ne réveilleront pas le système, vous perdrez alors ces données. La sortie d'hibernation dépend de la carte utilisée mais en général une ou plusieurs broches spécifiques peuvent être activées pour réveiller le système.

Note: GHI a constaté que la gestion standard de NETMF n'était pas adaptée pour l'hibernation et a donc développé une méthode spécifique pour y parvenir.

Note importante: Quand le système est en hibernation, l'USB ne fonctionne plus. Vous ne pourrez plus accéder au mode pas-à-pas ou même à la carte. Si votre programme met la carte en sommeil alors vous ne pourrez plus charger de nouvelle version. En fait, vous devrez passer par le Bootloader pour tout effacer et sortir la carte d'hibernation !

Note importante: En hibernation, l'horloge système est arrêtée (pas la RTC) donc l'heure NETMF ne sera pas disponible et vous devrez lire l'heure RTC et mettre à jour l'heure NETMF au réveil de la carte.

La sortie d'hibernation n'est possible que pour certains évènements. Consultez la doc de votre carte pour en savoir plus. Un moyen simple pour sortir d'hibernation est d'utiliser l'alarme. Si disponible car par exemple, la carte FEZ Rhino dispose d'une horloge 32Khz pour la RTC mais pas la FEZ Mini.

Suivez ce lien pour plus de détails : http://www.tinyclr.com/compare

Dans l'exemple suivant, je vais initialiser la RTC à une valeur aléatoire (mais correcte) puis faire clignoter une LED. Toutes les 3 secondes, l'alarme sera programmée pour se réveiller dans 10 secondes et mettra ensuite le système en hibernation. La carte sera complètement “morte” (pas de débogage possible) et la LED ne clignotera plus. Quand la carte se réveillera, la LED clignotera à nouveau pendant 3 secondes avant de s'arrêter à nouveau pour 10 secondes.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //Fait clignoter la LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //Règle une heure quelconque
        RealTimeClock.SetTime( new DateTime(2010, 1, 1, 1, 1, 1));
        while (true)
        {
            Thread.Sleep(3000);//Clignotement pendant 3 secondes
            RealTimeClock.SetAlarm(RealTimeClock.GetTime().AddSeconds(10));
            Debug.Print("Hibernation pendant 10 secondes!");
            // Dodo pour 10 secondes
            Power.Hibernate(Power.WakeUpInterrupt.RTCAlarm);

            Debug.Print("Bonjour !");
        }
    }
}
```

Une autre possibilité de réveil consiste à utiliser un port d'interruption. Par contre, vous devrez être prudent dans ce cas car n'importe quelle interruption causera un réveil. Par exemple, le module WiFi de la FEZ Cobra utilise en interne une des broches d'interruptions et cela réveillera donc le système. Vous devrez désactiver le WiFi avant l'hibernation. Ce n'est pas la seule astuce à connaitre ! Regardez (ou essayez) le code suivant. Il ne fonctionne pas !

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
```

```
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //fait clignoter la LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        // Règle la broche d'interruption
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0,false,
                                Port.ResistorMode.PullUp,
                                Port.InterruptMode.InterruptEdgeLow);
        while (true)
        {
            Thread.Sleep(3000);//fait clignoter la LED pendant 3 secondes
            // dodo
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //on se retrouve ici après le réveil
        }
    }
}
```

Pourquoi ça ne marche pas ? Quand vous créez une broche d'interruption (ou d'entrée) les interruptions ne sont activées que si le filtre anti-rebond est activé ou si un gestionnaire d'évènement est installé. Donc, pour que l'exemple précédent fonctionne, il faut activer le filtre anti-rebond. Voici le bon code :

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        // fait clignoter la LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        // paramètre la broche d'interruption avec le filtre activé
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0, true,
                                Port.ResistorMode.PullUp,
                                Port.InterruptMode.InterruptEdgeLow);
        while (true)
        {
            Thread.Sleep(3000);//clignote pendant 3 secondes
```

```
            // dodo
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //on se retrouve ici au réveil
        }
    }
}
```

L'autre solution avec le gestionnaire d'évènements :

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //fait clignoter la LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //paramètre la broche d'interruption
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0, false,
                                    Port.ResistorMode.PullUp,
                                    Port.InterruptMode.InterruptEdgeLow);
        LDR.OnInterrupt += new NativeEventHandler(LDR_OnInterrupt);

        while (true)
        {
            Thread.Sleep(3000);//clignote pendant 3 secondes
            // dodo
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //on se retrouve ici au réveil
        }
    }

    static void LDR_OnInterrupt(uint data1, uint data2, DateTime time)
    {
        // vide pour l'instant
    }
}
```

Notez qu'utiliser un port d'entrée est aussi bien qu'un port d'interruption puisqu'en interne les interruptions ne sont générées que lorsque le filtre anti-rebond est activé. Voici un exemple avec un port d'entrée au lieu d'un port d'interruption.

C'est un exemple pour la carte FEZ Domino (USBizi) qui fera clignoter la LED pendant 3 secondes puis entrera en hibernation. L'appui sur le bouton LDR réveillera la carte.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //fait clignoter la LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //paramètre le bouton LDR
        InputPort LDR = new  InputPort((Cpu.Pin)0, true,
                                Port.ResistorMode.PullUp);
        while (true)
        {
            Thread.Sleep(3000);//clignote pendant 3 secondes
            // dodo
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //on se retrouve ici au réveil
        }
    }
}
```

Enfin, vous pouvez provoquer le réveil avec plusieurs évènements. Par exemple, quand un bouton est appuyé ou que l'alarme se déclenche :

```
Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs |
Power.WakeUpInterrupt.RTCAlarm);
```


30. Watchdog

Note du traducteur : je conserve ici le terme anglais, bien plus connu sous cette forme pour cet usage.

Dans les systèmes embarqués, les matériels fonctionnent généralement en continu sans interaction avec l'extérieur. Du coup, si un dysfonctionnement apparaissait, il serait souhaitable d'avoir une sorte de bouton reset automatique. Le watchdog est ce bouton !

Relancement de l'exécution

Supponsons que vous fabriquiez un distributeur automatique qui envoie l'état de son stock sur le réseau. Si votre code envoie une exception qui n'est pas correctement gérée, alors votre programme s'arrêtera. Qui dit programme arrêté, dit distributeur hors-service. Quelqu'un devra donc se déplacer jusqu'à la machine pour la faire redémarrer. Le mieux est finalement d'utiliser un watchdog

Quand vous activez un watchdog, vous lui donnez un temps au-delà duquel il réinitialisera le système (timeout) et vous remettez à jour le compteur du watchdog périodiquement. Si le système se bloque, le compteur du watchdog atteindra la valeur du timeout et en retour il redémarrera le système.

Note importante: GHI a bien noté que le watchdog intégré dans NETMF ne correspondait pas aux besoins des utilisateurs; du coup, GHI a implémenté sa propre version. N'utilisez donc pas le watchdog de Microsoft.SPOT.Hardware mais plutôt celui de GHIElectronics.NETMF.Hardware.LowLevel. Si les deux espaces de noms doivent être utilisés, vous aurez des ambiguïtés sur le nom à utiliser; dans ce cas, vous devrez spécifier le nom complet du watchdog. Par exemple, au lieu d'utiliser Watchdog.Enable(timeout), vous écrirez GHIElectronics.NETMF.Hardware.LowLevel.Watchdog.Enable(timeout).

Cet exemple montre comment régler le timeout à 5 secondes et crée un thread pour remettre le compteur à zéro toutes les 3 secondes. Si quelque chose venait à mal se passer, le système redémarrerait au bout de 5 secondes.

Note importante: Une fois que vous avez activé le watchdog, vous ne pouvez plus l'arrêter. Vous devez donc remettre régulièrement à zéro son timeout. C'est la procédure pour s'assurer que rien ne viendra perturber accidentellement le bon fonctionnement du watchdog.

```
using System;
using System.Threading;
using GHIElectronics.NETMF.Hardware.LowLevel;

public class Program
{
    public static void Main()
    {
        // Timeout 5 secondes
```

```
        uint timeout = 1000 * 5;

        // Active le Watchdog
        Watchdog.Enable(timeout);

        // Démarre un thread pour le compteur
        WDTCounterReset = new Thread(WDTCounterResetLoop);
        WDTCounterReset.Start();

        // ....
        // votre programme se trouve ici

        // Si quelque chose cloche,
        // le thread s'arrêtera et le système redémarrera !
        Thread.Sleep(Timeout.Infinite);
    }

    static Thread WDTCounterReset;
    static void WDTCounterResetLoop()
    {
        while (true)
        {
            // RAZ du timeout toutes les 3 secondes
            Thread.Sleep(3000);

            Watchdog.ResetCounter();
        }
    }
}
```

A ce stade, vous pourriez vous demander : si le logiciel a planté, comment peut s'exécuter le code du watchdog ? En fait, le watchdog agit au niveau matériel, pas logiciel. Cela signifie que le compteur et la RAZ de ce compteur se font dans le processeur, sans l'aide d'aucun programme.

Limiter le temps imparti à des tâches

On revient à notre distributeur mais cette fois nous voulons gérer un problème potentiel. Comme NETMF n'est pas “temps-réel”, les tâches peuvent prendre plus de temps que prévu pour s'exécuter. Quand une personne utilise le distributeur, celui-ci fera tourner un moteur qui permettra au produit de sortir. Supposons qu'un chronométrage a mesuré qu'il fallait que le moteur tourne pendant 1 seconde pour cela. Maintenant, supposons que le système est en train d'exécuter un autre thread avec la carte SD. Supposons encore que la carte a un problème et qu'il faille 3 secondes pour la lire. Ce délai de 3 secondes pendant que le moteur tourne fera que l'utilisateur aura 3 produits au lieu d'un seul !

Si nous avions utilisé le watchdog avec un timeout de 1 secondes, l'utilisateur n'aurait eu qu'un produit et quand le temps effectif aurait dépassé 1 seconde le système aurait redémarré, ce qui aurait arrêté le moteur.

Voici un petit exemple.

```
// Timeout 1 seconde
uint timeout = 1000;

//....l'utilisateur achète quelque chose

// Active le Watchdog
Watchdog.Enable(timeout);
//fait tourner le moteur
//...
//arrête le moteur
//...
// Nous n'avons plus besoin du watchdog, nous devons donc mettre à jour le
timeout
Watchdog.Enable(Watchdog.MAX_TIMEOUT);
WDTCounterReset.Start();    // Voir dans l'exemple précédent

// ....
// votre programme commence ici
```

Détecter les arrêts dûs au Watchdog

Dans certains cas, vous voudrez savoir si le système a redémarré suite à l'activation du watchdog pour consigner ces infos ou lancer diverses procédures.

Voici comment cela fonctionne

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware.LowLevel;


public class Program
{
    public static void Main()
    {
        // cet indicateur n'est lisible ***QU'UNE SEULE FOIS*** à la mise sous
tension
        if (Watchdog.LastResetCause == Watchdog.ResetCause.WatchdogReset)
        {
            Debug.Print("Redémarrage dû au Watchdog");
        }
        else
        {
            Debug.Print("Appui sur le bouton Reset ou mise sous tension du
système");
        }
    }
}
```

31. Sans-fil

Les connexions sans-fil deviennent très importantes dans notre vie. Certaines nécessitent des vitesses de transfert élevées, d'autres une très basse consommation. Certaines proposent des connexions point-à-point alors que d'autres nécessitent un réseau maillé.

Le plus gros problème quand on crée un périphérique sans-fil est la certification. Non seulement ça doit être fait différemment selon les pays mais en plus c'est très cher. Cela peut coûter pas loin de $50 000 pour une certification. Heureusement, certaines compagnies proposent des modules certifiés. En utilisant de tels modules, vous n'aurez pas besoin de certification ou celle-ci sera largement facilitée.

31.1. Zigbee (802.15.4)

Zigbee a été conçu pour être utilisé dans des systèmes basse-consommation. Plusieurs capteurs peuvent être connectés sur un réseau Zigbee. Il consomme très peu d'énergie mais n'est pas très rapide, cependant.

Une implémentation très courante de Zigbee est fournie par les modules Xbee, de Digi. Il y a plusieurs sortres de modules : basse-consommation ou haute puissance d'émission pour couvrir de très grandes distances (20 km !). Ces modules sont proposés avec des antennes internes ou externes.

L'interface avec ces modules est un simple UART. Grâce à UART, ils peuvent être interfacés avec n'importe quelle carte NETMF. Si vous créez une connexion entre deux modules, elle se fait automatiquement. Si vous voulez vous connecter à plusieurs noeuds, alors vous devrez envoyer quelques commandes supplémentaires pour configurer le réseau. Je vous recommande de commencer par la connexion point à point automatique.

La connexion de Xbee à une carte FEZ Mini ou FEZ Domino se fait très facilement en utilisant l'extension Xbee.

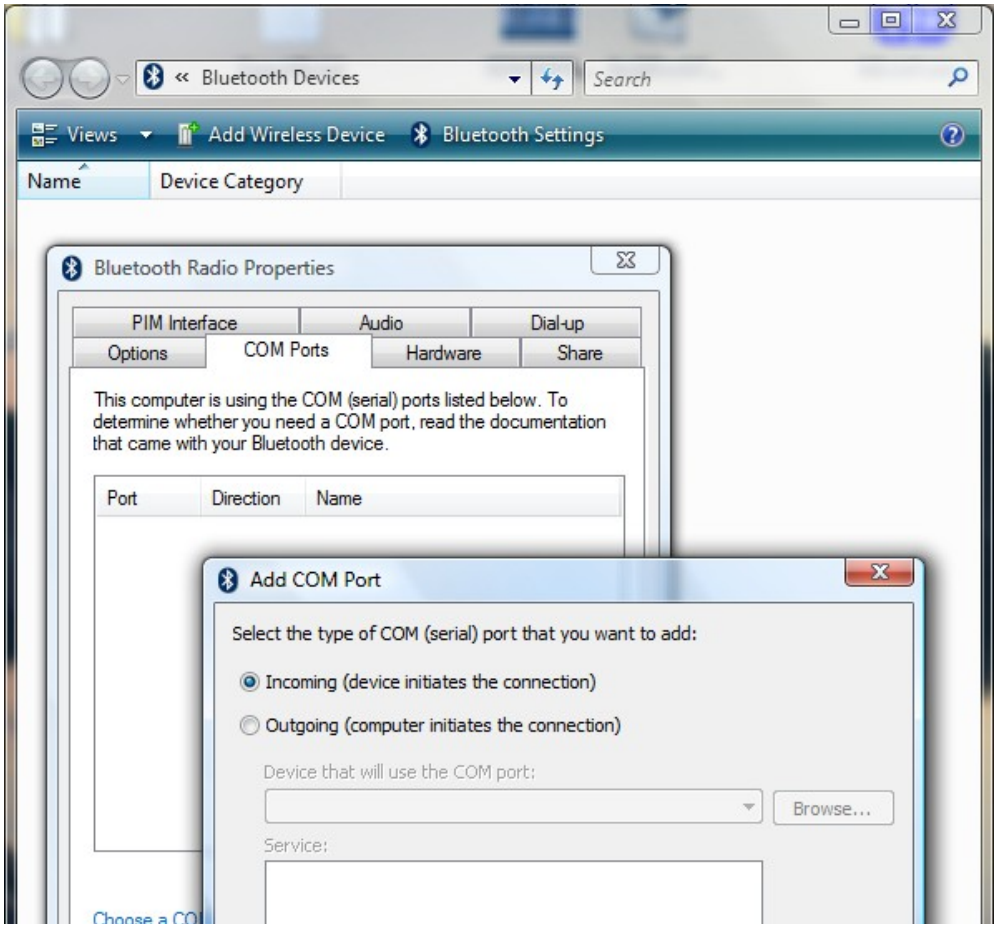
31.2. Bluetooth

Quasiment tous les téléphones portables possèdent de quoi se connecter à des périphériques Bluetooth. La technologie Bluetooth définit de multiples profils pour les connexions. Le profil audio est pratique pour connecter des écouteurs. Le profil SPP (Serial Port Profile) est utile pour établir une communication simulant une connexion série. C'est vraiment très similaire à la connexion Xbee. Si la plupart des téléphones ont le Bluetooth, beaucoup ne supportent pas le SPP et donc la connexion série avec votre portable est probablement impossible.

Sous Windows, on peut créer une connexion Bluetooth en quelques clics :

1. Dans le panneau de configuration, prendre “Périphériques Bluetooth”
2. Cliquez sur “Paramètres Bluetooth”
3. Regardez l'onglet “Ports COM”
4. Si vous avez un port installé, alors SPP est en service sur votre PC
5. Pour ajouter de nouveaux ports, cliquez sur “Ajouter...”
6. Créez un port pour les données entrantes et un autre pour les données sortantes

Windows crée 2 ports, un en entrée et l'autre en sortie. Cela peut perturber l'utilisateur car il ne peut pas utiliser le même port pour envoyer ou recevoir des données.



Côté cartes embarquées, il y a beaucoup de modules qui incluent le logiciel/matériel Bluetooth (ainsi que SPP). Ceux-ci se connectent facilement à un port série de la carte NETMF.

La connexion Bluetooth pour les cartes FEZ Mini ou FEZ Domino peut se faire très facilement avec le composant “Bluetooth interface”.

31.3. Nordic

Nordic Semiconductor a créé sa propre puce sans-fil : NRF24L01. Ces puces basse-consommation utilise la bande des 2,4GHz, qui est disponible dans de nombreux pays. Les puces Nordic autorisent les connexions point à point ou le maillage.

Olimex propose des circuits imprimés pour le NRF24L01. Ces circuits se connectent directement sur la plupart des cartes NETMF de GHI.

Il y a un proet ainsi qu'une vidéo montrant comment connecter deux périphériques utilisant la puce NRF24L01 :

http://www.microframeworkprojects.com/index.php?title=SimpleWireless

32. Objets dans la pile utilisateur

Ce chapitre concerne les allocations de grande taille de mémoire et ne s'applique pas à USBizi.

Les systèmes managés comme NETMF requièrent une gestion très complexe de la mémoire. Pour réduire la charge sur des petits système, la pile NETMF est limitée à 700 Ko. Il n'est pas possible d'utiliser des objets plus grands. A partir de NETMF 4.0, cependant, des gros tampons peuvent être alloués en utilisant une autre pile, séparée, appelée “Pile utilisateur” (Custom heap). Maintenant, vous pouvez créer des gros tampons ou de grosses images. En interne, ces objets ne sont pas dans la pile standard mais dans la pile utilisateur. Cela pose une question, par contre : quelle est la taille mémoire réservée pour la pile standard et la pile utilisateur ?

GHI fournit une API vous permettant de règler la taille de chaque pile : standard et utilisateur

```
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{

   public static void Main()
   {
      // On règle la pile utilisateur à 4 Mo
      if (Configuration.Heap.SetCustomHeapSize(4 * 1024 * 1024))
      {
         // Ca ne prend effet qu'après un redémarrage du système
         PowerState.RebootDevice(false);
      }
      // ...
      // Maintenant vous pouvez utiliser des objets jusqu'à 4Mo
   }
}
```

32.1. Gestion de la pile utilisateur

A la différence de la pile normale, qui est du code complètement managé, il ya plusieurs points à prendre en considération avec la pile utilisateur. Pour que les objets dans la pile utilisateur soient autmatiquement effacés, vous devez repsecter 3 points :

- La référénce à l'objet doit être perdue (c'est la seule condition pour la pile normale)
- Le ramasse-miettes doit être exécuté. Vous devrez certainement le forcer.

- Le système doit être au repos que que Finalize() soit lancée et “dispose” l'objet

Je ne détaillerai pas le ramasse-miettes, dispose ou Finalize. La seule chose que vous devez savoir c'est que les groas objets dans la pile utilisateur ne sont pas supprimés simplement et que vous devez impérativement le “disposer” quand vous avez fini de vous en servir.

Voici comment on “dispose” des gros objets :

```
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{

   public static void Main()
   {
      // Alloue un tampon de 1Mo
      LargeBuffer lb = new LargeBuffer(1024 * 1024);
      // On utilise ce tampon
      lb.Bytes[5] = 123;
      // ....
      // et quand on a terminé, on le "dispose" pour libérer la mémoire qu'il occupait
      lb.Dispose();
   }
}
```

Une meilleure solution consiste à utiliser le mot clé “using”. Avec celui-ci, “dispose” est appelé automatiquement une fois que l'exécution du code entre les accolades est terminée.

```
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{
   public static void Main()
   {
      // Alloue un tampon de 1Mo
      using (LargeBuffer lb = new LargeBuffer(1024 * 1024))
      {
         // utilisation du tampon
         lb.Bytes[5] = 123;
         // ...
      }
      // Dispose () a été appelé automatiquement
      // ...
   }
}
```


32.2. Grandes images

Les images plus grandes que 750Ko/2 sont automatiquement allouées sur la pile utilisateur. Par exemple, une image de 800x480 nécessitera 800x480x4 octets, environ 1,5Mo. Cette image sera donc sur la pile utilisateur et vous devrez la “disposer” vous-même à la fin de son utilisation ou utiliser le mot-clé “using”.

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Presentation.Media;
class Program
{
   public static void Main()
   {
      using(Bitmap largeBitmap = new Bitmap(800,480))
      {
         // On imagine que l'écran est 800x480
         // On y dessine un cercle
         largeBitmap.DrawEllipse(Colors.Green, 100, 100, 10, 10);
         // et quelques autres trucs
         // ...
         largeBitmap.Flush();// On affiche l'image sur l'écran
      }
      // A ce point, largeBitmap.Dispose a été appelée automatiquement
      // ...
   }
}
```

32.3. Le type LargeBuffer

Nous avons déjà utilisé ce type tout à l'heure. Ne l'utilisez que si vous avez réellement besoin d'un tampon de plus de 750Ko. Sur des systèmes embarqués, vous ne devriez pas avoir besoin de tels tampons. Mais, bien que ça ne soit pas recommandé, vous pouvez vous en servir quand même.

```
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{
   public static void Main()
   {
      // Alloue un tampon de 1Mo
      using (LargeBuffer lb = new LargeBuffer(1024 * 1024))
```

```
        {
            // Utilisation du tampon
            lb.Bytes[5] = 123;
            byte b = lb.Bytes[5];
        }
        // Grâce à "using", Dispose() a été appelée automatiquement
        // ...
    }
}
```

33. Penser petit

Beaucoup de développeurs NETMF proviennent du monde PC. Ils sont habitués à écrire du code qui fonctionne bien sur un PC, mais ce même code ne fonctionnera pas de manière optimale sur un logiciel embarqué. Un PC peut tourner à 4GHz et avoir 4Go de mémoire. Les cartes NETMF ont moins de 1% des ressources disponibles sur un PC. Je vais vous parler de différents endroits où vous devrez penser “petit”.

33.1. Utilisation de la mémoire

Avec une quantité de mémoire limitée, vous devez utiliser uniquement ce dont vous avez besoin. Les programmeurs PC ont tendance à créer de gros tampons même pour gérer des petites choses. Les développeurs “embarqué” étudient leurs besoins et allouent uniquement la mémoire nécessaire. Si je lis des données depuis une UART, je peux très bien utiliser un tampon de 100 octets pour la lecture ou même de 1000 octets. Mais si j'analyse mon code, je m'aperçois qu'en fait je ne reçois que 40 octets dans ma boucle. J'envoie certes un plus gros tampon, mais je ne reçois que 40 octets. Alors pourquoi utiliser un tampon de plus de 40 octets ? Eventuellement, je prendrai un petit peu plus, au cas-où, mais en tout cas sûrement pas 1000 octets !

Avec certains pilotes, le système NETMF mémorise beaucoup de données en interne. Par exemple, le système de fichiers, l'UART ou les drivers USB ont des tampons internes, prêts à être utilisé dans le code managé de l'utilisateur. Si nous avons besoin de gérer un fichier de 1Mo, on crée un tampon bien plus petit et on envoie en plusieurs parties. Pour jouer un fichier MP3 de 5Mo, un simple tampon de 100 octet ira lire des petits morceaux du fichier et les passera au décodeur MP3.

33.2. Allocation d'objet

Allouer/libérer des objets est très couteux. Alors n'allouez des objets que si nécessaire. N'oubliez pas non plus que vous programmez un système embarqué; et donc que les objets que vous allez utiliser sont toujours utilisés. Par exemple, vous allez toujours utiliser l'écran LCD ou le SPI.

Prenez par exemple ce code

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;
```

```
namespace MFConsoleApplication1
{
   public class Program
   {
      static void WriteRegister(byte register_num, byte value)
      {
         SPI _spi = new SPI(new SPI.Configuration(Cpu.Pin.GPIO_NONE,false,0,0,false,
                                    true,1000,SPI.SPI_module.SPI1));
         byte[] buffer = new byte[2];
         buffer[0] = register_num;
         buffer[1] = value;
         _spi.Write(buffer);
      }
      public static void Main()
      {
         WriteRegister(5, 100);
      }
   }
}
```

Pour simplement écrire un octet dans un puce SPI, j'ai alloué un objet SPI, un objet SPI.Configuration et un tableau d'octets. Trois objets pour envoyer un seul octet ! C'est envisageable si vous n'avez à le faire que quelques fois au début du programme, mais si vous appelez continuellement la méthode WriteRegister alors ce n'est pas la bonne façon de faire. Pour simplifier, ce code est tellement lent que vous ne pourrez même pas utiliser WriteRegister assez vite. Si en plus elle sert à envoyer une image sur un écran ou un MP3 à un décodeur, c'est encore pire. Cette fonction sera appelée plusieurs centaines de fois par seconde. Un deuxièmle problème est que ces objets sont créés, utilisés et laissés à l'abandon pour que le ramasse-miettes les supprime. Celui-ci va donc devoit entrer en action pour supprimer les objets et cela arrêtera notre programme pendant quelques millisecondes.

Voici une version modifiée de ce code pour appeler la fonction 1000 fois :

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
   public class Program
   {
      static void WriteRegister(byte register_num, byte value)
      {
         SPI _spi = new SPI(new SPI.Configuration(Cpu.Pin.GPIO_NONE,false,0,0,false,
                                    true,1000,SPI.SPI_module.SPI1));
         byte[] buffer = new byte[2];
         buffer[0] = register_num;
         buffer[1] = value;
         _spi.Write(buffer);
```

```
            _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);
            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
            Debug.Print("Durée = " + ms.ToString());
        }
    }
}
```

En exécutant ce code sur la carte FEZ (USBizi), on note que le ramasse-miettes a été appelé 10 fois. Il affiche son activité dans la fenêtre de sortie. Le temps mis pour l'exécution totale du programme est 1911ms, soit près de 2 secondes !

Maintenant, modifions le code comme ci-dessous. Nous avons créé l'objet SPI globalement, ainsi il sera toujours disponible. Nous allouons toujours le tampon dans chaque boucle.

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static SPI _spi = new SPI(new SPI.Configuration(
            Cpu.Pin.GPIO_NONE, false, 0, 0, false,
            true, 1000, SPI.SPI_module.SPI1));

        static void WriteRegister(byte register_num, byte value)
        {
            byte[] buffer = new byte[2];
            buffer[0] = register_num;
            buffer[1] = value;
            _spi.Write(buffer);
            _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);
            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
```

```
            Debug.Print("Durée = " + ms.ToString());
        }
    }
}
```

Dans cet exemple, le ramasse-miettes a été exécuté seulement deux fois et le code n'a duré que 448 millisecondes, moins d'une demi-seconde. Nous avons simplement déplacé une seule ligne de code et il est maintenant 4 fois plus rapide !

Mettons le tampon global lui aussi et voyons le résultat :

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static SPI _spi = new SPI(new SPI.Configuration(
            Cpu.Pin.GPIO_NONE, false, 0, 0, false,
            true, 1000, SPI.SPI_module.SPI1));
        static byte[] buffer = new byte[2];
        static void WriteRegister(byte register_num, byte value)
        {
            buffer[0] = register_num;
            buffer[1] = value;
            _spi.Write(buffer);
            _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);
            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
            Debug.Print("Durée = " + ms.ToString());
        }
    }
}
```

Ca nous donne 368ms et le ramasse-miettes n'a même pas été appelé !

Un test rapide que vous pouvez faire sur votre carte est de suivre l'activité du ramasse-miettes. Sur des cartes avec beaucoup de mémoire comme ChipworkX ça n'aidera pas beaucoup, donc vous devrez toujours vérifier votre code manuellement.

34. Sujets non traités

Certains sujets n'ont pas été traités dans ce livre. En voici une petite liste avec un résumé sur le propos concerné. J'ajouterai peut-être des chapitres pour ces sujets plus tard.

Note du traducteur : j'ai souvent conservé ici les termes anglais car je les trouve plus adaptés que les francisations hasardeuses qu'on peut parfois rencontrer.

34.1. WPF

Windows Presentation Foundation est une manière nouvelle et flexible pour créer des applications avec une interface graphique. Embedded Master et ChipworkX peuvent utiliser WPF mais pas USBizi et FEZ.

34.2. DPWS

Device Profile for Web Services permet à des périphériques d'être automatiquement détectés et utilisés sur un réseau. DPWS nécessite les sockets .NET pour fonctionner et donc ne peut être utilisé qu'avec Embedded Master ou ChipworkX.

34.3. Extended Weak Reference

Extended Weak Reference(EWR) permet aux développeur de sauvegarder quelques données en mémoire non volatile. EWR était principalement utilisé avant l'introduction du système de fichiers dans NETMF

34.4. Sérialisation

La sérialisation est une manière de convertir un objet en une représentation binaire. Un objet Mike fait d'un type Humain peut être sérialisé dans un tableau d'octets et ce tableau peut être transféré ensuite vers une autre machine. Cette machine sait ce qu'est le type Humain mais ne sait rien à propos de Mike. Elle va donc prendre les données reçues pour créer un nouvel objet basé dessus et ainsi aura une copie de l'objet Mike.

34.5. Runtime Loadable Procedures

Runtime Loadable Procedures(RLP) est une fonctionnalité exclusive de GHI qui permet à l'utilisateur d'écrire du code natif C/Assembleur et de l'utiliser ensuite à travers le code

managé (C#). Le code natif est des centaines de fois plus rapide mais est moins facile à gérer, par contre. Des tâches spécifiques, comme des calculs de CRC qui sont très gourmandes en ressources processeur, sont particulièrement adaptées à une telle fonctionnalités. L'application complète est faite en langage managé (C#) mais seule la méthode de calcul du CRC est faite en code natif (C/Assembleur).

A l'heure actuelle, ChipworkX et EMX sont les seules cartes qui supportent RLP.

34.6. Bases de données

Une base de données stocke des données (!) de telle façon qu'elles soient faciles et rapides à extraire ensuite. Rechercher un article ou trier des nombres est très rapide du fait de l'utilisation d'index en interne

A l'heure actuelle, ChipworkX et EMX sont les seules cartes au monde qui supportent les bases de données via SQLite.

34.7. Ecrans tactiles

NETMF supporte les écrans tactiles. Ces écrans forment une bonne combinaison avec les écran TFT. Un développeur peut créer une application graphique avec WPF et l'utilisateur peut la contrôler en utilisant l'écran tactile.

34.8. Evènements

Si vous avez un programme qui reçoie des données depuis un port série, vous devez vérifier continuellement ce port. Il peut ne pas y avoir de données à recevoir, mais comment le savoir sans tester en permanence ? Cela serait plus pratique si nous pouvions être averti que des données sont prêtes à être reçues. Cette notification peut venir d'un “évènement” (Event) qui se produit quand le pilote a reçu des données.

Le même raisonnement s'applique aux ports d'interruption vus au début. Ce livre ne couvre pas la création d'évènements mais nous avons quand même déjà vu comment ils étaient utilisés dans les ports d'interruption ou dans le support de l'USB Hôte.

34.9. Basse consommation

Il y a plusieurs moyens de réduire la consommation d'une carte. Des prochaines révisions de ce livre couvriront ce chapitre plus en détail.

34.10. Hôte USB brut

Nous avons appris comment accéder à différents périphériques USB grâce au support exclusif par GHI de l'Hôte USB. GHI autorise aussi l'écriture de pilotes en code managé pour la plupart des périphériques USB.

L'accès direct à USB est considéré comme une fonctionnalité très avancée et n'est pas dans le propos initial de ce livre.

Voici un projet qui utilise l'hôte USB brut avec une manette XBox

http://www.microframeworkprojects.com/index.php?title=Xbox_Controller

Un autre projet intéressant est le pilote NXT qui permet de contrôler le Mindstorm LEGO NXT depuis C# et Visual Studio :

http://www.microframeworkprojects.com/index.php?title=NXT_Mindstorm

35. Le mot de la fin

Si vous avez trouvé ce livre utile et qu'il vous a permis d'économiser du temps en recherches diverses, alors j'ai accompli ce que je voulais. Je vous remercie très sincèrement d'avoir téléchargé et lu ce livre.

35.1. Lectures complémentaires

Ce livre ne couvre que quelques bases de C# et du .NET Micro Framework. Voici une liste de diverses ressources pour aller plus loin :

- Mon blog est un toujours un bon endroit à visiter
  http://tinyclr.blogspot.com/
- Le site “Micro Framework Project” est également une excellente ressource
  http://www.microframeworkprojects.com/
- Un bon livre électronique gratuit pour aller plus loin avec C#
  http://www.programmersheaven.com/2/CSharpBook
- L'excellent livre de Jens Kuhner sur le .NET Micro Framework
  http://www.apress.com/book/view/9781430223870
- USB complete est un excellent livre sur l'USB
  http://www.lvr.com/usbc.htm
- Wikipedia reste mon favori pour trouver des informations sur un peu n'importe quoi!
  http://en.wikipedia.org/wiki/.NET_Micro_Framework
- La page principale .NET Micro Framework chez Microsoft
  http://www.microsoft.com/netmf

35.2. Responsabilité

Ce livre est gratuit uniquement si vous le téléchargez depuis le site de GHI Electronics. Utilisez-le pour votre propre connaissance et à vos risques et périls. Ni l'auteur, pas plus que GHI Electronics, ne sont responsables de quelque dommage ou perte que ce soit qui pourraient être causées par ce livre ou une information contenue dans ce livre. Il n'y a aucune garantie de validité sur les informations qu'il contient.