NET Micro Framework によるデバイスソフトウェア開発

第 1 部 基本編

目次

はじめに

このドキュメントでは、.NET Micro Framework を用いた組込みデバイス向けアプリケーションの具体的かつ実践的な実装方法や、Visual Studio 2010 の使用方法について説明します。

第 1 部の基礎編では、先ず、.NET Micro Framework 開発を行う際に知っておくべき前提知識と、開発を始めるための準備方法、及び、.NET Micro Framework ライブラリの概要について学習します。次に、.NET Micro Framework でデバイス向けアプリケーションを開発する際によく使う、ユーザーインターフェイス、ファイルシステム、ネットワークの 3 つの機能にそれぞれについて、段階を追ってプログラムを作成し、動作を確認しながら、開発方法の基礎を学んでいきます。

> **註:** .本ドキュメントで取り上げる、NET Micro Framework のバージョンは 4.1 です。使用する Visual Studio は Visual Studio 2010 のみです。それより前のバージョンの Visual Studio では実習はできません。

前提知識

サンプルを動作確認するにあたり、Visual Studio の基本的な操作手順等については割愛しており、当ドキュメントに沿って操作する上での前提知識として必要になります。また、ソースコード例の解説では、紙面の都合もあり、そのコードの特徴的な部分に焦点を当てて説明しています。そのため、基本的な言語文法の知識も前提として必要になります。

サンプルは主に Visual Basic で記載されていますが、C# や C++、Java などのいずれかのオブジェクト指向プログラミング言語の知識をお持ちであれば、サンプルコードの内容は想像がつくでしょう。

具体的な前提知識としては、主に以下の事項に関して必要となります。

- Visual Studio のソリューションやプロジェクトを開くことができる。また、ソリューションとプロジェクトの違いを理解している。
- Visual Studio の統合開発環境で使用するウィンドウ、たとえば、フォーム デザイナーやコード エディターなど、特定の機能のウィンドウを開くことができる。
- ビルドやデバッグの実行方法や、Web アプリケーションの実行方法が分かる。
- 1 つのプロジェクトから、別のプロジェクトを参照するための「参照の追加」(参照設定) を行うことができる。
- Visual Basic、または C#、C++ などのオブジェクト指向言語の基本的な文法が理解できる。特に、クラスライブラリを使用する上での、インスタンスの作成やメソッド呼び出し、プロパティの参照や設定など。

.NET Micro Framework とは

読者の皆さんの周りには、電気で動作する様々な機器が満ち溢れています。情報技術とハードウェア技術が発達した現在、デスクトップ PC やノート PC などのいわゆる PC や携帯電話以外にも、これら

の機器のほぼ全てにコンピュータが入っています。これら PC や携帯電話以外の様々な機器を本ドキュメントでは組込み機器と呼ぶことにします。.NET Micro Framework は、時計やリモコン、おもちゃ家電、自動車の車載パネル、小型の制御機器など、ハードウェアリソースが余り潤沢でない、小型組込み機器向けのファームウェアです。従来この領域のデバイスでは、組込み機器向けの OS を使い、C 言語で制御ロジックをプログラミングするのが一般的でしたが、.NET Micro Framework が搭載されている組込み機器の制御ロジックは、Windows プラットフォームでおなじみの C#でプログラミングが可能です。

.NET Micro Framework では、組込み機器開発に必要なセンサーデバイスを操作する I2C や SPI を使用するためのライブラリをはじめとして、マルチタッチ対応のグラフィックスユーザーインターフェイスや、ネットワークプログラミング、SD カードなどの外部メディア上ファイル操作、など、多彩なライブラリが用意されていて、C#の生産性とあいまって、組込み機器開発を強力に支援します。

.NET Micro Framework が動作するハードウェア仕様を以下に示します。

- CPU： 32bit CPU、MMU 無し可。ARM7、ARM9、Blackfin、SH2、SH4、x86 など
- ROM： 256Kbyte 以上
- RAM： 64Kbyte 以上

.NET Micro Framework は、様々な CPU アーキテクチャ、ハードウェアに移植可能です。移植の際、搭載する機能を取捨選択することが可能です。組込み機器の場合、専用用途の向けのハードウェアであり、ユーザーインターフェイスが数個のボタンしかなかったり、ハードディスクが無いものやネットワーク機能が無い事もあります。無駄な機能を積まないでファームウェアを構成できる点も、小型専用用途向けデバイス向けであるといえるでしょう。勿論、多くの機能を搭載すれば、必要な ROM や RAM の量は増えますが、非常に少ないリソースで動作可能なことがお判りいただけるでしょう。

利用シーン

前節で挙げたように、.NET Micro Framework は、様々な CPU アーキテクチャ、非常に少ないハードウェアリソースで動作可能です。省リソースハードウェア上で、Windows 開発で定評のあるグラフィックスユーザーインターフェイスやネットワークのライブラリを駆使して組込み機器が開発できます。代表的な利用シーンは、

- 時計や多機能リモコン、ネットワークセンサーデバイスをはじめとする小型組込み機器
- 白物家電やインターフォンや制御パネル等の屋内機器
- 車載パネルや各種設備機器のパネル
- 健康器具などの日常用品
- ポータブルナビやカメラ、自転車レコーダ等の携帯端末
- 玩具

など、ネットワーク連携や今時のグラフィックスユーザーインターフェイスが必要な機器や、C 言語では非常に面倒な可変長のデータを扱う機器に最適です。

註: .NET Micro Framework は全ての組込み機器で使えるわけではありません。開発対象の組込み機器への機能や性能要求がハイスペックな場合は、Windows Embedded Comact（Windows CE の後継 OS）や Windows Embedded Standard（Windows XP/7 の組込み向け OS）など、上位の、より最適な OS を選んでください。

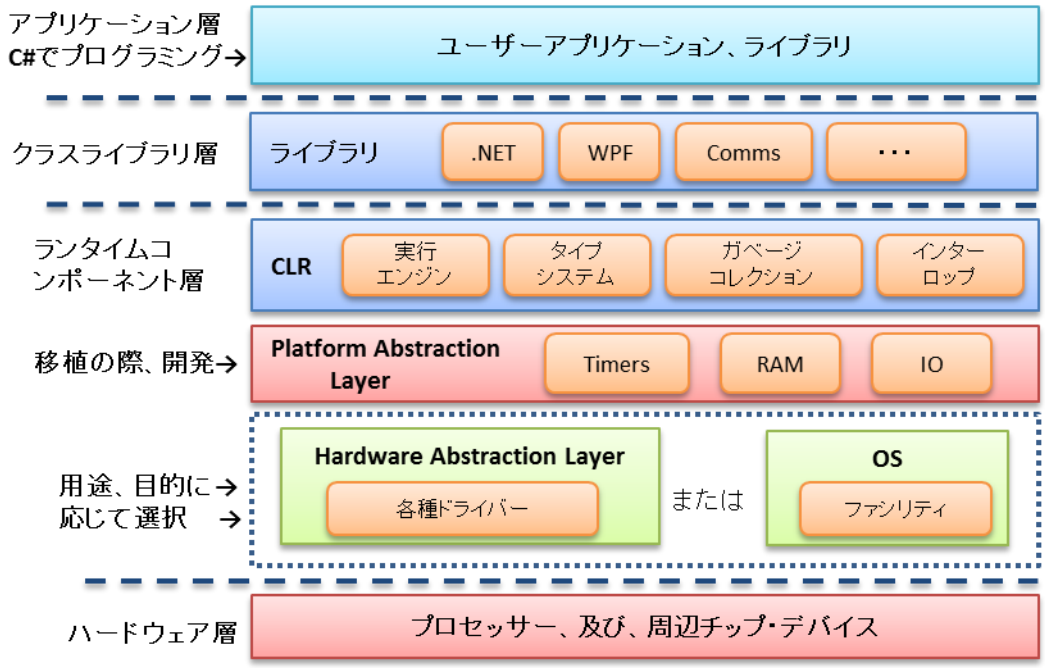
様々なプラットフォームへの移植

.NET Micro Framework はハードウェアに直接移植して使うこともできますが、組込み向け Linux やμITron、T-Kernel など、組込み機器でよく使われる OS 上に移植することも可能です。これにより、既に開発され資産化しているハードウェアや OS、ミドルウェアをそのまま再利用し、グラフィカルユーザーインターフェイスやネットワークプログラミングは、.NET Framework の各種クラスライブラリを使った C#での開発を行うことも可能です。

.NET Micro Framework は、通常の.NET Framework と同様、バーチャルマシンでの実行になります。そのため、残念ながらハードリアルタイムが必要な制御ロジックは組めませんが、RTOS 上に.NET Micro Framework を移植することにより、ハードリアルタイム制御は RTOS 側で、その他は.NET Micro Framework でプログラム開発といった分担処理が可能です。

また、提供ライセンスは、Apache Ver. 2 ライセンスで、無償で各種組込み製品にお使いいただけます。もちろんファームウェアレベルのソースコードも参照と改変が可能です。下図は、.NET Micro Framework のアーキテクチャです。

図 .NET Micro Framework のアーキテクチャ



.NET Micro Framework の移植作業は、以下の URL で公開されている Porting Kit を使って行います。移植方法の詳細は、Porting Kit のドキュメントをご確認ください。

http://download.microsoft.com/download/A/A/4/AA41297E-0889-4B10-A966-EF4F5E457670/PKProductNoLibs.zip

.NET Micro Framework は、これまで非常に限定された開発環境しかなかった組込みソフトウェア開発に、Windows アプリケーション開発並みの開発スタイルを持ち込める画期的なファームウェアと開発環境です。

.NET Micro Framework に関する情報を公開しているサイトのリストを巻末に挙げておきます。

.NET Micro Framework 開発の基本

この節では、.NET Micro Framework のアプリケーション開発を始める前の準備と、開発に必要な基本知識を学習します。説明にしたがって SDK のインストールを行い、アプリケーション開発に関する基礎を習得してください。

.NET Micro Framework SDK のインストール

.NET Micro Framework で開発を行うには、.NET Micro Framework SDK のインストールが必要です。

> **註:** 開発には Visual Studio 2010 が必要です。開発 PC にインストールされていない場合は、有償版を購入するか、Visual Studio 2010 C# Express Edition、もしくは、Visual Studio 2010 評価版を予めインストールしておいてください。

以下の URL から、SDK をダウンロードします。

http://download.microsoft.com/download/0/9/5/0958CB08-E209-4DFD-A945-9DA0FE64B3A4/SDK.zip

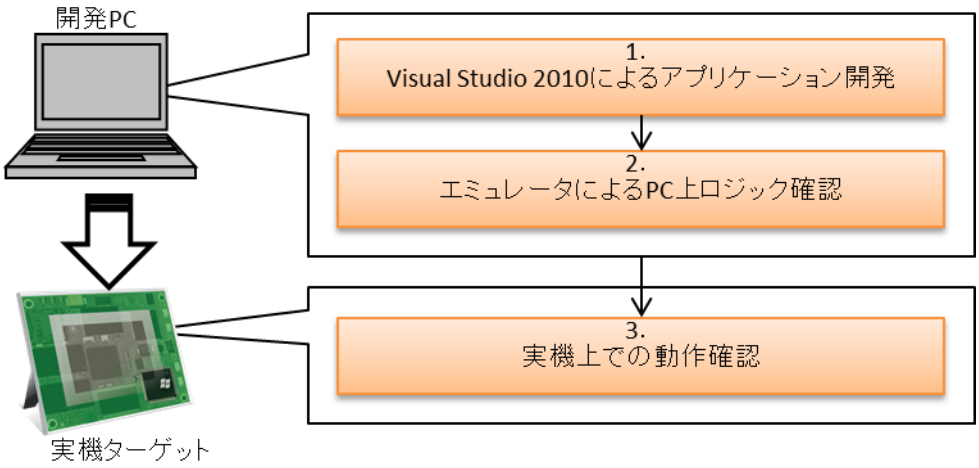
ダウンロードした SDK.zip を解凍し、MicroFrameworkSDK.msi をダブルクリックします。インストールが開始されるので、指示に従って操作を行ってください。

インストールが終われば、.NET Micro Framework の開発、Ready です。

アプリケーション開発、基本フロー

.NET Micro Framework を使った開発の流れを下図に示します。

図 アプリケーション開発フロー



.NET Micro Framework のアプリケーションは、通常の PC アプリケーションと同じく、専用テンプレートを使って新規プロジェクトを作成することからはじめます。開発言語は C#のみです。通常の C#アプリケーションと同じ方法でプログラムをコーディングします。

組込み機器のソフトウェアは、通常の PC 向けアプリケーションとは異なり、実際に動作するのは PC 上ではなく、開発ターゲットの組込み機器上での動作になります。この様な開発スタイルをクロス開発環境と呼びます。

必要なコーディングが終わったら、組込み機器上で動作確認を行う前に、“エミュレータ”を使って、デバッグや実行に制約が少なく、信頼性が高い PC 上で動作を確認します。組込み機器開発では、ソフトウェアとハードウェアが同時並行的に開発されることが多く、プログラミング初期の段階ではまだハードウェアの信頼性が低く、動作に不具合があっても、それがソフトウェアの問題なのか、ハードウェアの問題なのか、切り分けが難しい場合が多々あります。実機のハードウェアに依存しない部分は、なるべく、“エミュレータ”上での動作確認を行いましょう。

PC 上での動作確認が終わったら、実機上での動作確認の流れになります。

> **註:** このドキュメントのスコープ外ではありますが、Visual Studio 2010 には C#のコードメトリクスを計測する機能があります。また、Visual Studio 2010 の最上位エディションである Ultimate には、設計内容をモデル化するための UML エディタが用意されています。品質の高いソフトウェアを開発する為に、これら機能の利用をお勧めします。加えて、昨今の組込み開発では一人きりで開発することは殆ど無く、複数の開発者がチームで開発するのが一般的です。Team Foundation Server を活用した、構成管理、プロジェクト運営も活用してください。

アプリケーションの作成

では、最初にアプリケーションの作成方法を説明します。

先ず、Visual Studio 2010 を起動し、通常の PC アプリケーション開発のときと同様に、メニューの“ファイル”→“新規作成”→”プロジェクト…“を選択します。

SDK をインストールすると、C#のテンプレートに.NET Micro Framework 関連開発用の 4 つのテンプレートが追加されます。それぞれ、

- Class Library
  ソフトウェア部品ライブラリ開発用テンプレート
- Console Application
  LCD 等の表示デバイスを持たないデバイスのアプリケーション開発用テンプレート
- Device Emulator
  独自のエミュレータ開発用テンプレート
- Windows Application
  LCD 等の表示デバイスを持つグラフィカルなユーザーインターフェイスを使うデバイスのアプリケーション開発用テンプレート

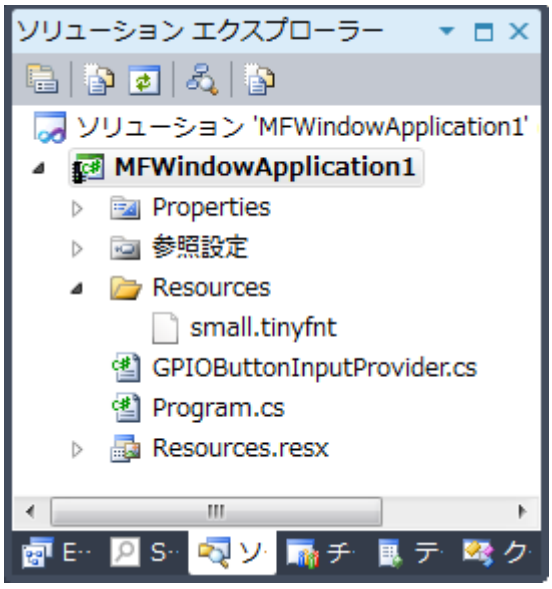
が用意されます。Visual Studio の開発を経験している読者なら、3 番目を除く 3 つのテンプレートの用途は想像がつくと思われるので説明は省きます。前述の通り、小型特定用途向け組込み機器のハードウェアは、通常の PC やスマートフォン等とは異なり、千差万別です。実際の開発では、各デバイスの特徴を反映したエミュレータが必要になります。3 番目のテンプレートは独自のデバイス向けエ

ミュレータ開発用テンプレートです。このテンプレートを使って開発したアプリケーションは、PC 上で 1 番目と 4 番目のテンプレートで作ったプログラムのテスト用環境として動作します。

本ドキュメントでは、4 番目の Windows Application テンプレートを使用します。今後、特に断り無く、「アプリケーションのプロジェクトを作成する」といった場合は、このテンプレートを使用するものとします。

では早速、Windows Application プロジェクトを選択し、適切なプロジェクト名を入力し、ダイアログの"OK"ボタンをクリックしてください。

グラフィカルユーザーインターフェイスを持つ組込み機器デバイス向けのプロジェクト一式が出来上がります。用意されるファイルは以下のような構成になります。



Program.cs は、表示パネルに"Hello World!"を表示し、エミュレータのスイッチ（後ほど説明）を押すと、デバッグコンソールに、押されたボタンの ID を表示するサンプルプログラムが記述されています。

GPIOButtonInputProvider.cs は、ボタンの押下をハンドルする為のサンプルコードが記述されています。

Resources フォルダーの下の small.tinyfnt は、"Hello World!"と表示する為のフォントです。PC やスマートフォントは異なり、組込み機器のグラフィックスユーザーインターフェイスでテキストを表示するには、開発者自身が文字を描画する為のフォントデータを用意する必要があります。.NET Micro Framework では、使いたいフォントデータを組込み機器に搭載する為の機構が用意されており、必要なフォントデータを自由に組込めます。

プロジェクトを作成したら、アプリケーション用のプログラムのコーディングを開始します。通常の C#によるプログラミングとなんら変わりはありません。

> **註:** .NET Micro Framework で提供されるクラスライブラリ群は、実行環境や用途が異なる為、通常の.NET Framework（Full .NET と呼びます）や Windows Embedded Comact に搭載されている.NET Compact Framework とは構成が異なります。同名のクラスがあっても全ての機能がサポ

ートされているとは限らず、また、専用のクラスや異なるメソッド、プロパティ構成の場合もあるので、ご注意ください。

ここでは、何も行わずに、メニューから、“ビルド”→”ソリューションのビルド“を選択、もしくは F6 キーをクリックして、プロジェクトのビルドを行ってください。通常の C#アプリと変わらず、ビルドが行われます。

エミュレータによる実行とデバッグ

ビルドが正常終了すると、エミュレータを使って動作を確認することができます。デバッグの方法や修正方法は、通常の C#プログラムとなんら変わるものではありません。

Program.cs を開き、下のソースコードに示す行にブレークポイントを張ってください。

```
namespace MFWindowApplication1
{
    public class Program : Microsoft.SPOT.Application
    {
        public static void Main()
        {
            Program myApplication = new Program();   ← ここにブレークポイント

            Window mainWindow = myApplication.CreateWindow();
```

メニューから、“デバッグ”→“デバッグ開始”を選択するか、F5 キーをクリックしてください。

通常の C＃プログラムをデバッグする時と同様、ブレークポイントを設定した行で動作が停止します。その状態で、F5 や F10 ,F11 キーをクリックすれば、実行継続、ステップオーバー実行、ステップイン実行、など、通常のデバッグ用機能の利用が可能です。Main メソッドの最後の行まで実行すれば、以下のエミュレータが起動し、アプリケーションが実行されます。

エミュレータには、以下の機能が用意されています。

- 320×240 のタッチ対応 LCD　※タッチはマウスクリックで代替
- 5 つのスイッチ
- 2 枚の SD カードの抜き差しと読み書き　※メニューの“Insert/Eject”
- シリアル COM ポート　※メニューの“Emulator Serial Ports”

これらの機能が、開発ターゲットのデバイスに装備されているなら、そのまま動作確認を行うことができます。

> **註:** 実機のハードウェア構成とエミュレータの搭載機能の対応には注意が必要です。ハードウェアで実装されていない機能は、たとえエミュレータ上で正常実行しても、当然実機上では動きません。

エミュレータの右上の“×”をクリックするか、Visual Studio のデバッグ終了を指示すれば、デバッグを終了できます。

実機による実行とデバッグ

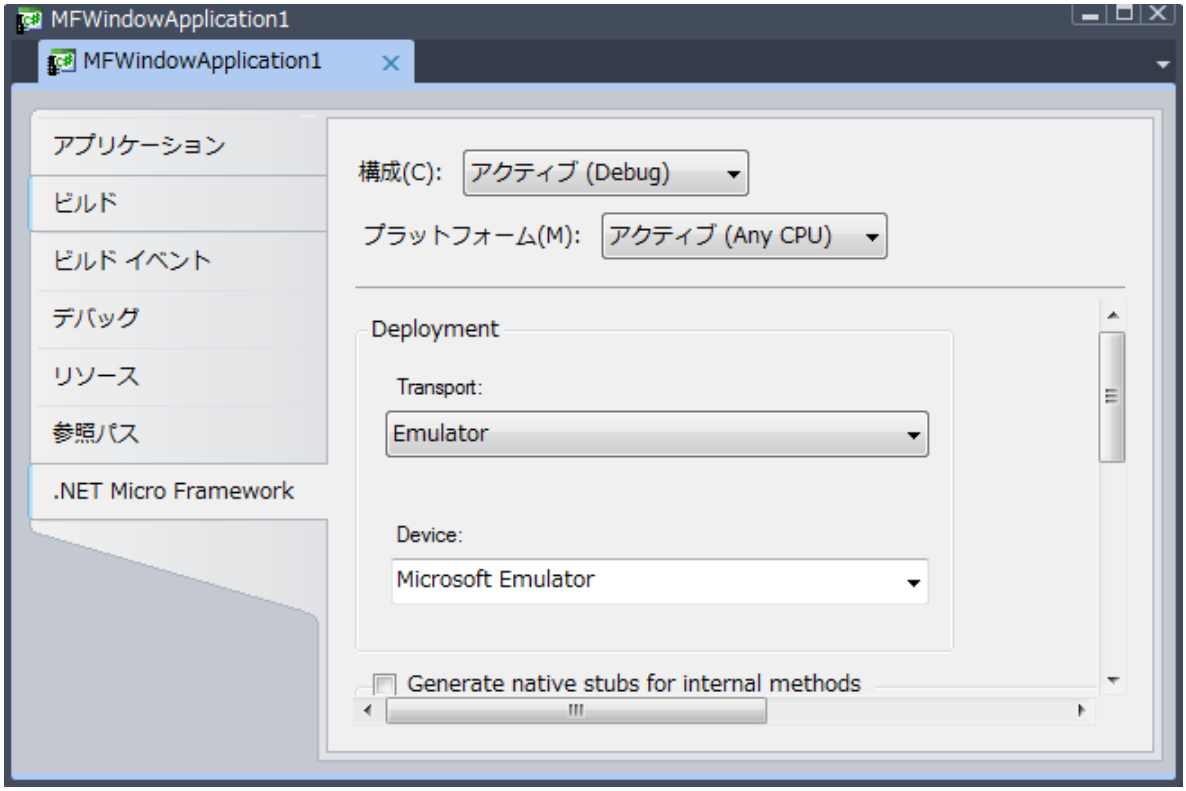
エミュレータ上でプログラムの実行を確認したら、いよいよ実機上での実行です。.NET Micro Framework のプログラムを実行するには、.NET Micro Framework のファームウェアを搭載した実際のデバイスが必要です。本当の組込み機器開発では、メーカーがハードウェアを開発し、.NET Micro Framework を移植することになるわけですが、それでは“ちょっとお試し”は、出来ません。巷で購入できる、.NET Micro Framework 対応のデバイスを先ずはご用意ください。対応デバイスの一覧は、
http://www.microsoft.com/en-us/netmf/hardware/default.aspx
に載っています。

本ドキュメントでは、比較的廉価で入手しやすい、GHI 社製 FezCobra を例に説明を行います。この製品の詳細は、http://tinyclr.jp/ を参照してください。

> **註:** FezCobra を使うにあたっての各種セットアップ手順は、FezCobra 付属の説明書を参照してください。

実機上での実行は、以下の手順で行います。

1. FezCobra を PC に USB を介して接続
2. ソリューションエクスプローラで、プロジェクトを右クリックし、“プロパティ”を選択 “.NET Micro Framework”タブを選択
3. Deployment の Transport を USB に変更
4. Device に接続したデバイス名が表示されるので、それを選択し、変更情報を保存



これで設定は終了です。

後は、エミュレータの時と同様に F5 クリックで、ビルドされたアッセンブリがデバイスに転送されてデプロイされ、実機上での実行が開始されます。

更にエミュレータ上での時と同様、ブレークポイントが設定されている行で一旦実行が停止し、ステップ実行によるソースコードレベルデバッグが可能です。

本ドキュメントの以降の内容は、エミュレータ上で動作可能なものに限定して説明しています。組込みソフトウェア開発の醍醐味は何といっても実機上での動作にあるわけですが、ハードウェアが手元に無くても、PC を使って、自習を続けてください。

開発ターゲットは、前述の設定で、Transport を“Emulator”に選択して変更内容を保存すれば、再びエミュレータに戻ります。

機能概要

この節では、.NET Micro Framework で提供される各種機能のうち、最も頻繁に利用されると思われる機能について説明します。次節以降の自習を始める前に、目を通し、基本事項の知識を習得してください。実際の使い方については、次節以降で説明を行います。

ユーザーインターフェイス

.NET Micro Framework が提供するユーザーインターフェイス機能は、Windows Vista から導入された Windows Presentation Foundation をベースに、小型組込み機器向け用ユーザーインターフェイスクラスライブラリとして構成されています。テキストはもちろん、フルカラーで四角や楕円、多角形、図形のグラデーションでの塗りつぶし、ビットマップイメージや簡単なアニメーションの実装も可能です。描画オブジェクトを適切に配置する為のパネルやウィンドウ、リストボックスなども用意されています。

また、小型組込み機器においてもタッチインターフェイスがデフォルトになった観がありますが、マルチタッチ、ジェスチャー機能もサポートしています。

ファイルシステム

小型組込み機器の場合、外部ストレージとして SD カードをはじめとするメディアを利用する場合が多々あります。.NET Micro Framework にはメディアの抜き差しや、ファイルとして読み書きする為のクラス群が用意されています。これらの機能を使えば、設定情報の保存や、何らかの方法で作成・取得した各種データ、画像情報などの保存が可能です。

ネットワーク

グラフィックスユーザーインターフェイスに並んで最も強力な機能の一つがネットワークです。TCP/IP や UDP/IP のサポートはもちろん、Socket 通信といった低レベルなインターフェイスから、HTTP プロトコルまで.NET Framework とほぼ同様な機能が用意されています。小型組込みデバイスにおいてさえ、ネットワークにつなぐ場合にはセキュリティーが重要な要件になりますが、HTTPS への対応や暗号化機能も用意されています。

最近の IT システム連携の主流として使われている REST 形式の API への対応も HttpWebRequest、HttpWebResponse クラスを用いて用意に対応可能です。加えて、Web サービスでよく使われる XML フォーマットのデータを解析する為の、XML パーサークラスも用意されています。

他にも、ネットワークに接続した時にローカルネット内で、ダイナミックにサービスを発見し、連携する為の DPWS（Device Profile for Web Service）にも対応しています。サービス API を記述した

WSDL ファイルから、必要な Proxy や Stub を生成する為のツールも用意されているので、DPWS 対応機器との連携も容易に開発できます。

その他の機能

小型組込み機器では、ハードウェアに専用のチップやセンサーデバイスを搭載することが一般的です。.NET Micro Framework では、これら周辺デバイスとメインボードをバス接続する時に良く使われる I2C や SPI といったインターフェイスを制御するクラスライブラリも用意されています。このライブラリを利用することにより、センサーデバイスからのデータ取得処理も C#でのプログラミングが可能です。

また、USB ホストや USB クライアントとして振舞う為のクラス群が用意されており、PC にクライアントとして接続した時の処理や、例えば GPS 搭載の USB ドングルを接続した時にホストとして振る舞い、データをやり取りする処理を行うことも可能です。

他にも、.NET Framework で用意されている各種 Stream クラスやリフレクション機能も利用可能など、小型組込み機器制御ソフトウェア開発に必要なクラスライブラリが豊富に揃っています。

以上、.NET Micro Framework が提供する機能の概要を紹介しました。次節から、それぞれのテーマに絞って、開発方法を自習していきます。

ユーザーインターフェイス開発方法

この節では、グラフィックスユーザーインターフェイスの作り方の基礎について、段階的にクラスライブラリを使ってコーディングをすることにより学習していきます。エミュレータで実際の表示や動作を確認しながら進めてください。

.NET Micro Framework のユーザーインターフェイス関連クラスは、WPF(Windows Presentation Foundation)のサブセットです。残念ながら XAML はサポートしていませんが、C#でユーザーインターフェイスクラスのオブジェクトを作成し関連付けや描画を行いながら、画面を組み立てていきます。

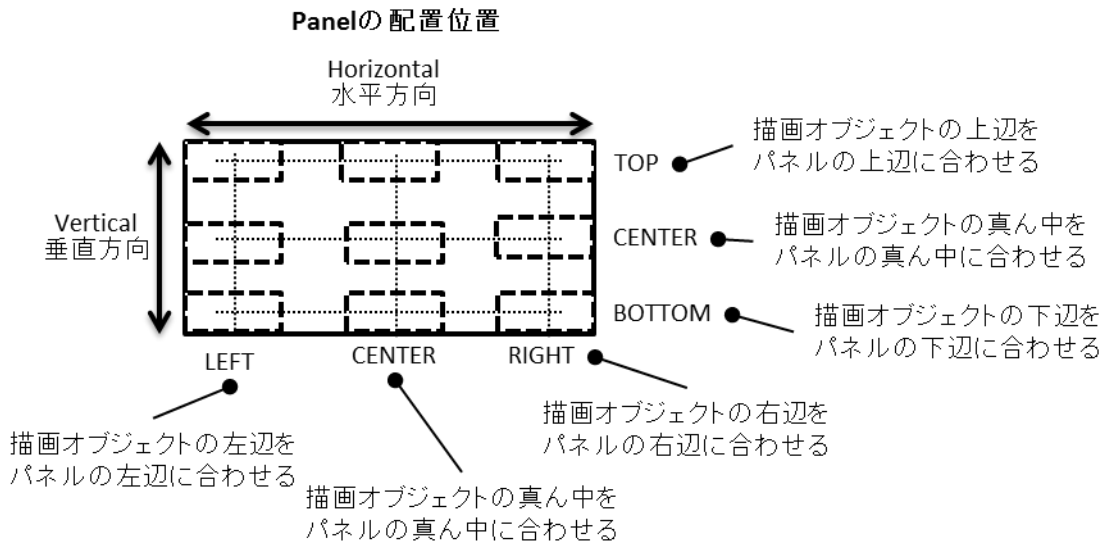
WPF と同様、表示に使用される全てのグラフィックスの要素は UIElement を継承しています。

.NET Micro Framework のグラフィックスユーザーインターフェースは、Window オブジェクトをベースとし、その上にグラフィックス要素を乗せていきます。一般的には、パネルを Window オブジェクトの子要素として設定し、パネルに、テキストや図形、画像、リストボックスを配置します。

パネルのクラスには Panel と StackPanel の二つがあり、このクラスの Children プロパティに描画要素を追加していくと、それぞれの特性に応じて、描画オブジェクトの位置を自動計算し配置します。

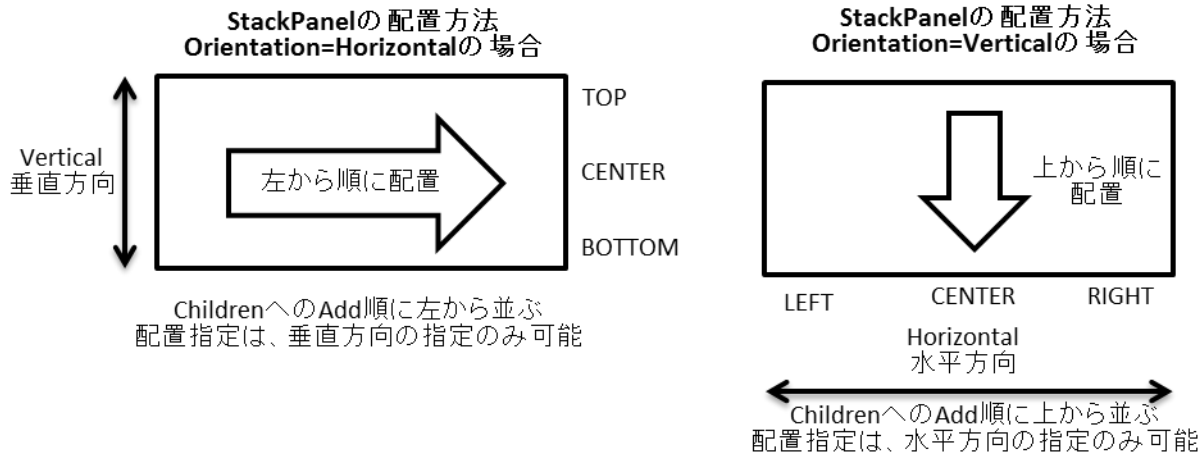
Panel クラスは、以下の様な配置を行うパネルです。

図 Panel の配置位置



StackPanel の場合は、以下の様な配置を行います。描画オブジェクトを順番に並べて配置したい場合に使います。

図 StackPanel 配置位置



図から判る様に、パネルでは、横や縦の位置を指定して図形要素を配置することが出来ません。図形要素を、位置を指定して配置するには、Canvas オブジェクトを使用します。

グラフィックスユーザーインターフェイスのクラス群を使用するには、C#のソースファイルで、以下の名前空間を using 宣言してください。

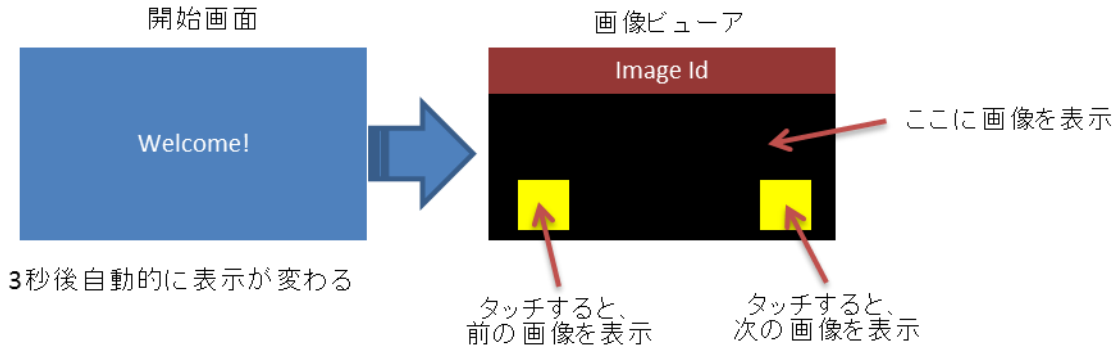
```
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Controls;
using Microsoft.SPOT.Presentation.Shapes;
using Microsoft.SPOT.Presentation.Media;
```

開発するアプリケーション

この節では、以下に説明するアプリケーションを作成していきます。

- デバイス起動後、Welcome メッセージを表示
- 一定時間後、イメージビューワーを表示
- イメージビューワーは、ファイル名と画像を表示する

図 作成するアプリケーションの概要



アプリケーション用の Window を作る

UI 表示のベースは、Window オブジェクトです。.NET Micro Framework で何らかの表示を行うには、先ず、Window オブジェクトを作成し表示する必要があります。元々用意されている Window オブジェクトをそのまま作成して表示することも出来ますが、ここでは、Window クラスを継承して、Welcome メッセージを表示する StartWindow と、ImageViewerWindow の 2 つ Window サブクラスを定義します。

先ず、StartWindow クラスを定義します。プロジェクトに StartWindow.cs という名前で C#ファイルを新規作成し、以下の様なコードを書きます。

StartWindow

```
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Controls;
using Microsoft.SPOT.Presentation.Shapes;
using Microsoft.SPOT.Presentation.Media;

class StartWindow : Window
{
    public StartWindow()
    {
        this.Width = SystemMetrics.ScreenWidth;
        this.Height = SystemMetrics.ScreenHeight;

        BuildContent();

        this.Background = new SolidColorBrush(Colors.Blue);
    }

    public void BuildContent()
    {

    }

    private DispatcherTimer timer;

    public void Show()
    {
        this.Visibility = Visibility.Visible;

        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 3);
        timer.Tick += new EventHandler(timer_Tick);
        timer.Start();
    }

    void timer_Tick(object sender, EventArgs e)
    {
        timer.Stop();
        var viewer = new ImageViewerWindow();
        viewer.Show();
    }
```

```
    }
```

コンストラクタ内では、Window の表示サイズと、背景色を指定しています。表示サイズはデバイスに搭載されている LCD 等の種類によって異なります。SystemMetrics の ScreenWidth、及び、ScreenHight は、デバイスごとの幅と高さのピクセル数を格納しています。これらの値を Window の幅、高さを意味するプロパティ Width と Hight に代入することにより、Window をデバイスの表示サイズに合わせています。Background プロパティは、Window の背景色を指定します。ここでは、Window の背景色を Blue（青）に設定しています。SolidColorBrush クラスは、UI オブジェクトをべた色で塗りつぶす為の Brush クラスです。Brush クラスの詳細は後ほど節を改めて説明します。

コンストラクタ内でコールしている BuildContent（）メソッドは、表示内容を構築する処理を記述する為のメソッドです。後でこのメソッドに処理を追加していきます。

このクラスでは、Show()メソッドをコールするとウィンドウの可視性（Visibility）を可視（Visible）にセットすることにより、このオブジェクトの表示を指示するメソッドです。Window を可視化するとともに、DispatherTimer クラスを起動して 3 秒後に timer_Tick メソッドをコールするようにコーディングされています。

timer_Tick メソッドでは、ImageViewerWindow オブジェクトを生成して表示するロジックが記述されています。

次に、ImageViewerWindow クラスを定義します。内容はクラス名と背景色を黒にしている以外違いはありません。

ImageViewerWindow

```
    using Microsoft.SPOT.Presentation;
    using Microsoft.SPOT.Presentation.Controls;
    using Microsoft.SPOT.Presentation.Shapes;
    using Microsoft.SPOT.Presentation.Media;

    class ImageViewerWindow : Window
    {
        public ImageViewerWindow()
        {
            this.Width = SystemMetrics.ScreenWidth;
            this.Height = SystemMetrics.ScreenHeight;

            BuildContent();

            this.Background = new SolidColorBrush(Colors.Blue);
        }

        public void BuildContent()
        {

        }

       public void Show()
        {
```

```
            this.Visibility = Visibility.Visible;
        }

    }
```

ImageViewerWindow の Show()メソッドは単にこの Window を表示するだけなので、可視性のセットのみのコードになっています。

次に、StartWindow を起動時に表示するコードを記述します。Windows Application プロジェクトを作成した際に自動生成された Program.cs ファイルを開きます。CreateWindow（）メソッド内のロジックを以下のロジックに置き換えます。

```
        public Window CreateWindow()
        {
            var window = new StartWindow();
            window.Show();

            return window;
        }
```

CreateWindow()メソッドは、Progaram クラスの Main()メソッド内でコールされ、表示するアプリケーションウィンドウとして処理されます。

この状態で、Visual Studio の F5 をクリックし、エミュレータでデバッグ実行をしてみてください。青い画面が 3 秒間表示され、その後黒い画面が表示されます。

この様に Window、及び、Window のサブクラスは、複数作成し必要に応じて都度切り替えていくことが可能です。表示した Window がいらなくなったら、Close()メソッドをコールすることにより、表示を終了します。

UI 要素の配置

次に、StartWindow、ImageViewerWindow に表示用の UI 部品を配置していきます。

先ず、StartWindow です。このウィンドウには中央に“Welcome!”というテキストを表示します。

Panel クラスと Text クラスを使います。BuildContent()メソッドに以下のコードを記述します。

```
        public void BuildContent()
        {
            var panel = new Panel();

            Font font = Resources.GetFont(Resources.FontResources.small);
            Text text = new Text(font, "Welcome!");
            text.ForeColor = Colors.White;
            text.VerticalAlignment = VerticalAlignment.Center;
            text.HorizontalAlignment = HorizontalAlignment.Center;
            panel.Children.Add(text);

            this.Child = panel;
        }
```

先ず Panel オブジェクトを一つ作成し、StartWindow が継承している Window クラスの Child プロパティに代入します。

次に、テキスト描画で使うフォントを Resources クラスから取り出します。そして描画する文字列と使用するフォントを使って Text オブジェクトを作成します。Text の ForeColor プロパティはテキストの色を指定するプロパティです。ここでは白を指定しています。また、Text クラスには、VerticalAlligment（縦方向の配置）と HorizontalAlignment（水平方向の配置）という二つのプロパティがあり、Text をパネルに貼り付ける際の配置のポリシーを指定します。このコードでは、どちらも中央に貼り付けるよう指定しています。作成した Text オブジェクトを panel オブジェクトの Children プロパティに追加することによって、貼り付けます。

最後に panel オブジェクトを StartWindow が継承している Window クラスの Child プロパティに代入することにより、ウィンドウに panel を貼り付けて完成です。

ここで F5 をクリックするとエミュレータに以下の様な表示が 3 秒間表示されるようになります。

text の VerticalAlignment、HorizontalAlignment の値を様々に変え、テキストの表示位置の変化をチェックし、Panel クラスの配置機能を確認してください。開発する組込み機器の表示が単純で、単に表示画面の上下左右や中心に描画オブジェクトを配置するだけであれば、Panel クラスの利用で十分です。

テキストや図形の配置機能を持つパネル系のクラスとして、Panel クラスのほかに、StackPanel クラスが用意されています。Panel クラスの場合、Children プロパティの Add()メソッドによる描画オブジェクトの登録の順番に関係なく、それぞれの描画オブジェクトの Alignment の指定によって配置場所が決まるのに対し、StackPanel は、Children の Add()メソッドによる描画オブジェクトの登録順に従って縦方向、または、横方向に順番に配置されていきます。縦方向に描画オブジェクトを配置したい場合は、オブジェクトの生成の際、

```
var voPanel = new StackPanel(Orientation.Vertical);
```

と指定して、StackPanel オブジェクトを生成します。

この時、描画オブジェクトの VerticalAlignment プロパティの指定は無視され、HorizontalAlignment プロパティの指定のみが有効です。

逆に横方向に描画オブジェクトを配置していきたい場合は、

```
var voPanel = new StackPanel(Orientation.Horizontal);
```

で、オブジェクトを生成します。この時は描画オブジェクトの VerticalAlignment 指定のみが有効になります。

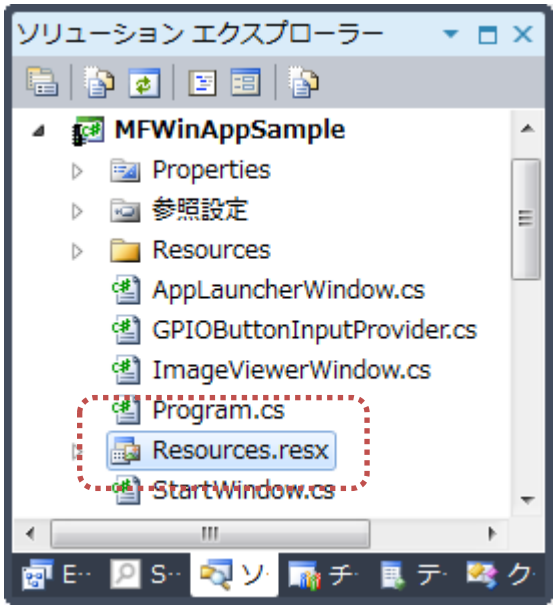
StackPanel は、描画オブジェクトの高さや幅に従って順番に描画オブジェクトを配置する場合に利用できます。

次に ImageViewerWindow の表示要素を組み立てていきます。

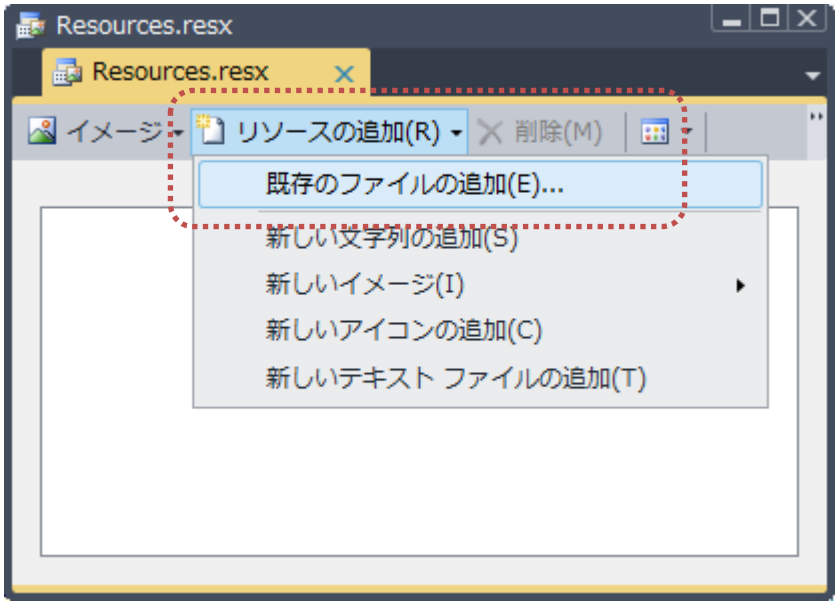
このサンプルでは、リソースに登録された画像情報を表示するので、リソースに画像を登録します。

画像ファイルのリソース登録は、以下の手順で行います。

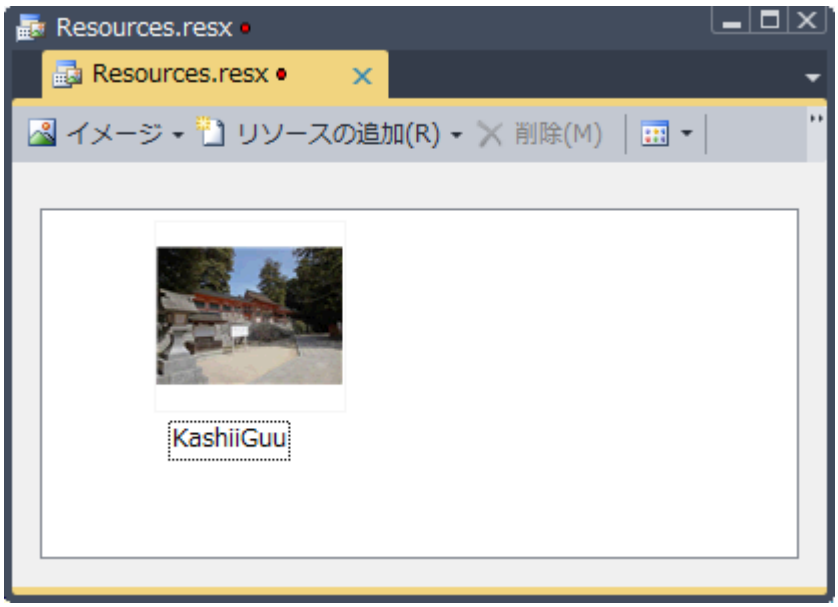
1. ソリューションエクスプローラで Resource.resx をダブルクリックし、リソースエディタを開く



2. リソースエディタのメニューから、“リソースの追加”→“既存ファイル追加”を選択し、追加したい画像ファイルを選択



以上で、下図の様に、リソースに画像が追加されます。画像ファイルは、プロジェクトのディレクトリの下の Resources フォルダーに入れておくと良いでしょう。



リソースに登録された画像情報は、実機での実行時、実機デバイスに配置するアッセンブリに組込まれ、実機デバイス上に転送されます。実際の開発では、アプリケーションの背景やアイコンの画像など、リソースに登録しておくと良いでしょう。

話を元に戻して ImageViewerWindow の表示要素を組み立てます。

ImageViewerWindow は、下図に示すように、StackPanel と Panel、Text、Canvas、Image、Polygon クラスで構成します。先ず、BuildContent()メソッドを以下の様に修正します。

```
        private Canvas imageViewCanvas;
        private void BuildContent()
        {
            var mainPanel = new StackPanel(Orientation. Vertical);
```

```
            var imageTitlePanel = new Panel();
            imageTitlePanel.Height = 20;
            imageTitlePanel.Width = this.Width;
            BuildImageTitlePanel(imageTitlePanel);

            imageViewCanvas = new Canvas();
            imageViewCanvas.Height = this.Height - imageTitlePanel.Height;
            imageViewCanvas.Width = this.Width;
            BuildImageView(imageViewCanvas);

            mainPanel.Children.Add(imageTitlePanel);
            mainPanel.Children.Add(imageViewCanvas);

            this.Child = mainPanel;
        }
```

先ず、垂直方向に並べていく StackPanel を mainPanel という名前で作ります。そして、画像名を表示する上側のイメージタイトル表示用に、imageTitlePanel という名前の Panel オブジェクトと、画像を描画する場所として imageViewCanvas という名前の Canvas オブジェクトを作成し、mainPanel に追加します。mainPanel は垂直方向に並べる指定がされているので、imageTitlePanel と imageViewCanvas は上下に整列します。

次に、imageTitlePanel の内容を構築する BuildImageTitlePanel()を説明します。

```
        private Text imageTitleText;
        private void BuildImageTitlePanel(Panel panel)
        {
            var font = Resources.GetFont(Resources.FontResources.small);
            imageTitleText = new Text(font, "");
            imageTitleText.ForeColor = Colors.White;
            imageTitleText.VerticalAlignment = VerticalAlignment.Center;
            imageTitleText.HorizontalAlignment = HorizontalAlignment.Center;
            panel.Children.Add(imageTitleText);
        }
```

このメソッドでは、画像のタイトルを表示する為のテキストを作成し、Panel の中央に配置しています。テキストの中身は、表示する画像が決まらないと確定しないので、空文字で初期化しています。また、画像を決定するメソッドがテキストを更新できるように、imageTitleText はメンバー変数として宣言してあります。

次に、imageViewCanvas の内容を構築する BuildImageViewCanvas()を説明します。

```
        private Image currentImage;
        private void BuildImageView(Canvas canvas)
        {
            // 画像描画
            var bitmap = Resources.GetBitmap(Resources.BitmapResources.KashiiGuu);
            currentImage = new Image(bitmap);
            DrawImageOnView(currentImage);
            SetImageTitle(Resources.BitmapResources.KashiiGuu.ToString());

            // 画像表示変更用タッチ領域用に四角形と文字列を作成 - 前へ移動
            var prevPart = DrawChangePart("Prev", 180, 15);

            // 画像表示変更用タッチ領域に四角形と文字列を作成 - 後へ移動
```

```
        }

        private void SetImageTitle(string title)
        {
            imageTitleText.TextContent = title;
        }
```

ここでは、画像の表示を DrawImageOnView()メソッドで、複数画像を切り替える（この機能は後で追加します）為の四角形を作成する DrawChangePart()メソッドを 2 回コールしています。SetImageTitle()メソッドは、描画した画像の ID を ImageTitlePanel のテキストに設定するメソッドです。このメソッドでは、先ほどリソースに登録した画像を Resources クラスの GetBitmap()メソッドを使って取り出して、DrawImageOnView()メソッドに渡しています。

次に DrawImageOnView()メソッドです。

```
        private void DrawImageOnView(Bitmap bitmap)
        {
            currentImage = new Image(bitmap);
            currentImage.Width = imageViewCanvas.Width;
            currentImage.Height = imageViewCanvas.Height;
            Canvas.SetTop(currentImage, 0);
            Canvas.SetLeft(currentImage, 0);
            imageViewCanvas.Children.Add(currentImage);
        }
```

画像を表示する為の Image オブジェクトを渡された Bitmap オブジェクトを基に作成し、Canvas オブジェクトに配置しています。

そして Canvas クラスの SetTop()と SetLeft()メソッドを使って、Image オブジェクトのキャンバス上での位置を指定し、canvas の Children プロパティに追加しています。この形式は PC 向けの WPF における描画位置指定の形式と同一です。

キャンバス上の描画位置を指定するメソッドは、SetTop()、SetLeft()の他に SetBottom()、SetRight()の二つが存在し、描画する矩形の指定が可能です。画像以外のテキストや四角形、楕円、多角形など全ての描画オブジェクトのキャンバス上での位置指定はこの形式で行います。

次に、DrawChangePart()です。

```
        private Rectangle DrawChangePart(string feature, int top, int left)
        {
            var rectangle = new Rectangle(30, 30);
            rectangle.Fill = new SolidColorBrush(Colors.Yellow);
            Canvas.SetTop(rectangle, top);
            Canvas.SetLeft(rectangle, left);
            imageViewCanvas.Children.Add(rectangle);
            var font = Resources.GetFont(Resources.FontResources.small);
            var text = new Text(font, feature);
            text.ForeColor = Colors.Blue;
            Canvas.SetTop(text, top + 10);
            Canvas.SetLeft(text, left + 5);
            imageViewCanvas.Children.Add(text);
```

```
            return rectangle;
        }
```

Rectangle オブジェクトを、四角形を幅と高さを指定して生成して、キャンパスに、画像の貼り付けと同様な形式で描画しています。さらにここでは、四角形の上に引数で渡されたテキストを描画しています。

この四角形は、後でタッチイベントをハンドリングする際に使う矩形領域です。

F5 をクリックし実行するとエミュレータ上で、下図の様に表示されます。

表示の上の部分（ImageTitlePanel）は、この Window オブジェクトの背景色である Black になっています。そのままでは Cool ではないので、この部分に色をつけましょう。実は Panel と StackPanel は、描画オブジェクトの配置のみをつかさどるクラスであり、他の UI 向けオブジェクトとは異なり背景色を指定する方法がありません。これら 2 つのクラスのオブジェクトの背景色を指定するには、Border クラスを使います。

BuildContent()メソッドを以下の様に修正します。

```
        private void BuildContent()
        {
            var mainPanel = new StackPanel(Orientation.Vertical);

            var border = new Border();        ← Borderオブジェクト作成
            border.Background = new SolidColorBrush(Colors.Brown);
            border.SetBorderThickness(0);
            var imageTitlePanel = new Panel();
            imageTitlePanel.Height = 20;
            imageTitlePanel.Width = this.Width;
            BuildImageTitlePanel(imageTitlePanel);
```

```
            border.Child = imageTitlePanel;   ←BorderにPanelを登録

            var imageViewCanvas = new Canvas();
            imageViewCanvas.Height = this.Height - imageTitlePanel.Height;
            imageViewCanvas.Width = this.Width;
            BuildImageView(imageViewCanvas);

            mainPanel.Children.Add(border);   ←Borderオブジェクトを登録
            mainPanel.Children.Add(imageViewCanvas);

            this.Child = mainPanel;
        }
```

imageTitlePanel を mainPanel に直接追加するのでなく、一旦 Border オブジェクトに登録します。Border オブジェクトの背景色を指定し、imageTitlePanel を子要素として登録します。出来上がった Border オブジェクトを imageTitlePanel に変わって子要素に加えます。これで無色透明だったパネルに色がつきました。

次節はタッチイベントをハンドリングすることにより画像を変える処理を例として、タッチイベントのハンドリング方法を学びます。後 2 つ画像ファイルをリソースに加え、BuildImageView の先頭部分を以下の様に修正してください。

```
        private Image currentImage;
        private Bitmap[] images = new Bitmap[3];
        private string[] imageNames = new string[3];
        private int currentIndex = 0;
        private void BuildImageView(Canvas canvas)
        {
            images[0] = Resources.GetBitmap(Resources.BitmapResources.KashiiGuu);
            imageNames[0]=Resources.BitmapResources.KashiiGuu.ToString();
            images[1] = Resources.GetBitmap(Resources.BitmapResources.KashiiGate);
            imageNames[1] = Resources.BitmapResources.KashiiGate.ToString();
            images[2] = Resources.GetBitmap(Resources.BitmapResources.Chozu);
            imageNames[2] = Resources.BitmapResources.Chozu.ToString();

            DrawImageOnView(images[currentIndex]);
            SetImageTitle(imageNames[currentIndex]);
```

これで、3 つの画像がアプリケーションに読み込まれ、images[0]に格納された画像が表示されます。実行しても見た目は変わりありません。

イベントハンドリング

.NET Micro Framework では、LCD 等の表示装置に対するタッチ操作をハンドリングするフレームワークが用意されています。

.NET Micro Framework が提供するタッチ操作ハンドリング実装方法は二種類用意されています。一つ目は、UIElement に用意されているイベントにハンドラーを登録して処理する方法です。

UIElement には、タッチ操作が発生した時に発生する、以下の 6 つのイベントが用意されています。

- TouchUp
  タッチが終了した
- TouchDown
  タッチされた
- TouchMove
  タッチされたまま移動
- TouchGestureStart
  ジェスチャー操作開始
- TouchGestureChange
  ジェスチャー移動
- TouchGestureEnd
  ジェスチャー終了

目的と用途に応じて、イベントハンドラーを作成し、これらのイベントに登録することにより、システムからタッチやジェスチャーに関するイベントを受け取ることが出来ます。

二つ目は、描画クラスのサブクラスを定義し、そのサブクラスで、UIElement に定義されている、イベントハンドリング用メソッドを実装する方法です。UIElement には、先にあげたイベント名の頭に“On”をつけたメソッド（OnTouchUp が、TouchUp に相当）が定義されています。これらのメソッドをサブクラスで実装していた場合、タッチやジェスチャーイベントが発生すると対応するメソッドがコールされます。

> **註:** ..NET Micro Framework ではマルチタッチをサポート（ハードウェアの対応必要）しています。一つ目のイベント形式の場合、タッチ毎にイベントが発生しハンドリングが必要です。
>
> 二つ目のメソッド形式の場合は、描画オブジェクトの表示内で同時に発生したタッチイベントはまとめて一回のメソッドコールになります。この場合は、引数でタッチされた複数の座標を取り出すことが可能です。各描画オブジェクトは自身の表示領域に画面上のある座標が含まれるかを判断するメソッドをもっているので、Window や Panel、Canvas など複数の子要素を管理しているクラスが、マルチタッチイベントを受けた際の子要素への制御を行うような処理に向いているといえるでしょう。

ここでは、一つ目のイベント（TouchUp、TouchDown）の使い方を、これまで作成したアプリケーションの、画像表示領域でで作成した四角形のタッチで画像を変更する処理の追加を通して学習します。

タッチ機能を使うには、開発中のプロジェクトに以下のアッセンブリを追加する必要があります。

- 32Bit OS の場合
  C:¥Program Files¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Touch.dll
- 64Bit OS の場合
  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Touch.dll

さらに、アプリケーション開始時に、システムにタッチ機能を使うことを以下のコードで通知します。場所は、Program.cs ファイルの Program.Main メソッドの myApplication.Run メソッドをコールする直前です。

```
            Microsoft.SPOT.Touch.Touch.Initialize(myApplication);

            // Start the application
            myApplication.Run(mainWindow);
        }
```

これで準備完了です。早速始めましょう。

先ず、BuildImageView()メソッド内で、DrawChangePart()メソッドで作成した四角形にタッチイベントハンドラーを登録する処理を追加します。

```
        private void BuildImageView(Canvas canvas)
        {
            // 画像描画
            images[0] = Resources.GetBitmap(Resources.BitmapResources.KashiiGuu);
            imageNames[0]=Resources.BitmapResources.KashiiGuu.ToString();
            images[1] = Resources.GetBitmap(Resources.BitmapResources.KashiiGate);
            imageNames[1] = Resources.BitmapResources.KashiiGate.ToString();
            images[2] = Resources.GetBitmap(Resources.BitmapResources.Chozu);
            imageNames[2] = Resources.BitmapResources.Chozu.ToString();

            DrawImageOnView(images[currentIndex]);
            SetImageTitle(imageNames[currentIndex]);

            // 画像表示変更用タッチ領域用に四角形と文字列を作成 - 前へ移動
            // 画像表示変更用タッチ領域用に四角形と文字列を作成 - 前へ移動
            var prevPart = DrawChangePart("Prev", 180, 15);
            prevPart.TouchDown +=
                new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchDown);
            prevPart.TouchUp +=
                new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchUp);

            // 画像表示変更用タッチ領域に四角形と文字列を作成 - 後へ移動
            var nextPart = DrawChangePart("Next", 180, 275);
            nextPart.TouchDown +=
                new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchDown);
            nextPart.TouchUp +=
                new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchUp);
        }
```

入力にはインテリセンス機能を活用してくださいね。

次に、イベントハンドラー側のメソッドを実装していきます。コードは以下の通りです。

```
    void prevPart_TouchDown(object sender, Microsoft.SPOT.Input.TouchEventArgs e)
    {
        // imagesに格納されている前の画像を表示する。
        // タッチされていることを示す為にprevPartの色を赤に変える
        if (currentIndex == 0)
        {
            currentIndex = images.Length - 1;
```

```
        }
        else
        {
            currentIndex--;
        }
        imageViewCanvas.Children.Remove(currentImage);
        DrawImageOnView(images[currentIndex]);
        var rectangle = sender as Rectangle;
        rectangle.Fill = new SolidColorBrush(Colors.Red);
        e.Handled = true;
    }

    void prevPart_TouchUp(object sender, Microsoft.SPOT.Input.TouchEventArgs e)
    {
        // prevPartの色を元に戻す
        var rectangle = sender as Rectangle;
        rectangle.Fill = new SolidColorBrush(Colors.Yellow);
        e.Handled = true;
    }
```

images[]配列に格納された画像を一つ前に進めます。

タッチが始まった時 prevPart_TouchDown がコールされ、タッチが終わったとき prevPart_TouchUp がコールされます。prevPart_TouchDown では、現在表示中の images[]配列内の場所を示す currentIndex をデクリメント（先頭の場合は最後に移動）し、キャンバスに描画済みの Image オブジェクトを Remove()メソッドを使って削除し、改めて DrawImageOnView()メソッドで新しい Bitmap オブジェクトを書き込みます。

さらにここでは、組込み機器の UI 機能の貧弱さを補う為、四角形の色を変えて現在タッチ中であることを示すロジックを入れています。タッチイベントを受信した UIElement（この場合は prevPart_TouchDown を登録した対象の四角形）は引数の sender で送られてきているので、この引数を Rectangle にキャストして、Fill を赤で更新しています。

イベント引数として渡される e は、GetPosition()メソッドが用意されており、このメソッドを使ってタッチされた位置座標を取得できます。

最後の行で、イベント引数 e の Handled プロパティを true にセットすることにより、このイベントが既に目的を達し、使用済みであることをシステムに教えています。WPF では、描画オブジェクトが複数同じ場所に重なっていることが多々あります。それぞれの描画オブジェクトにハンドラーを登録しておけば、一つのタッチ事象で複数のハンドラーメソッドで処理を行うことが出来ます。一方で、.NET Micro Framework が使われる領域の小型組込み機器はハードウェアの性能が低いことが予想されるので、極力無駄な処理による CPU 処理力浪費を避けなければなりません。

タッチ操作が終わった時にコールされる prevPart_TouchUp()メソッドでは、四角形の色を戻す処理を行っています。

ここまでコーディングが済んだら、エミュレーターで F5 実行してください。

“Prev”というテキストでマーキングされた黄色い矩形をマウスでクリックすると、表示画像が変わります。

ここで、“おや？”と思いませんでしたか？そうです、表示画像の変更は目論見通りですが、画像を変更する二つの矩形が消えてしまいました。“Prev”の矩形があった場所をクリックしても何も起こりません。これは、DrawImageOnView()メソッド内で、画像をキャンバスに描画する際、Children プロパティに Add()メソッドで Image オブジェクトを追加しているのが原因です。Canvas.Children では描画オブジェクトの登録順を保持していて、その順番に従って上書きしていくようになっています。今作成中のアプリケーションの仕様では、キャンバス上で、Image オブジェクトが一番最初に格納されている必要があるのです。DrawImageOnView()メソッドのロジックを以下の様に修正します。

```
        private void DrawImageOnView(Bitmap bitmap)
        {
            currentImage = new Image(bitmap);
            currentImage.Width = imageViewCanvas.Width;
            currentImage.Height = imageViewCanvas.Height;
            Canvas.SetTop(currentImage, 0);
            Canvas.SetLeft(currentImage, 0);
            imageViewCanvas.Children.Insert(0, currentImage);
        }
```

最後の行で、Insert()メソッドを使って、currentImage が imageViewCanvas.Children の先頭に挿入されるように変更しました。

これで、画像変更の際の四角形の消失問題の修正、完了です。

残りの nextPart_TouchDown()、nextPart_TouchUp()の 2 つのメソッドの実装方法は容易に想像がつくかとは思うのですが、念のため、以下に掲載しておきます。

```
    void nextPart_TouchDown(object sender, Microsoft.SPOT.Input.TouchEventArgs e)
    {
        // imagesに格納されている次の画像を表示する。
        // タッチされていることを示す為にnextPartの色を赤に変える
        currentIndex++;
        currentIndex %= images.Length;
        DrawImageOnView(images[currentIndex]);
        var rectangle = sender as Rectangle;
        rectangle.Fill = new SolidColorBrush(Colors.Red);
        e.Handled = true;
    }

    void nextPart_TouchUp(object sender, Microsoft.SPOT.Input.TouchEventArgs e)
    {
        // nextPartの色を元に戻す
        var rectangle = sender as Rectangle;
        rectangle.Fill = new SolidColorBrush(Colors.Yellow);
        e.Handled = true;
    }
```

画像や図形の、もう一つ別の描画方法

テキストや図形、画像オブジェクトは、Panel、StackPanel、Canvas の Children プロパティに追加することによって、描画できることを説明してきましたが、描画オブジェクトを描画する方法は、実はもう一つあります。

参考の為、もう一つの描画方法の概要をここで紹介しておきます。

Window クラスには OnRender()メソッドが仮想メソッドとして定義されており、サブクラスで OnRender()メソッドを実装することで、描画を行うタイミングを捕捉できます。このメソッドでは引数として、DrawingContext クラスのオブジェクトを受け取ることが出来ます。DrawingContext クラスには、DrawImage()や DrawText()、DrawRectangle()、･･･等々、画像（Bitmap）やテキスト、各種図形の描画を指示するメソッド群が用意されています。これらのメソッドは、Panel や Canvas を使った時よりも高速な描画処理が可能です。一旦描画した図形は単なるビットマップ情報になってしまうので、Panel や Canvas を使ったときの様に画面を構成する描画オブジェクトを識別した細かな制御は出来ませんが、ビットマップ主体の高速なアニメーションを実現したい時に向いています。

以上、本節では、ユーザーインターフェイスの実装方法に関する基本を学習してきました。作成したアプリケーションに登場した、オブジェクトのプロパティを変える、四角形を楕円やポリゴンに変える、画像そのものをジェスチャー操作で切り替える等、色々試してみてください。

ネットワークアクセス

.NET Micro Framework の魅力の一つは、ネットワークプログラミングを簡単に行えることです。ネットワークの世界では TCP/IP や HTTP をはじめとする様々なプロトコルが標準化されています。また通信系のプログラミングではデータの送受信やデータを扱う処理において、動的かつ可変長のデータを扱う場面が多々あります。.NET Micro Framework では標準プロトコルを利用する為の様々なクラスを使い、メモリ等の資源管理を必要としない C#でアプリケーションが作成できる為、従来の C/C++と比べて、生産性は格段にあがります。

この節では、実習を通じて、ネットワークアクセスプログラミングの基本を学んでいきます。

Web サーバーからの HTML 読み出し

小型組込み機器をネットワーク上のサービスと連携しようとした場合、接続プロトコルは殆ど HTTP であるといっても過言ではないでしょう。.NET Micro Framework では、Full .NET と同様、HttpWebRequest と HttpWebResponse が用意されていて、この 2 つのクラスを使うことにより、HTTP で提供されているサービスに容易に接続することが可能です。

> **註:** ..NET Micro Framework は、PC やサーバー向けの.NET Framework とは異なり、WCF（Windows Communication Foundation）はサポートされておらず、サービス参照追加による Proxy クラスの自動生成は出来ません。しかし、SOAP や REST API に接続する基本機能は全て揃っています。

先ず、一般の Web サーバーにアクセスし、HTML をテキストデータで取得するプログラムを作成します。新しく Console Application テンプレートで、プロジェクトを"MFAppHttpWebAccess"という名前で作成します。HTTP に関するネットワーク機能を使うため、プロジェクトに以下のアッセンブリを追加します。

- 32Bit OS の場合

  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Native.dll
  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥System.Http.dll
  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥System.IO.dll
- 64Bit OS の場合

  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Native.dll
  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥System.Http.dll
  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥System.IO.dll

プロジェクトに GetHttpContent クラスを新規追加します。新たに作成された GetHttpContent.cs ファイルの先頭に、以下の using 文を追加します。

```
    using System.IO;
    using System.Net;
```

GetHttpContent.cs の中身を以下の様にコーディングします。

```
class GetHttpContent
{
    private Uri requestUrl;
    public GetHttpContent(string url)
    {
        requestUrl = new Uri(url);
    }

    public string GetContent()
    {
        string content = null;
        HttpWebRequest request =
            HttpWebRequest.Create(requestUrl) as HttpWebRequest;
        try
        {
            HttpWebResponse response =
                request.GetResponse() as HttpWebResponse;
            if (response != null)
            {
                Stream resStream = response.GetResponseStream();
                content = new StreamReader(resStream).ReadToEnd();
            }
        }
        catch (Exception ex)
        {
            Debug.Print(ex.Message);
        }
        return content;
    }
}
```

このクラスは、コンストラクタでアクセスする Web の URL を指定し、GetContent()メソッドで指定の URL から HTTP データを取得します。

このコードを見れば判るように、.NET Micro Framework による Web への HTTP アクセスは、PC やサーバー向け.NET Framework の時と全く同じです。HttpWebRequest の Create()メソッドを使用して HttpWebRequest オブジェクトを生成し、そこから HttpWebResponse オブジェクトを取得して、そこから、データを読み出すストリームを取り出して、データを読み出します。

Program.cs ファイルの Main()メソッドを以下の様に修正します。

```
public static void Main()
{
    var getHttpContent = new GetHttpContent("http://msdn.microsoft.com");
    string content = getHttpContent.GetContent();
    Debug.Print(content);
}
```

修正が終わったら、F5 でデバッグ実行して動作を確認してみます。

デバッグ出力ビューに、Web サーバーから取得した HTTP データが表示されます。

> **註:** .デバッグ実行の前に、開発で使っている PC をネットワークに接続しておいてください。エミュレータはプログラム実行時、実際にネットワークにアクセスに行きます。

URL で指定するアドレスをいろいろと変更して、お試しください。

以上、Web サーバーへのアクセスは非常に簡単であることがお判りいただけたでしょう。クラウドや Twitter などの昨今の主流の Web サービスの API の殆どが、REST 形式の API で提供されています。それらの API を利用する場合には送信データの暗号化や、HttpWebRequest オブジェクトのヘッダーへのデータ登録等が必要になりますが、処理の基本は全く同じです。

SOAP で提供される Web サービスの利用も、HttpWebRequest オブジェクトのヘッダー情報の構築や、GetResponse()メソッドをコールする前に、SOAP エンベロープなど、SOAP を利用するのに必要な送信データを組み立て、HttpWebRequest から GetRequestStream()メソッドで取り出したストリームに、送信データを書き込んでから GetResponse()メソッドをコールすることによって、比較的簡単にプログラムを作成することが可能です。

REST API や SOAP API の利用方法は、続編の自習書を参考にしてください。

> **註:** ..NET Micro Framework では、Web Browser は提供されていません。.NET Micro Framework を使うことが想定されるレベルの小型組み込み機器で、通常の Web の内容を表示する必要はまずありません。開発対象が Web の内容を正確に表示する必要があるなら、WEC や WES といった OS を利用することをお勧めします。

デバイスの IP アドレス取得

デバイスをネットワークに接続するアプリケーションでは、機器自身の IP アドレスが必要になる場合があります。自身の IP アドレスを取得するには、以下の二つの名前空間を using 宣言し、

```
using Microsoft.SPOT.Net;
using Microsoft.SPOT.Net.NetworkInformation;
```

以下のコードを記述します。

```
string address = null;
NetworkInterface[] nis = NetworkInterface.GetAllNetworkInterfaces();
if (nis.Length > 0)
{
    address = nis[0].IPAddress;
}
```

ネットワークからの時刻情報取得

インターネット上には、現在時刻を提供するサービスが存在しています。ネットワークに接続された小型組み込み機器デバイスが現在時刻を得る必要がある場合、これらのサービスの利用は非常に便利です。.NET Micro Framework には、時刻提供サービスの中で、http://www.nist.gov から現在時刻を取得するライブラリが提供されています。

デバイスの起動時や、時計合せでこのサービスを使ってシステム時計を正確に設定し、その後システム時計をもとに現在時刻の表示や、Web サービスへの現在時刻送信などを行うとよいでしょう。

この時刻取得ライブラリを、“ユーザーインターフェイス開発”の節で作成した、MFAppWinSample プロジェクトに組み込んで、一日ごとにネットワークから時刻を取得してシステムタイムを更新し、タイトルバーの右辺に時刻を表示する機能を追加しましょう。

まず、以下のアッセンブリをプロジェクトの参照に追加します。

- 32Bit OS の場合

  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Time.dll

- 64Bit OS の場合

  C:¥Program Files (x86)¥Microsoft .NET Micro Framework¥v4.1¥Assemblies¥le¥Microsoft.SPOT.Time.dll

Program.cs ファイルを開き、Program クラスに以下のメソッドを追加します。

```
        private static void UpdateSystemTime()
        {
            var setting = new TimeServiceSettings();
            setting.PrimaryServer = new byte[] { 10, 192, 53, 107 };
            setting.RefreshTime = 86400;

            TimeService.Settings = setting;
            TimeService.Start();
            TimeService.SetTimeZoneOffset(540);
        }
```

この処理は、ネットワーク上の時刻サービスとシステムクロックを一日おきに同期させるロジックです。最後の SetTimeZoneOffset()メソッドで、グリニッジ標準時と日本の時差を指定しています。

このメソッドを Main()メソッド内でコールします。以下の行に、追加してください。

```
            Microsoft.SPOT.Touch.Touch.Initialize(myApplication);

            UpdateSystemTime();

            // Start the application
            myApplication.Run(mainWindow);
        }
```

これでネットワーク上の時刻サービスとシステムクロックの同期処理は完成です。

次に画像表示用 Window に時刻表示を加えます。ImageViewerWindow クラスに BuildImageTitlePanel()メソッドに、時刻表示を行うための Text オブジェクト処理を追加します。

```
        private Text imageTitleText;
        private Text clockText;
        private void BuildImageTitlePanel(Panel panel)
        {
            var font = Resources.GetFont(Resources.FontResources.small);
            imageTitleText = new Text(font, "");
            imageTitleText.ForeColor = Colors.White;
            imageTitleText.VerticalAlignment = VerticalAlignment.Center;
            imageTitleText.HorizontalAlignment = HorizontalAlignment.Center;
            panel.Children.Add(imageTitleText);

            clockText = new Text(font, DateTime.Now.ToString("hh:mm:ss"));
            clockText.ForeColor = Colors.White;
            clockText.VerticalAlignment = VerticalAlignment.Center;
            clockText.HorizontalAlignment = HorizontalAlignment.Right;
            panel.Children.Add(clockText);
        }
```

Text クラスの clockText オブジェクトを、時刻表示用にクラスのメンバー変数として加えます。現在時刻を初期値として設定します。

次にコンストラクタを以下のように修正します。

```
        public ImageViewerWindow()
        {
            this.Width = SystemMetrics.ScreenWidth;
            this.Height = SystemMetrics.ScreenHeight;

            BuildContent();

            this.Background = new SolidColorBrush(Colors.Black);

            var timer = new DispatcherTimer();
            timer.Interval = new TimeSpan(0, 0, 1);
            timer.Tick += new EventHandler(timer_Tick);
            timer.Start();
        }
```

clockText 変数を一秒ごとに現在時刻で表示するための、タイマーを加えます。

timer_Tick()は、以下のように実装します。

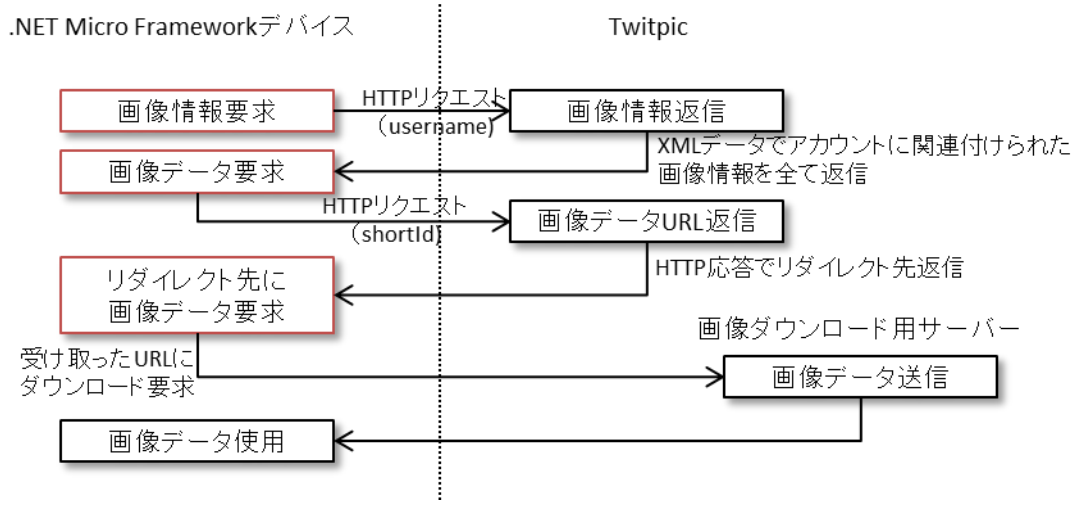
```
        void timer_Tick(object sender, EventArgs e)
        {
            clockText.TextContent = DateTime.Now.ToString("hh:mm:ss");
        }
```

これで、修正は終わりです。皆さんの身の回りを見渡しても判るように、表示デバイスを持つ小型組み込み機器は、大抵時刻を表示しているものです。時刻表示は基本的な技であり、応用が利くので確実にものにしてください。

HTTP による Web サービスアクセス

本節では、これまで作成してきたアプリケーションに、ネットワークからダウンロードした画像を表示する機能の追加を通じて、HTTP による Web サービスへのアクセス方法を学習します。

画像は、比較的ポピュラーでアクセスが容易な Twitpic からダウンロードします。Twitpic は Twitter のアカウントを使って、画像をアップロード、公開できる Web サイトです。Twitpic からの画像ダウンロードに関する処理の流れは、以下の通りです。



註: .本ドキュメントで使用している Twitpic の API は、2011 年 3 月 29 日現在のものです。この API は時とともに変わる可能性がありますので、正しく動かない場合は、http://twitpic.com で最新の情報を確認してください。

先ず Twitpic にアクセスして画像をダウンロードする処理を作成します。

Twitpic は、http://api.twitpic.com/2/users/show.xml?username=*twitterAccount* にアクセスすると *twitterAccount* で指定された Twitter アカウントで登録された画像情報を XML 形式で送り返してきます。※ブラウザで確かめてください。

新しく、“TwitpicImageWebClient”という名前で、プロジェクトにクラスを追加し、この処理を実装します。

クラスのソースコードファイルの先頭に、以下の名前空間の using 宣言を加えます。

```
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Xml;
using Microsoft.SPOT.Hardware;
```

Twitpic から取得した画像情報を格納する為のクラス、“TwitpicImageProfile”を追加します。

```
class TwitpicImageProfile
{
    public string Id { get; set; }          // 画像Id
    public string ShortId { get; set; }     // 画像ショートId
    public string Format { get; set; }      // 画像フォーマット
    public int Width { get; set; }          // 画像幅
    public int Height { get; set; }         // 画像高さ
    public int Size { get; set; }           // データ長
    public bool IsDownloaded { get; set; }  // ダウンロード済み？
    public byte[] imageByte { get; set; }   // 画層データ
}
```

プロパティの Id から Size までは、ブラウザで先に挙げた URL にアクセスして表示される XML の要素に対応しています。IsDownload は、処理時に画像を既にダウンロードしたかどうかを保持します。imageByte は、ダウンロードした画像を保持するプロパティです。この二つは後で加える機能で使います。

次に、TwitterImageWebClient のコンストラクタとメンバー変数を追加します。

```
class TwitpicImageWebClient
{
    private const string twitpicRestAPI=
        "http://api.twitpic.com/2/users/show.xml?username=";
    public TwitpicImageWebClient()
    {
    }
```

twitpicRestAPI 変数は、先ほどの URL の固定部を保持しています。

このクラスに、指定したアカウントに登録されている画像情報リストを取得するメソッドを追加します。

```
        public TwitpicImageProfile[] GetImageProfiles(string account)
        {
            TwitpicImageProfile[] imageProfiles = null;
            if (Proxy != null)
            {
                HttpWebRequest.DefaultWebProxy = Proxy;
            }
            Uri requestUri = new Uri(twitpicRestAPI + account);
            HttpWebRequest request = HttpWebRequest.Create(requestUri)
                as HttpWebRequest;
            try
            {
                HttpWebResponse response = request.GetResponse()
                    as HttpWebResponse;
                imageProfiles = ResolveImageProfiles(response);
            }
            catch (Exception ex)
            {
                Debug.Print(ex.Message);
            }

            return imageProfiles;
        }
```

Twitpic のサイトから登録画像情報を取得する処理は非常に簡単です。既に学習した HttpWebRequest と HttpWebResponse クラスを使って、Twitip API の Url にアクセスします。

Response から取り出したストリームを通じて取得できるのは、ネットワーク上でのデータ交換で標準的な XML 形式のデータです。受信した XML データから情報を取り出す為に ResolveImageProfiles メソッドをコールしています。このメソッドのコードは以下の通りです。

```
    private TwitpicImageProfile[]
        ResolveImageProfiles(HttpWebResponse response)
    {
        ArrayList profiles = new ArrayList();
        XmlReader reader = XmlReader.Create(response.GetResponseStream());

        bool isInImages = false;
        TwitpicImageProfile profile = null;
        while (reader.Read())
        {
            switch (reader.NodeType)
            {
            case XmlNodeType.Element:
                if (reader.Name.CompareTo("images") == 0)
                {
                    isInImages = true;
                }
                else
                {
                    if (isInImages)
                    {
```

```
                    if (reader.Name.CompareTo("image") == 0)
                    {
                        profile = new TwitpicImageProfile();
                    }
                    else
                    {
                        if (profile != null)
                        {
                            if (reader.Name.CompareTo("short_id") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
                                {
                                    profile.ShortId
                                        = reader.Value.ToString();
                                }
                            }
                            else if (reader.Name.CompareTo("id") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
                                {
                                    profile.Id
                                        = reader.Value.ToString();
                                }
                            }
                            else if (reader.Name.CompareTo("width") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
                                {
                                    profile.Width
                                        = int.Parse(reader.Value.ToString());
                                }
                            }
                            else if (reader.Name.CompareTo("height") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
                                {
                                    profile.Height
                                        = int.Parse(reader.Value.ToString());
                                }
                            }
                            else if (reader.Name.CompareTo("size") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
                                {
                                    profile.Size
                                        = int.Parse(reader.Value.ToString());
                                }
                            }
                            else if (reader.Name.CompareTo("type") == 0)
                            {
                                reader.Read();
                                if (reader.NodeType == XmlNodeType.Text)
```

```
                                {
                                    profile.Format = reader.Value.ToString();
                                }
                            }
                        }
                    }
                }
                break;
            case XmlNodeType.EndElement:
                if (profile != null)
                {
                    if (reader.Name.CompareTo("image") == 0)
                    {
                        profiles.Add(profile);
                        profile = null;
                    }
                }
                else
                {
                    if (isInImages)
                    {
                        if (reader.Name.CompareTo("images") == 0)
                        {
                            isInImages = false;
                        }
                    }
                }
                break;
            }
        }

        TwitpicImageProfile[] imageProfiles
            = new TwitpicImageProfile[profiles.Count];
        for (int i = 0; i < profiles.Count; i++)
        {
            imageProfiles[i] = profiles[i] as TwitpicImageProfile;
        }

        return imageProfiles;
    }
```

XML データ処理

非常に長いメソッドですが、基本は、

1. XML テキストを解析する為の XmlReader オブジェクトを作成
2. XmlReader で XML を解析しながら、画像情報を抽出して TwitterImageProfile オブジェクト作成 while(reader.Read())ブロック
3. 出来上がった画像情報リストを配列形式に格納しなおしてリターン

です。.NET Micro Framework では、XML データの構造を解析しながら要素を取り出す高速なパーサーが用意され、XmlReader クラスとして提供されています。XmlReader は、Read メソッドをコール

するごとに XML テキストの構成要素であるタグ（<>で囲まれたテキスト）を一つづつ取り出していきます。XmlReader オブジェクトの NodeType プロパティには、直前の Read()メソッドをコールした際に解析したノードの種類が格納されています。ノードの種類には、開始タグ（例えば<image>）、終了タグ（例えば</image>）、テキスト（例えば<id>…</id>にはさまれた文字列）などがあります。XML テキストはタグによる木構造であり、Read()をコールするごとに、

1. 現在位置（親）の開始タグ
2. 子要素の開始タグ
3. 子要素のテキスト
4. 子要素の終了タグ
5. 子要素が複数ある場合は上の 3 つの繰り返し
6. 親の終了タグ

の順番で XML データの内容を確認していけるので、タグの名前を確認し、そのタグのテキストを取り出していけば、XML テキストからデータを取り出すことが出来ます。ブラウザで XML データを表示しながら、ソースコードと見比べて処理の流れを確認してください。

最後に ArrayList から配列に変換していますが、小型組込み機器の場合、メモリーリソースはあまり潤沢ではありません。配列の利用と比較して ArrayList はより多くのメモリーリソースを消費します。より多くのメモリを消費すれば、パフォーマンスの劣化や他のスレッドの処理でメモリ枯渇を招いてしまいます。登録された画像の数は、XmlReader を使って最後まで解析しないと得られないので、仕方なく ArrayList を利用しましたが、使い終わったらよりメモリ消費の少ない配列に入れ換えて、無駄なメモリを使わないようにしています。

PC アプリケーションと同じ C#が使えるからといって、小型組込み機器のメモリ量が増えるわけではないので、省メモリなプログラミングを心がけてください。

以上で、Twitpic に登録された全ての画像の識別子情報やサイズ情報を取得することが出来ました。

画像データのダウンロード

次に、取得した画像情報を元に実際の画像データをダウンロードする処理を作成していきます。

TwitpicImageWebClient に、以下の 2 つのメソッドを加えます。

```
        public bool DownloadImage(TwitpicImageProfile imageProfile)
        {
            var webClient = new ImageWebClient(twitpicServer, twitpicPort);
            try
            {
                string format;
                imageProfile.imageByte =
                    webClient.DownloadImgae(
                    "/show/full/" + imageProfile.ShortId,
                    out format);
                if (imageProfile.imageByte != null)
                {
                    imageProfile.Format = format;
                    imageProfile.IsDownloaded = true;
                }
```

```
            }
            catch (Exception ex)
            {
                Debug.Print(ex.Message);
            }
            return imageProfile.IsDownloaded;
        }

        public int DownloadImages(TwitpicImageProfile[] imageProfiles)
        {
            int downloadedImages = 0;
            for (int i = 0; i < imageProfiles.Length; i++) {
                if (DownloadImage(imageProfiles[i]))
                {
                    downloadedImages++;
                }
            }
            return downloadedImages;
        }
```

DownloadImage()メソッドは、GetImageProfiles()メソッドで取得した画像情報の一つづつに対して、画像をダウンロードを行います。

DownloadImages()メソッドは、DownloadImage()メソッドを使って GetImageProfiles()メソッドで取得した全ての画像をダウンロードするメソッドです。

先ず、画像を 1 つだけダウンロードする DownloadImage()メソッドについて説明します。ここでは、ソースコードの見通しを良くする為に、ImageWebClinet という別のクラスに画像データダウンロード処理を担わせ、ここでは単に、ダウンロードした画像データを下に TwitpicImageProfile オブジェクトの ImageByte、Format、IsDownloaded プロパティを更新しています。

ネットワーク環境下での通信はデバイス内に閉じた環境とは異なり、常にベストエフォートな環境です。通信状況によって画像を正常にダウンロードできない場合も想定して画像データを正しくダウンロードした時だけ、IsDownload を true にしています。

一方、DownloadImages()メソッドでは、GetImageProfiles()メソッドで取得した全ての TwitpicImageProfile オブジェクトに対して、DownloadImage()メソッドを利用して画像データのダウンロードを試みます。ダウンロードが成功した数をカウントし、コール側に返しています。

さて、実際に、Web サーバーから画像をダウンロードする ImageWebClient ですが、このクラスの処理は非常に長く複雑です。きちんと解説すると、基礎編のレベルを大幅に超えてしまうので、ここでは、基礎のレベルでのポイントを紹介し、コードの掲載のみに留めることとします。

PC 向けの.NET Framework には、WebClient という名前のクラスが提供されていて、このクラスを使うと、非常に簡単に指定した URL から画像をダウンロードし、ファイルに保存できるのですが、小型組込み機器では、ファイルシステムすら載っていないデバイスも普通な世界でもある為、非常に残念ながら WebClient クラスは、.NET Micro Framework では提供されていません。

画像データはストリームで取得する処理であり、HttpWebRequest、HttpWebResponse クラスを使っての処理が出来ない為、ImageWebClient では、より低レベルな Socket クラスを使って、TCP/IP 接続を行い画像データのダウンロードを行っています。

Twitpic からのデータダウンロードの API は、http://twitpic.com/show/full/*shortId* で、ブラウザを使うと簡単に表示可能ですが、ブラウザの URL を確認すると、入力した URL ではなく別の URL に変わっていることが判ります。実は、この URL にアクセスすると、別の URL にリダイレクトして画像を取得するように応答が帰ってきて、ブラウザはそれを見てリダイレクトされた URL を再表示する仕組みになっています。ImageWebClient クラスには、その処理も入れておきました。

また、画像データは一般的に巨大な為、ダウンロードしたデータを通常の byte 配列に格納しようとすると、配列のコンストラクトで、アウトオブメモリエクセプションが発生する可能性があります。

ImageWebClient クラスではそれを防ぐ為、巨大なメモリを必要とする処理向けに用意されている Microsoft.SPOT.Hardware.LargeBuffer クラスを使って、byte 配列を確保しています。

以上、ポイントのみの説明ですが、後は、実際にエミュレータで動かし処理の流れを追って、概要を把握してください。

```
using System;
using Microsoft.SPOT;

namespace MFWinAppSample
{
    using System.Net;
    using System.Net.Sockets;
    using System.Text;
    using System.Threading;
    using Microsoft.SPOT.Hardware;

    class ImageWebClient
    {
        private string targetServer;
        private int connectionPort;

        public ImageWebClient(string server, int port)
        {
            targetServer = server;
            connectionPort = port;
        }

        public byte[] DownloadImage(string request,out string format)
        {
            format = null;
            const int microsecondsPerSecond = 1000000;
            using (Socket socket = ConnectSocket())
            {
                byte[] bytesToSend = Encoding.UTF8.GetBytes(
                    "GET " + request + " HTTP/1.1¥r¥n" +
                    "Host: " + targetServer + "¥r¥n" +
                    "Connection: Close¥r¥n¥r¥n");
                socket.Send(bytesToSend, bytesToSend.Length, 0);

                byte[] buf = new byte[1024];
                byte[] imageData = null;

                DateTime timeoutAt
                    = DateTime.Now.AddSeconds(30 * microsecondsPerSecond);
                while (socket.Available == 0 && DateTime.Now < timeoutAt)
```

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

string header = string.Empty;
string page = string.Empty;

bool headerReceived = false;
int offset = 0;
int size = 1;
int index = 0;

while (socket.Poll(
    30 * microsecondsPerSecond,
    SelectMode.SelectRead))
{
    if (socket.Available == 0)
    {
        break;
    }
    Array.Clear(buf, 0, size);
    int bytesRead = socket.Receive(
        buf, offset, size, SocketFlags.None);
    if (bytesRead == 0)
    {
        break;
    }
    if (headerReceived == false)
    {
        header += new string(Encoding.UTF8.GetChars(buf));
        if (buf[0] == '¥n' && header.IndexOf("¥r¥n¥r¥n") > 0)
        {
            headerReceived = true;
            size = buf.Length;
            Debug.Print(header);
            // Check Redirection
            string tagStatus = "HTTP/1.1 ";
            string status = header.Substring(tagStatus.Length);
            status = status.Substring(0, status.IndexOf("¥r¥n"));
            if (status.IndexOf("302") == 0)
            {
                // リダイレクトの指示を受けたので、
                // ダウンロード先を変更
                string imageUri = header.Substring(
                    header.IndexOf("Location: ")
                    + "Location: ".Length);
                imageUri =
                    imageUri.Substring(0, imageUri.Length
                                - "¥r¥n¥r¥n".Length);
                string redirectedServer = imageUri.Substring(
                    0, imageUri.IndexOf("/", "http://".Length)).
                    Substring("http://".Length);
                string redirectedRequest = imageUri.Substring(
                    imageUri.IndexOf("/", "http://".Length + 1));
                ImageWebClient imageClient =
                    new ImageWebClient(redirectedServer, 80);
```

```
                        imageData =
                            imageClient.DownloadImage(
                                redirectedRequest, out format);

                        Debug.Print("Redirect to server="
                            + redirectedServer
                            + ",request="
                            + redirectedRequest);
                        break;
                    }
                    else if (status.IndexOf("200") == 0)
                    {
                        // 画像のダウンロード成功。画像のフォーマットを取得
                        string tagContentType = "Content-Type: ";
                        string contentType = header.Substring(
                            header.IndexOf(tagContentType)
                                + tagContentType.Length);
                        contentType = contentType.Substring(
                            0, contentType.IndexOf("¥r¥n"));
                        format = contentType;
                        Debug.Print("To be downloading...");
                    }
                    else
                    {
                        Debug.Print(
                          "Download Failed - Status: '" + status + "'");
                        break;
                    }

                    // 画像データのデータ長を取り出す。
                    string tag = "Content-Length:";
                    int lenIndex = header.IndexOf(tag);
                    if (lenIndex > 0)
                    {
                        string sizeStr = header.Substring(
                            lenIndex + tag.Length,
                            header.IndexOf("¥r¥n",
                            lenIndex) - lenIndex - tag.Length);
                        int len = int.Parse(
                            header.Substring(
                            lenIndex + tag.Length,
                            header.IndexOf("¥r¥n",
                            lenIndex) - lenIndex - tag.Length));

                        Debug.Print("len = " + len.ToString());

                        // 画像データが巨大なのでLargeBufferクラスを利用
                        LargeBuffer lbuffer = new LargeBuffer(len);
                        imageData = lbuffer.Bytes;
                    }
                }

            }
            else
            {
                Array.Copy(buf, 0, imageData, index, bytesRead);
                index += bytesRead;
```

```
                    }
                }

                return imageData;
            }
        }

        private Socket ConnectSocket()
        {
            IPHostEntry hostEntry = Dns.GetHostEntry(targetServer);
            Socket socket = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);
            socket.Connect(new IPEndPoint(hostEntry.AddressList[0],
                connectionPort));
            return socket;
        }
    }
}
```

註:このコードは、http://www.codeproject.com/KB/dotnet/UsingGoogleMapInMF.aspx に公開されているコードを元に作成されています。

さて、画像ダウンロード機能を持つクラスが出来たので、前節まで作成したアプリケーションに、本機能を加えます。

前節までは、ビューワに表示する画像は、リソースに登録した画像を使っていましたが、これを Twitpic からダウンロードした画像に変更します。

ImageViewerWindow クラスの BuildImageView()メソッドのリソースからビットマップを作成している部分を以下のように修正します。

```
    private void BuildImageView(Canvas canvas)
    {
        TwitpicImageWebClient twitpic = new TwitpicImageWebClient();
        TwitpicImageProfile[] imageProfiles =
            twitpic.GetImageProfiles("titterAccount");
        int numOfDownloaded = twitpic.DownloadImages(imageProfiles);

        if (images.Length > 0)
        {
            images = new Bitmap[numOfDownloaded];
            imageNames = new string[numOfDownloaded];
            for (int i = 0; i < imageProfiles.Length; i++)
            {
                if (imageProfiles[i].imageByte != null)
                {
                        Bitmap.BitmapImageType format =
                            twitpic.GetImageFormat(imageProfiles[i]);
                        images[i] = new Bitmap(
                            imageProfiles[i].imageByte, format);
                        imageNames[i] = imageProfiles[i].ShortId;
                }
            }

            DrawImageOnView(images[currentIndex]);
```

```
            SetImageTitle(imageNames[currentIndex]);
        }
```

4 行目の *twitterAccount* の部分は、Twitter 上に存在する適当なアカウント名に置き換えてください。

WebClient 系のクラスが非常に煩雑で長かったのに比べ、こちらの修正は簡単です。
TwitpicImageWebClient オブジェクトを作成し、TwitpicImageProfile オブジェクトを収集し、それらオブジェクト個々に対応する画像データをダウンロードしています。ダウンロードできた画像が 1 個以上あれば、images 配列に Bitmap オブジェクトを作成して登録、その後描画を行っています。

F5 でエミュレータを起動して、実行結果を確認してください。

以上、かなり難易度の高い内容も交えながら、直ぐに役立ちそうな機能を使って、.NET Micro Framework のネットワークプログラミングを学習してきました。一見難しそうですが、ネットワークへのアクセスや XML データの解析は、様々な場面でほぼ同じパターンを利用可能です。実際にプログラムをステップ実行し、処理の流れを確認し、パターンを把握することをお勧めします。

ファイルシステム

前節の最後で作成したアプリケーション、いかがでしたか。実はネットワークからの画像ダウンロードは非常に時間がかかるので、「なんだこりゃ」と思った方も多いでしょう。

ただでさえ画像データのサイズが大きくダウンロードに時間がかかるのに、小型組み込み機器の場合、PC に比べて性能の低い CPU を使っているので、遅さ倍増かもしれません。しかし、処理時間の遅さは、ほぼネットワークの通信速度に依存してしまうので、.NET Micro Framework を使わなくてもそれほど変わりはないでしょう。そこで、ダウンロードした画像をデバイスのストレージに格納し、無駄なダウンロードを行わない機能を追加し、我慢は一回だけのアプリケーションに修正します。

> **註:** .他の方法として、画像ダウンロードの処理は別スレッドにして、バックグラウンドで処理し、ユーザーインターフェイスは応答できるようにしておくとよいでしょう。

ファイルを操作するクラス

.NET Micro Framework には、File、Directory、Path、Stream、FileStream、StreamReader/StreamWriter、TextReader/Writer 等々、通常の PC 向け.NET Framework で提供されるファイル操作用のクラスが提供されています。使い方は、PC 向けのものと全く同じです。PC 上でファイル操作をプログラミングするのと同様にアプリケーションを書いてください。本書ではこれらに関する使い方はこれ以上説明しませんので、豊富にある書籍やネットワーク上の技術情報を参照してください。

…しかし、ここで話が終わらないのが、小型組込み機器向けソフトウェア開発の良いところですね。実は、小型組込み機器には、ハードディスクを搭載していないものが多いのです。デフォルトでファイルシステムを搭載していないデバイスすら存在すらあるしまつ。ファイルを操作するクラス群が揃っていてもファイルへのデータ保存がこれではできません。

RemovableMedia クラス

大丈夫です。大抵の小型組み込み機器にはハードディスクの代わりに、SD カードなど、リムーバブルメディアカードを差し込めるものが多くあります。メディアカードがあれば、ファイルをカードに書き込むことができます。.NET Micro Framework では、RemovableMedia クラスが用意され、このクラスにイベントハンドラを登録しておいて、メディアカードの抜き差し時、そのイベントをハンドルすることが可能です。

まず、プロジェクトに、新しいクラスを MediaCardStorage という名前で追加します。このクラスのコードは以下の通りです。

```
using System;
using Microsoft.SPOT;

namespace MFWinAppSample
{
```

```
    using Microsoft.SPOT.IO;
    using System.Collections;
    using System.IO;

    public delegate void OnInsertMediaCardHandler(MediaCardStorage storage);

    public class MediaCardStorage
    {
        private static bool isInitialized = false;
        private static MediaCardStorage myInstance = null;

        public bool IsUsable
        {
            get
            {
                return (CurrentVolume != null);
            }
        }

        public VolumeInfo CurrentVolume { get; set; }

        public static void Initialize()
        {
            RemovableMedia.Insert +=
                new InsertEventHandler(RemovableMedia_Insert);
            RemovableMedia.Eject +=
                new EjectEventHandler(RemovableMedia_Eject);
        }

        public static MediaCardStorage GetInstance()
        {
            if (myInstance == null)
            {
                myInstance = new MediaCardStorage();
            }
            return myInstance;
        }

        private MediaCardStorage()
        {
            RemovableMedia.Insert += new InsertEventHandler(RemovableMedia_Insert)
;
            RemovableMedia.Eject += new EjectEventHandler(RemovableMedia_Eject);
        }

        static void RemovableMedia_Eject(object sender, MediaEventArgs e)
        {
            if (isInitialized)
            {
                GetInstance().CurrentVolume = null;
            }
        }

        static void RemovableMedia_Insert(object sender, MediaEventArgs e)
        {
            if (isInitialized)
            {
```

```
                GetInstance().CurrentVolume = e.Volume;
                GetInstance().OnInsertMediaCard(GetInstance());
            }
        }

        public event OnInsertMediaCardHandler OnInsertMediaCard;
    }
}
```

> **註:** .このコードをビルドするには、新たにアッセンブリを参照追加する必要があります。アッセンブリは、既に紹介しているディレクトリに用意されています。必要なアッセンブリは、大抵の場合、名前空間と同じです。対応するアッセンブリが無い場合は、http://msdn.microsoft.com/en-us/library/ee435793.aspx を参照して、使用するクラスを探し、左上の Assembly の記述を参照してください。これ以降に出てくるコードをビルドするにも、アッセンブリの参照追加が必要なものがあります。適宜参照を追加して、作業を進めてください。

RemovableMedia クラスは、Insert と Eject というイベントが用意されています。メディアカードが差されたとき、Insert が発火し、メディアカードが抜かれたとき、Eject が発火します。RemovableStorageClas では、これら二つのイベントにそれぞれ、RemovableMedia_Insert メソッドと、RemovableMedia_Eject メソッドを登録しています。

メディアカードが差された時、それを受けて RemovableMedia_Insert()メソッドでは、引数で送られてきた Volume 変数を CurrentVolume プロパティに保持しています。

メディアカードを読み書きするデバイスは、アプリケーションにとって固定リソースであり、ここでは RemovableStorageClase をシングルトンで定義しています。

このクラスの使い方は、次の通りです。

- アプリケーション初期化時に、Initialize()メソッドをコール
- 必要に応じて、GetInstance()でオブジェクトを取得し、CurrentVolume プロパティを参照

RemovableMedia_Insert の引数で渡された Volume 変数には、差し込まれたメディアカード上のルートディレクトリの情報が格納されています。リムーバブルメディアカードにファイルを作成したい場合は、

```
        var storage = MediaCardStorage.GetInstance();
        string fileName = "text.txt"
        string path =
            Path.Combine(storage.CurrentVolume.RootDirectory, fileName);
        var stream = File.Create(fileName );
        TextWriter writer = new StreamWriter(stream) as TextWriter;
        writer.WriteLine("Hello");
```

の様にコーディングを行います。要は、RemovableMedia クラスの差し込み時のイベントとともに渡された引数の Volume プロパティの中の、RootDirectory プロパティに格納された文字列にファイル名を Path クラスの Combine メソッドで繋げば、メディアカード上のファイルパスが出来上がるという寸法です。

では、前節まで作成してきたアプリケーションにメディアカードへのファイル蓄積機能を追加しましょう。

アプリケーションに画像ファイルの保存と読み出し機能を加える

まず、Program クラスの Main()メソッドに、RemovableMediaStorage の初期化コードを追加します。

```
        public static void Main()
        {
            Program myApplication = new Program();

            Window mainWindow = myApplication.CreateWindow();

            Microsoft.SPOT.Touch.Touch.Initialize(myApplication);
            MediaCardStorage.Initialize();

            UpdateSystemTime();

            // Start the application
            myApplication.Run(mainWindow);
        }
```

次に、ImageViewerWindow の BuildImageView()メソッドに以下の修正を加えます。

・前節で加えたネットワークからの画像ダウンロード処理と描画処理をオミット

・画像描画用 Canvas オブジェクトへのタッチイベントハンドラ登録

```
        private void BuildImageView(Canvas canvas)
        {
#if false    ←ここを追加
            TwitpicImageWebClient twitpic = new TwitpicImageWebClient();
            TwitpicImageProfile[] imageProfiles =
                twitpic.GetImageProfiles("embedded_george");
            int numOfDownloaded = twitpic.DownloadImages(imageProfiles);

            if (images.Length > 0)
            {
                images = new Bitmap[numOfDownloaded];
                imageNames = new string[numOfDownloaded];
                for (int i = 0; i < imageProfiles.Length; i++)
                {
                    if (imageProfiles[i].imageByte != null)
                    {
                        Bitmap.BitmapImageType format =
                            twitpic.GetImageFormat(imageProfiles[i]);
                        images[i] = new Bitmap(imageProfiles[i].imageByte, format)
;
                        imageNames[i] = imageProfiles[i].ShortId;
                    }
                }

                DrawImageOnView(images[currentIndex]);
                SetImageTitle(imageNames[currentIndex]);
            }
#endif       ←ここを追加
```

```
            // 画像表示変更用タッチ領域用に四角形と文字列を作成 - 前へ移動
            var prevPart = DrawChangePart("Prev", 180, 15);
            prevPart.TouchDown +=
                new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchDown);
            prevPart.TouchUp +=
                new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchUp);

            // 画像表示変更用タッチ領域に四角形と文字列を作成 - 後へ移動
            var nextPart = DrawChangePart("Next", 180, 275);
            nextPart.TouchDown +=
                new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchDown);
            nextPart.TouchUp +=
                new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchUp);

             // 画像表示領域用Canvasにタッチダウンハンドル用ハンドラー登録
            imageViewCanvas.TouchDown +=
                new TouchEventHandler(imageViewCanvas_TouchDown);        ←追加
        }
```

これで、アプリケーション起動時のダウンロードがなくなりました。次にキャンバスにタッチされた時の処理を追加します。

```
        void imageViewCanvas_TouchDown(object sender, TouchEventArgs e)
        {
            Thread thread = new Thread(DownloadImageFiles);
            thread.Start();
        }
```

そのまま、画像のダウンロードに突入すると、タッチイベントの処理がメインで実行され、パネルへの他のイベントが取れなくなってしまうので、新しくスレッドを作成して、そのスレッド上でダウンロードを行うようにします。

実際にスレッドで処理される内容は以下の通りです。

```
        private void DownloadImageFiles()
        {
            TwitpicImageWebClient twitpic = new TwitpicImageWebClient();
            var profiles = twitpic.GetImageProfiles("embedded_george");
            int index = 0;
            while (true)
            {
                if (twitpic.DownloadImage(profiles[index]))
                {
                    images = new Bitmap[1];
                    imageNames = new string[1];
                    Bitmap.BitmapImageType format =
                        twitpic.GetImageFormat(profiles[index]);
                    images[0] = new Bitmap(profiles[index].imageByte, format);
                    imageNames[0] = profiles[index].ShortId;

                    currentIndex = 0;
                    DrawImageOnView(images[currentIndex]);
                    SetImageTitle(imageNames[currentIndex]);

                    StoreImage(profiles[index],format);

                    index++;
```

```
                break;
            }
            index++;
        }

        int rest = profiles.Length - index;
        if (rest > 0)
        {
            ArrayList availables = new ArrayList();
            for (int i = 0; i < availables.Count; i++)
            {
                if (twitpic.DownloadImage(profiles[i]))
                {
                    availables.Add(profiles[i]);
                    StoreImage(profiles[index],
                        twitpic.GetImageFormat(profiles[index]));
                }
            }

            Bitmap current = images[0];
            string currentname = imageNames[0];
            images = new Bitmap[availables.Count + 1];
            imageNames = new string[availables.Count + 1];
            images[0] = current;
            imageNames[0] = currentname;
            for (int i = 0; i < availables.Count; i++)
            {
                TwitpicImageProfile profile =
                    (TwitpicImageProfile)availables[i];
                images[i + 1] =
                    new Bitmap(profile.imageByte,
                        twitpic.GetImageFormat(profile));
                imageNames[i + 1] = profile.ShortId;
            }
        }
    }
```

この処理では、TwitpicImageWebClient から画像情報を取り出して、最初の項目を先ずダウンロードし、パネルへの表示とメディアカードへの格納をしてしまいます。中にはダウンロードし損ねる場合もあるので、単純に先頭の画像情報を使うのではなく while 文でループしています。

残った画像情報は for ループで回し、メディアカードへ保存しながら、パネルで表示する images バッファに格納していきます。

そして次が、メディアカードへのファイルの書き込み処理です。

```
    private void StoreImage(
        TwitpicImageProfile profile,
        Bitmap.BitmapImageType format)
    {
        var storage = MediaCardStorage.GetInstance();
        if (storage.IsUsable)
        {
            if (profile.IsDownloaded)
            {
                string path = Path.Combine(
                    storage.CurrentVolume.RootDirectory,
```

```
                    profile.ShortId + "." + format.ToString());
                if (!File.Exists(path))
                {
                    FileStream stream = File.Create(path);
                    stream.Write(profile.imageByte, 0, profile.Size);
                    stream.Close();
                }
            }
        }
    }
```

MediaCardStorage の GetInstance()メソッドを使ってオブジェクトを取り出し、IsUsable プロパティで、メディアカードが差さっているか確認します。差さっていれば、既に同じ名前のファイルがないか確認し、無ければ、イメージファイルとしてメディアカードに保存します。見ての通り、ルートディレクトリの取得方法はちょっと違いますが、ほかは通常のファイル処理と全く同じです。

最後に、メディアカードを差し込んだ時に、メディアカードから画像ファイルを取り込んで、表示するように変更します。

先ず、ImageViewerWindow クラスの BuildImageView()メソッドで、MediaCardStorage へのイベント登録処理を加えます。

```
    private void BuildImageView(Canvas canvas)
    {
        // 画像表示変更用タッチ領域用に四角形と文字列を作成 - 前へ移動
        var prevPart = DrawChangePart("Prev", 180, 15);
        prevPart.TouchDown +=
            new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchDown);
        prevPart.TouchUp +=
            new Microsoft.SPOT.Input.TouchEventHandler(prevPart_TouchUp);

        // 画像表示変更用タッチ領域に四角形と文字列を作成 - 後へ移動
        var nextPart = DrawChangePart("Next", 180, 275);
        nextPart.TouchDown +=
            new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchDown);
        nextPart.TouchUp +=
            new Microsoft.SPOT.Input.TouchEventHandler(nextPart_TouchUp);

        // 画像表示領域用Canvasにタッチダウンハンドル用ハンドラー登録
        imageViewCanvas.TouchDown +=
            new TouchEventHandler(imageViewCanvas_TouchDown);

        MediaCardStorage.GetInstance().OnInsertMediaCard +=          ←追加
            new OnInsertMediaCardHandler(ImageViewerWindow_OnInsertMediaCard);
    }
```

そして、イベントハンドラを実装します。

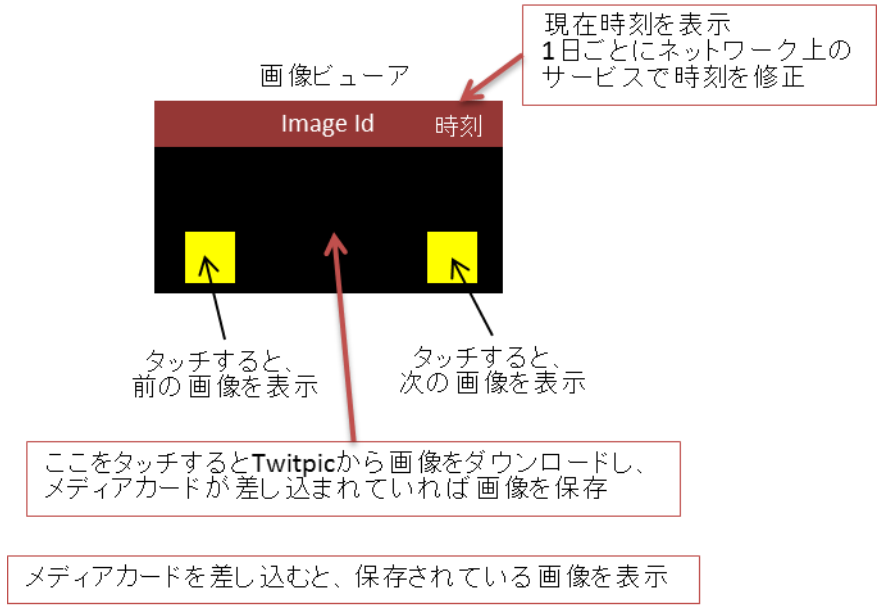
```
    void ImageViewerWindow_OnInsertMediaCard(MediaCardStorage storage)
    {
        // 画像情報をメディアカードから取り出し描画する
        ArrayList loadedImages = new ArrayList();
        ArrayList loadedImageNames = new ArrayList();
        string[] files = Directory.GetFiles(
            storage.CurrentVolume.RootDirectory);
        for (int i = 0; i < files.Length; i++)
        {
```

```
                FileInfo fi = new FileInfo(files[i]);
                if (fi.Extension.CompareTo("jpeg") == 0)
                {
                    FileStream stream = File.Open(
                        fi.FullName, FileMode.Open, FileAccess.Read);
                    LargeBuffer lbuf = new LargeBuffer((int)fi.Length);
                    byte[] buf = lbuf.Bytes;
                    int readBytes = stream.Read(buf, 0, (int)fi.Length);
                    loadedImages.Add(buf);
                    loadedImageNames.Add(fi.Name);
                }
            }

            images = new Bitmap[loadedImages.Count];
            imageNames = new string[loadedImageNames.Count];

            for (int i = 0; i < loadedImages.Count; i++)
            {
                images[i] = new Bitmap(
                    loadedImages[i] as byte[], Bitmap.BitmapImageType.Jpeg);
                imageNames[i] = loadedImageNames[i] as string;
            }

            currentIndex = 0;
            DrawImageOnView(images[currentIndex]);
        }
```

こちらも保存の時と同様、ルートディレクトリの値を取得するところと、バッファを LargeBuffer から調達している以外は、通常のファイル操作と変わりません。

ここまで修正が終わったら、エミュレータで実際の動きを追ってみてください。

最終的に出来上がったアプリケーションは、正しく実装されていれば、起動後 3 秒間青い画面で Welcome!を表示した後、以下の動作を行います。



最後に、エミュレータ上でリムーバブルメディアの抜き差しをエミュレートする方法を説明して、本節は終わりとします。

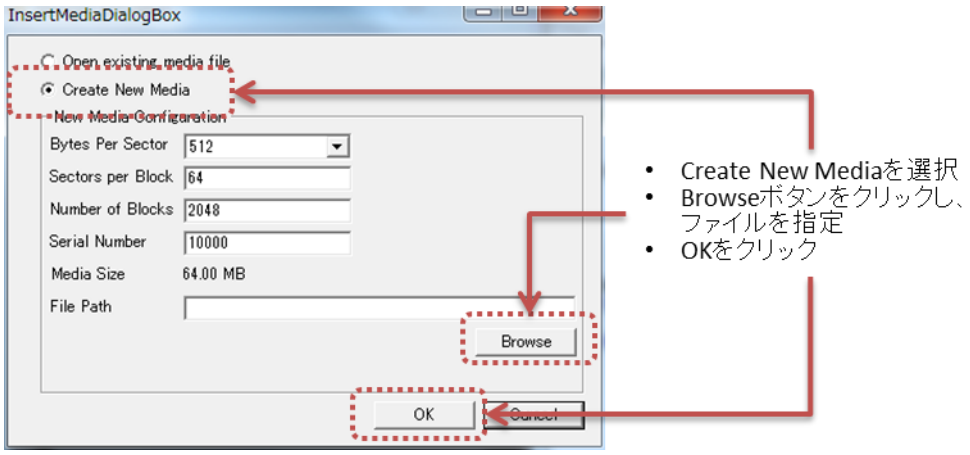
・メディアカードの抜き差しのエミュレーション

F5 実行でエミュレータを表示します。

エミュレータのメニューで、“Insert/Eject”→“SD1: Insert Media”を選択します。

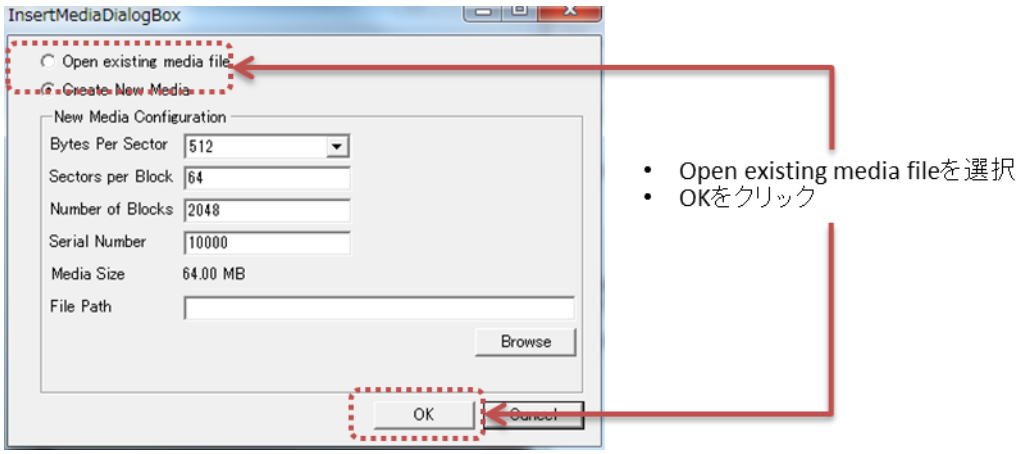
エミュレータは、PC のハードディスク上のファイルをエミュレータ上のメディアカードにマップし、アプリケーションで書き込んだ情報を再利用することができます。上の操作を行うと、以下のダイアログが表示されます。

図の操作に従って、メディアカード向けファイルを指定し、OK をクリックします。



これでメディアカード用のファイルが作成され、アプリケーション上は、メディアカードが差しこまれた事象のエミュレートができます。

既に一度メディアカード向けファイルを作成している場合は、



Open existing media file を選択すると、ファイル選択のダイアログが表示されるので、そこでファイルを選択します。

まとめ

第 1 部の基礎 編では、.NET Micro Framework の概要、開発にあたっての準備、アプリケーション開発に関する基本事項を取り上げました。

特にアプリケーション開発にあたってよく使われると思われる、ユーザーインターフェイスの構築とネットワークプログラミングについて、基礎的な事項を、簡単なアプリケーションを構築しながら、解説しました。例題で挙げたアプリケーションを出発点として、様々な改良を施し、.NET Micro Framework 上でのアプリケーション開発の基礎を習得してください。

本書で取り上げた内容が、.NET Micro Framework を今後使用する上で、役立って頂ければ幸いです。

参考 URL

- .NET Micro Framework 開発者向け技術情報
  http://msdn.microsoft.com/ja-jp/netframework/bb267253.aspx
- .NET Micro Framework 開発者フォーラム
  http://social.msdn.microsoft.com/Forums/ja-JP/netmicroframeworkja/threads
- .NET Micro Framework ホームページ(英語)
  http://www.netmf.com
- .NET Micro Framework Platform SDK(英語)
  http://msdn.microsoft.com/en-us/library/ee436350.aspx
- 組込み開発者向け技術情報
  http://msdn.microsoft.com/windows/kumikomidev