4. C++ と C# で学ぶ WiiRemote プログラミング

デザイン後送

この章ではWiimoteLibというAPIを利用して、WiiRemoteのプログラミングの基礎を解説していきます。プログラミング環境として、無料で利用できる「Visual C++ 2008 Express Edition」もしくは「Visual C# 2008 Express Edition」を使います。つまりプログラミング言語として、C++とC#を同時に並列で扱います。

本章のC#.NETによるプログラミングサンプルは第1章で紹介した金沢工業高等専門学校講師小坂崇之氏によるものです。プログラミング初心者でもわかりやすく理解できるように、できるだけ多くのサンプルを丁寧に順を追って解説することで、WiiRemoteプログラミングを体験できるようになっています。

Kosaka Laboratory
URL http://www.kosaka-lab.com/

4.1 プログラミング環境のセットアップ

この節では、無料で利用できるVisual C++ 2008 Express EditionとVisual C# 2008 Express Editionをセットアップします。すでにこれらの製品の上位バージョン（Standard Editionなど）をインストールされている方は、必要なところだけ参照してください。

Visual C++ 2008 Express Editionのセットアップ

ここではMicrosoft Visual C++ 2008 Express Editionのセットアップについて解説します。すでにVisual C++や.NETといった開発環境をお使いの方は、読み飛ばしていただいても問題ありません。

まず、Microsoftのホームページから Microsoft Visual C++ 2008 Express Editionをダウンロードします。「Webインストール（ダウンロード）」をクリックすると、Webインストール版セット

アップファイル「vcsetup.exe」をダウンロードできます。

必要なハードディスク容量の確認、起動中のアプリケーションの終了などを行ってからインストールウィザード「vcsetup.exe」を起動します。

図4-1　Microsoft Visual Studio Express製品のホームページ

Visual C++ 2008 Express Edition

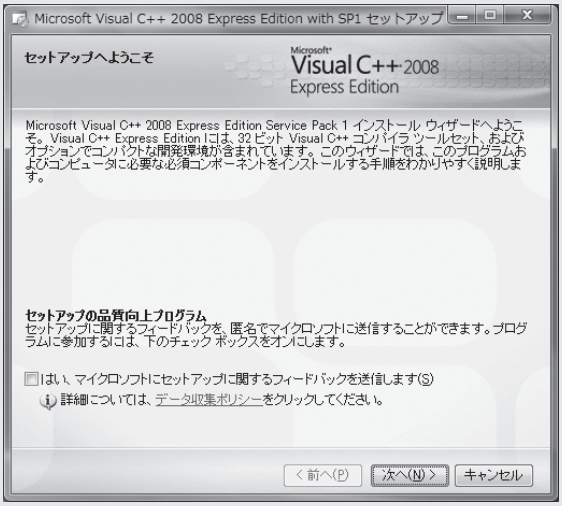
URL http://www.microsoft.com/japan/msdn/vstudio/express/

POINT ▸▸▸ このホームページにはVisual Studio製品を使う上での役に立つ情報がたくさんあります。とりあえずVC2008を使ってみたい方は「はじめての方のためのインストール方法」を読んでみるとよいでしょう。

1　ウィザードの起動

「セットアップの品質向上プログラム」はチェックしてもしなくても、どちらでもかまいません。「次へ」をクリックして進みます。

図4-2　VC2008インストールウィザード（1）



2　ライセンス条項に同意する

ライセンス条項をよく読んで「同意する」を選んでください。また「Visual Studio でオンラインのRSS コンテンツを受信して表示できるようにする」も、特に問題がなければチェックしましょう。イベントやサービスパックなどの更新情報がVC2008 起動直後に表示されるスタートページに自動的に表示されるようになります※1。「次へ」をクリックして進みます。

3　インストールオプションの選択

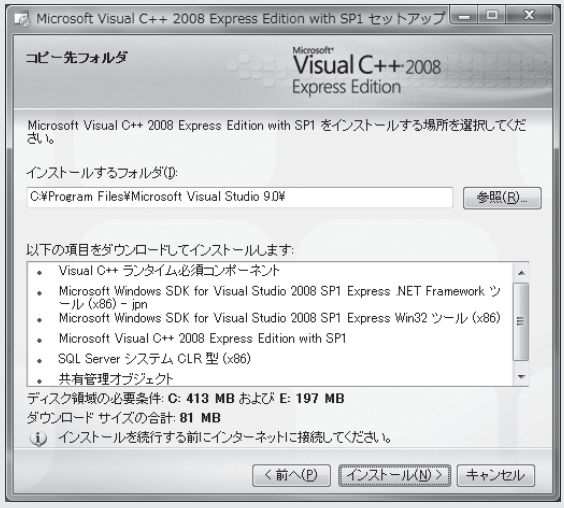
「インストールオプション」を選択します。ここで表示されている3つのオプションは、どれもインストールしなくても問題ありません。「MSDN Express ライブラリ」はF1 キーで呼び出せるドキュメントで、オンライン版のほうが充実しているのですが、筆者は電車の中でコーディングをすることが多いのでインストールしています（オンライン版と統合して利用できます）。「SQL Server 2005」、「Silverlight」は使う予定がなければインストールしなくてよいでしょう。「次へ」をクリックして進みます。

図4-3　VC2008インストールウィザード（2）

4　インストール先フォルダの指定

「コピー先フォルダ」とダウンロードパッケージのリストです。「インストールするフォルダ」は本書ではデフォルトのままとして解説します。

図4-4　VC2008インストールウィザード（3）



※1：ここでのRSS受信の設定は後でも［オプション］→［環境］→［スタートアップ］で変更できます。

「インストール」をクリックすると、ダウンロードとインストールが実行されます。

図 4-5　VC2008 インストールウィザード (4)

以上が Windows XP シリーズにおける VC2008 セットアップの流れです (Windows Vista の場合も、「管理者権限で実行」をする必要がある点などいくつか細かい点が異なりますが、ほぼ同じ流れです)。興味がある方は VC2008 のウィザードやオンラインチュートリアルなどを使って簡単なプログラミングを試してみるとよいでしょう。

Visual C# 2008 Express Edition のセットアップ

Visual C# 2008 Express Edition のセットアップ方法も、前述の Visual C++ 2008 Express Edition のセットアップと流れは変わりません。また、Microsoft の公式サイトを参考にしてください。

Microsoft Visual C# 2008 Express Edition インストール方法
URL http://www.microsoft.com/japan/msdn/vstudio/express/beginners/2008/vcsharp.aspx

Visual Studio混在環境を解決するVersion Selector

もし、すでにVisual C++やVisual Studioの過去のバージョン（.NET2003/2005など）をお使いで、かつ2008年時点の最新版であるVisual C++ 2008 Express（以下VC2008）を試してみたいの読者は、ここでちょっと回り道をして試してみることをおすすめします。

VC2008にはVersion Selectorという機能があり、異なるバージョンのVisual Studio製品の混在を可能にします。各バージョンにおいてソリューションファイルは拡張子「.sln」と変わりませんが、自動的にこの拡張子「.sln」に関連づけられたアプリケーションである「Version Selector」がファイル内部のバージョン記述を自動的に読み取り、ファイルがダブルクリックされたときは適切なバージョンのVCを起動します。この機能のおかげで、複数のバージョンの開発環境を安全してインストールできるようになります。

4.2 WiimoteLibの概要

WiimoteLibはBrian Peek氏による.NET環境で利用できるAPIのオープンソースプロジェクトです。Microsoftが支援するオープンソースプロジェクト支援サイト「CodePlex」で公開されています。WiimoteLibを用いることで、.NET環境で簡単にWiiRemoteを利用するアプリケーションを開発できます。Version1.6.0.0からはWiiRemoteだけでなくWiiBoradにも対応しています。

Brian Peek
URL http://www.brianpeek.com/

CodePlex
URL http://www.codeplex.com/

WiimoteLibは.NETで開発されたAPIであり、言語に依存しませんが、C#とVisual Basicがメインのターゲットのようです。C++でのサンプルは配布されておらず、またネット上の情報もあまり存在しないのですが、本章を読み進めていくことで、各種言語で問題なく利用できることがわかるでしょう。

表　追加　後送

POINT ▸▸▸ C++にはさまざまな派生言語仕様が存在しますが、本章では特に.NETプログラミングを扱いやすい「C++/CLI」を扱います。CLIとは「Common Language Infrastructure（共通言語基盤）」の略で、C++に慣れ親しんだプログラマに、違和感の少ない形で先進的な.NETプログラミング環境を提供しています。本章ではその特徴を利用し、同じプログラミングをC++とC#という2つの言語で解説していきます。どちらか好きな言語で進めてください。

WiimoteLibのライセンス

WiimoteLibのライセンスは「Microsoft Permissive License（Ms-PL）」です。Ms-PLは、最も制限の緩いMicrosoftのソースコードライセンスで、ソースコードを商用または非商用の目的で表示、変更、再頒布できます。また希望する場合は、変更したソースコードに対してライセンス料を課金することもできます。以下に条文の日本語参考訳を引用しておきます。

Microsoft Permissive License (Ms-PL)
URL http://www.microsoft.com/japan/resources/sharedsource/licensingbasics/permissivelicense.mspx

● Microsoft Permissive License (Ms-PL)　公開日：2007年7月9日

本ライセンスは、付属するソフトウェアの使用に適用されます。本ソフトウェアを使用する場合は、本ライセンスに同意したものとみなします。本ライセンスに同意しない場合、本ソフトウェアを使用することはできません。

1. 定義

本ライセンスでは、「複製する」、「複製」、「二次的著作物」、および「頒布」という用語は、米国の著作権法の下で使われる場合と同じ意味を有します。「コントリビューション」とは、オリジナルのソフトウェア、またはソフトウェアに対する追加もしくは変更を意味します。「コントリビューター」とは、本ライセンスの下で自らのコントリビューションを頒布する者を意味します。「対象特許」とは、コントリビューションが直接抵触する、コントリビューターの有する特許権の請求範囲を意味します。

2. 権利の付与

(A) 著作権に関する許諾 - 第３条「条件および制限」を含む本ライセンスの条件に従って、各コントリビューターは使用者に対し、コントリビューションを複製し、コントリビューションの二次的著作物を作成し、コントリビューションまたは作成した二次的著作物を頒布する、非独占的、世界全域、無償の著作権ライセンスを付与します。

(B) 特許権に関する許諾 - 第３条「条件および制限」を含む本ライセンスの条件に従って、各コントリビューターは使用者に対し、本ソフトウェアのコントリビューションまたは本ソフトウェアのコントリビューションの二次的著作物を作成し、作成させ、使用し、販売し、販売を提案し、輸入し、および / またはその他の方法で処分する、対象特許に基づく非独占的、世界全域、無償の特許権ライセンスを付与します。

3. 条件および制限

(A) 商標の除外 - 本ライセンスでは、コントリビューターの名前、ロゴ、または商標を使用する権限は与えられません。

(B) 使用者が、本ソフトウェアによる侵害を主張する特許に関し、コントリビューターに対して特許侵害を主張する場合、当該コントリビューターによる本ソフトウェアについての使用者に対する特許ライセンスは自動的に終了します。

(C) 本ソフトウェアの全部または一部を頒布する場合、本ソフトウェアに付属するすべての著作権、特許権、商標、および出所の表示を保持する必要があります。

(D) 本ソフトウェアの全部または一部をソースコードの形式で頒布する場合は、頒布物に本ライセンスの完全な写しを含めた上で、本ライセンスの条件の下でのみ頒布することができます。本ソフトウェアの全部または一部をコンパイル済みまたはオブジェクトコード形式

で頒布する場合は、本ライセンスに抵触しない条件のライセンスの下でのみ頒布することができます。

（E）本ソフトウェアは現状有姿にてライセンスされます。本ソフトウェアの使用に伴う危険は、すべて使用者が負うものとします。コントリビューターからの明示的な保証または条件は一切ありません。使用地の法律に基づき、本ライセンスでは変更できないその他の消費者の権利が存在する場合があります。使用地の法律に基づいて許可される範囲で、コントリビューターは、商品性、特定目的に対する適合性、非侵害について、黙示的な保証を否定します。

（本サイトはhttp://www.microsoft.com/resources/sharedsource/licensingbasics/permissivelicense.mspxの参考訳です）

WiimoteLibのセットアップ

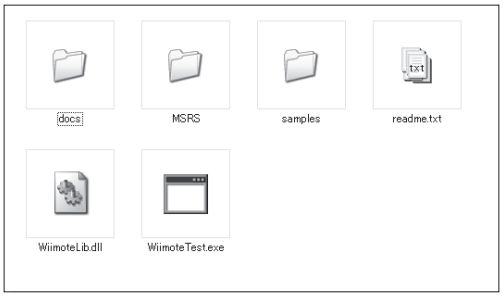
ライセンスを確認したら、WiimoteLibをダウンロードします。原稿出版時点での最新版は2009年1月19日に公開されたWiimoteLib v1.7です。

Managed Library for Nintendo's Wiimote
URL http://www.codeplex.com/WiimoteLib

図4-6　Managed Library for Nintendo's Wiimote

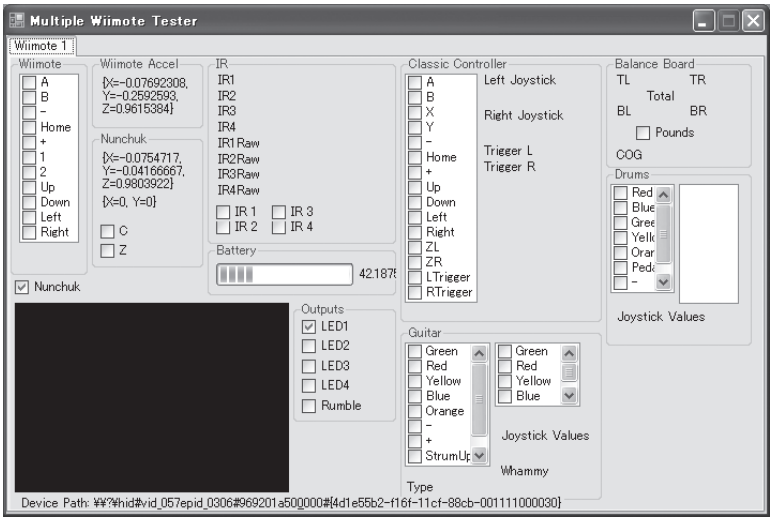
クリックすると、ライセンス条文が英語で表示されるので、確認したら「I Agree」クリックし、ダウンロードを開始します。ダウンロードしたZIPファイルを展開すると、以下のファイルがあるはずです。

図4-7　WiimoteLib1.7の同梱ファイル



まずは動作確認をします。お使いのBluetoothスタック管理ソフトウェアを起動して、WiiRemoteを1台接続します。接続できたら、展開したフォルダの中にある「WiimoteTest.exe」をダブルクリックして実行してみてください。正しく実行できると、数秒間の初期化の後、図4-8のような実行画面が表示されるはずです。WiiRemoteを振ったり、ボタンを押すことで、リアルタイムで値が取得できていることが確認できます。これでWiimoteLibを使えることが確認できました。フォームの右上の「×」(閉じるボタン)をクリックすると終了します。もし手元に複数のWiiRemoteやセンサーバー、ヌンチャク、ギターコントローラー、WiiBoardなどありました

図4-8　WiimoteLibのデモプログラム「WiimoteTest.exe」



ら、ぜひ接続して動作を試してみてください。

エラーが発生した場合、特に「Error reading data from Wiimote...is it connected?」と表示された場合は、Bluetoothでの接続に問題があります。Bluetoothスタックが正しくWiiRemoteを接続しているかどうか、確認してみてください。

図4-9 「WiimoteTest.exe」の起動に失敗したら接続を確認しよう

展開したファイルのうち「WiimoteLib.dll」が一番重要なファイルです。「WiimoteTest.ext」もこのDLLが同じディレクトリに存在しなければ正しく動作しません。なおWiimoteLibには特にインストーラはありません。展開したフォルダごと「マイドキュメント¥Visual Studio 2008¥Projects¥WiimotLib_1.7」に移動しておくと、このあとの作業が楽になるでしょう（WiimoteLib.dllだけでもいいのですが、ヘルプや複数のバージョンのDLLが混在すると、あとあと厄介です）。

なお「docs」フォルダにある「WiimoteLib.chm」がヘルプファイルです。ドキュメントもしっかりと整備されています。

これでセットアップは終わりです！ 4.5節以降ではVisual C# 2008 Express Edition（以降、C#と表記）、Visual C++ 2008 Express Edition（以降、C++と表記）を用いてWiiRemoteを制御していきます。

しかしその前に、それぞれの言語ごとにWiimoteLibの組み込みとプログラミングを体験しておきましょう。C#で学習する場合は次の4.3節に、C++で学習する場合は4.4節に進んでください。一度流れを覚えたら、以後はすべてのWiiRemoteプロジェクトで共通です。言語もC++かC#、どちらか1つでかまいません（悩むようなら本書においてはC#を推薦しておきます）。

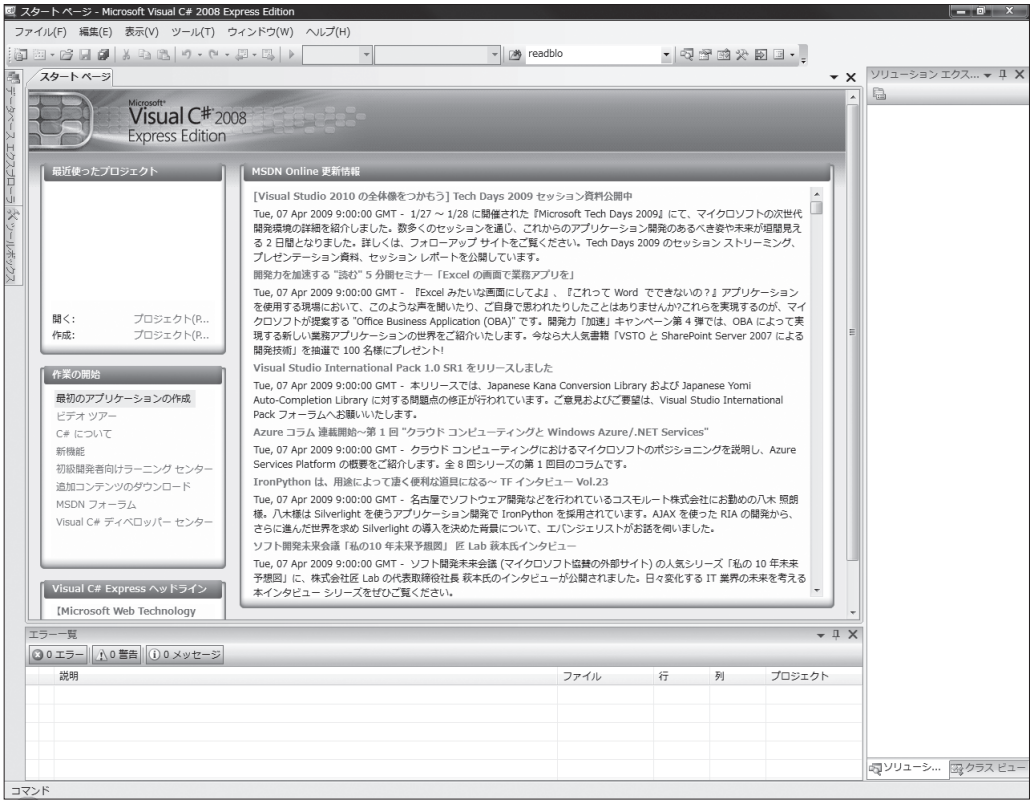
4.3 WiimoteLibをプロジェクトに組み込む【C#編】

まずはウィザードを使って、新しい空のプロジェクトを作成しましょう。

空のプロジェクトの作成

Visual C# 2008 Express Editionを起動します。

図4-10　Visual C#を起動したところ



［ファイル］→［新しいプロジェクト］→［Windows フォームアプリケーション］を選択し、「プロジェクト名」に「WiimoteLib01」という名前を付けて［OK］ボタンをクリックします。

図 4-11　新しいプロジェクトを作成

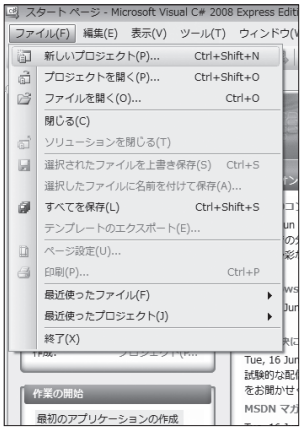
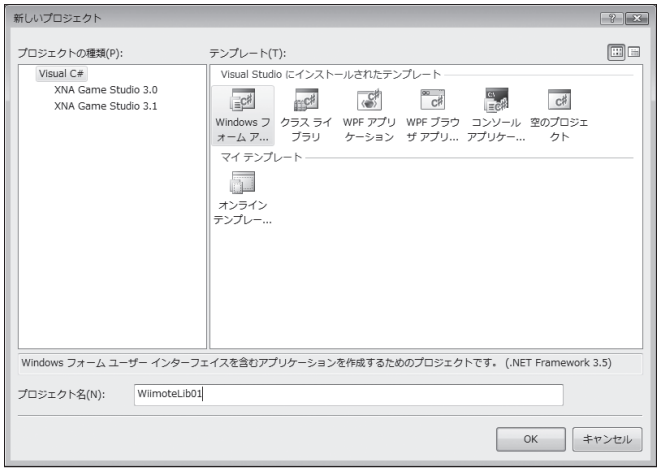
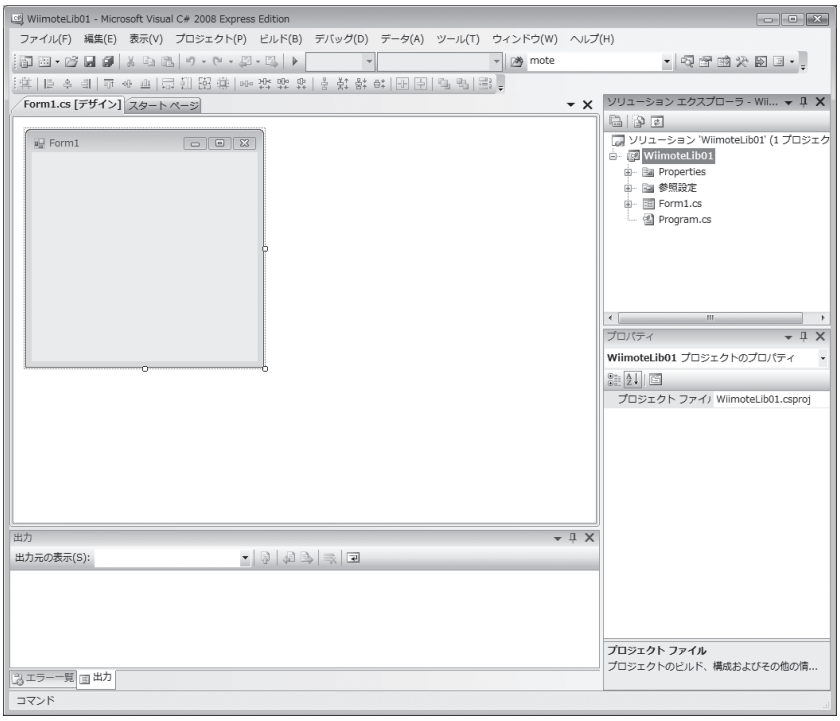


図 4-12　「プロジェクト名」に「WiimoteLib01」という名前を付けて［OK］



数秒待つと新しいプロジェクトが作成されます。興味があればここで「F5」キーを押して、実行してみるとよいでしょう。

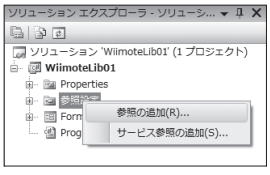
図4-13　新しいプロジェクトが作成されたところ



WiimoteLibの追加

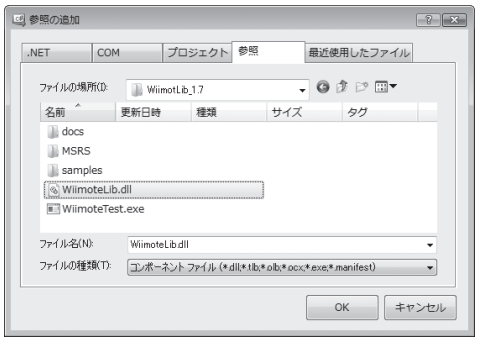
先ほど作成した空のプロジェクトにWiimoteLibを組み込んでいきましょう。右側に表示されている、ソリューションエクスプローラの［参照設定］を右クリックし、［参照の追加...］を選択します。

図4-14　［参照設定］を右クリック



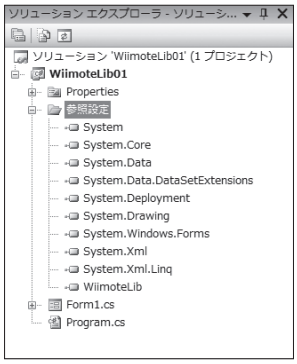
参照の追加から［参照］を選択し、WiimoteLib.dllを選択します。「マイドキュメント¥Visual Studio 2008¥Projects」に置いた「WiimoteLib_1.7¥WiimoteLib.dll」を選択し、「OK」ボタンをクリックします。

図 4-15　WiimoteLib.dll のファイルを指定する



これで、ソリューションエクスプローラの参照設定に WiimoteLib が追加されたはずです。

図 4-16　ソリューションエクスプローラーに現れた WiimoteLib



それでは、最小限のプログラムの実行結果を表示するためのフォームを作成しましょう。ソリューションエクスプローラの「Form.cs」を右クリックして「コードの表示」で表示される C# のコードに、最も重要な最初の 1 行「using WiimoteLib;」を追加します。

コード 4-1　C# WiimoteLib の読み込み

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

次ページにつづく

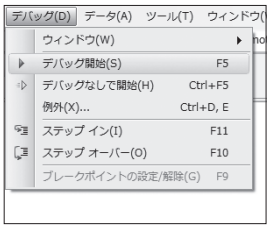
```
  using WiimoteLib;     //ここを1行追加します

  namespace WiimoteLib01  {   //指定したプロジェクト名
    public partial class Form1 : Form {

    public Form1() {
      InitializeComponent();
    }

  }
}
```

以上がC#環境でWiimoteLibを用いるための最初の一歩の操作です。まだWiiRemoteらしいことは何もできませんが、これでWiimoteLibのクラスが利用できるようになりました。次のステップで、実際に動作を確認してみましょう。

プログラムの実行

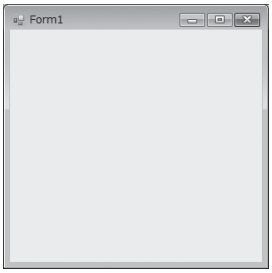
「F5」キー、または、[デバッグ]→[デバッグ開始]を押してプログラムを実行してみましょう。

図4-17 ［デバッグ］→［デバッグ開始］



プログラムにエラーがなければ図4-18のように表示されるはずです。

図4-18 何もないフォームが表示された



このプログラムは単にフォームを生成するプログラムです。「×」ボタンを押してフォームを閉

じてプログラムを終了させましょう。以後、このプログラムをベースに WiiRemote を制御するプログラムを追加していきます。

4.4 WiimoteLib をプロジェクトに組み込む【C++ 編】

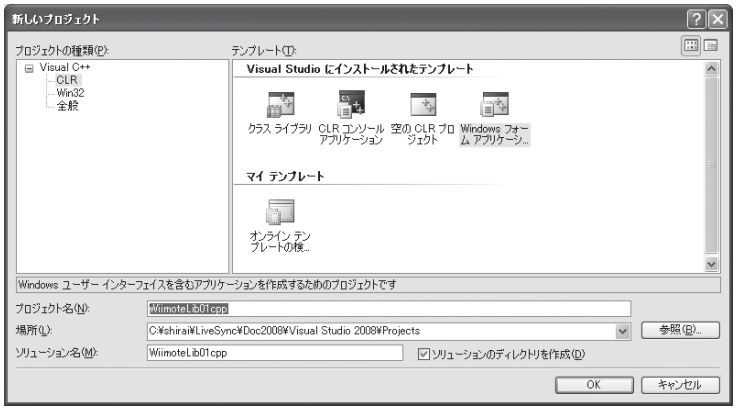
ここでは、C++ を使って WiimoteLib でのプログラミングを体験していきます。
まずはウィザードを使って、新しい空のプロジェクトを作成しましょう。

空のプロジェクトの作成

Visual C++ 2008 Express Edition を起動します。

［ファイル］→［新しいプロジェクト］から、［CLR］の［Windows フォームアプリケーション］を選択し、「プロジェクト名」に「WiimoteLib01cpp」という名前を付けて「OK」ボタンをクリックします。

図 4-19 「プロジェクト名」に「WiimoteLib01cpp」という名前を付けて［OK］ボタンをクリック



新しいプロジェクトが作成されました。興味があればここで「F5」キーを押して、実行してみるとよいでしょう。何もないフォームが表示され、「×」ボタンを押すと終了します。

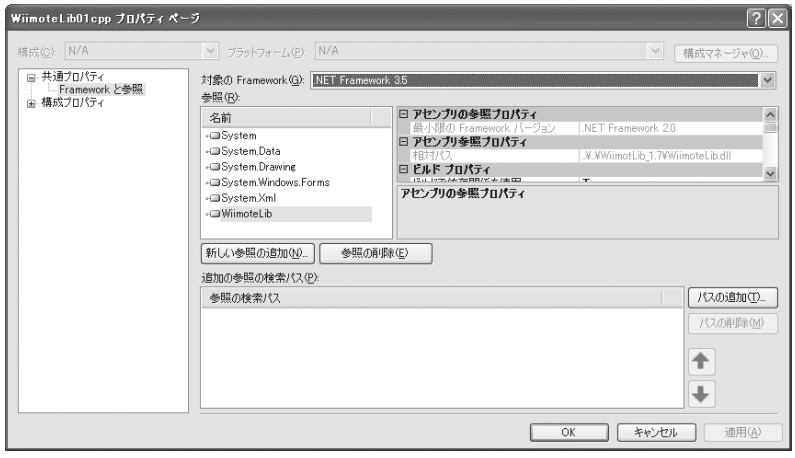
WiimoteLibの追加

先ほど作成したプロジェクトにWiimoteLibを組み込んでみましょう。ソリューションエクスプローラでプロジェクト（ここでは「WiimoteLib01cpp」）を右クリックして［参照］を選択します。

プロジェクトのプロパティページから［新しい参照の追加］をクリックし、［参照］タブをクリックし、ファイル選択ダイアログで、マイドキュメントの「Visual Studio 2008¥Projects」に置いた「WiimoteLib_1.7¥WiimoteLib.dll」を選択し「OK」ボタンをクリックします。

図4-20　プロジェクトを右クリックして［参照］を選択

図4-21　WiimoteLibが参照に追加された



次に、WiimoteLibの初期化コードを書きます。ソリューションエクスプローラの「Form1.h」を右クリックして「コードの表示」を選ぶと、Form1.hのコードが表示されます。このコードの先

頭12行目に以下のように、必要な1行を書き加えてください。

コード4-2 C++ WiimoteLibクラスを宣言(Form1.h)

```
#pragma once

namespace WiimoteLib01cpp {

  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;

  using namespace WiimoteLib;  //これを1行追加します

<以下略>
```

以上がC++/CLIでWiimoteLibを用いるための必要最低限のプログラムです。「F5」キーを押してプログラムを実行させてみましょう。

プログラムにエラーがなければC#と同様、何もないフォームが表示されるはずです。今のところ、このプログラムは単にフォームを生成するだけのプログラムですが、WiimoteLibのクラスがusing namespace宣言によって問題なく組み込まれていることがわかります。以後、このプログラムをベースにWiiRemoteを制御するプログラムを追加していきます。

4.5 バイブレータのON/OFF

ここからは、さらにWiimoteLibのAPIを用いてプログラミングしていきます。解説はC++とC#を並列して進めますが、.NETフレームワークのおかげでGUIの設計などはまったく同じ操作でできます。

まず、PC画面上に表示されるFormボタンによって、WiiRemoteの振動機能（バイブレーター）の動作を操るプログラムを作ります。

WiimoteLibの宣言と接続

前節のとおり、WiimoteLibを組み込んだプロジェクトのメインのコード（Form1.csもしくはForm1.h）に、以下の３行を追加します。

コード4-3 C# Form1.cs

```
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Data;
    using System.Drawing;
    using System.Linq;
    using System.Text;
    using System.Windows.Forms;
    using WiimoteLib;   //WiimoteLibを宣言

    namespace WiimoteLib01 {
     public partial class Form1 : Form {

        Wiimote wm = new Wiimote();     //Wiimoteクラスの作成

      public Form1() {
       InitializeComponent();
        wm.Connect();           //Wiimoteに接続

 }
     }
    }

＜以下略＞
```

コード4-4 C++ Form1.h

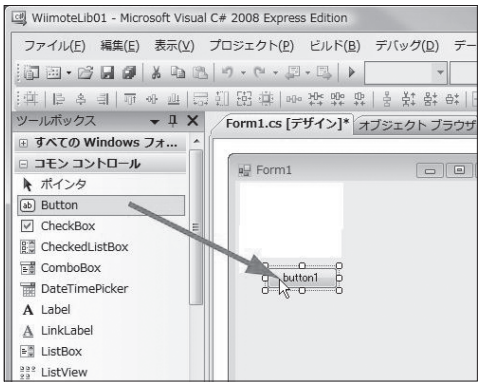
```
#pragma once
namespace WiimoteLib01cpp {
        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;
        using namespace WiimoteLib;     //WiimoteLibを宣言
<中略>
        public ref class Form1 : public System::Windows::Forms::Form
        {
        public: Wiimote^ wm;    //Wiimoteクラスの入れ物
        public:
                Form1(void)
                {
                        wm = gcnew Wiimote(); //Wiimoteクラスの作成;
                        InitializeComponent();
                        //
                        //TODO: ここにコンストラクタ コードを追加します
                        //
                        wm->Connect();  //Wiimoteに接続

                }
<以下略>
```

C#もC++も多少の記号や予約語は違えど、ほとんど同じであることがおわかりいただけたでしょうか？　C++ではwmという入れものをForm1クラスのPublicメンバーとして用意しています。

バイブレーター ON/OFF ボタンの作成

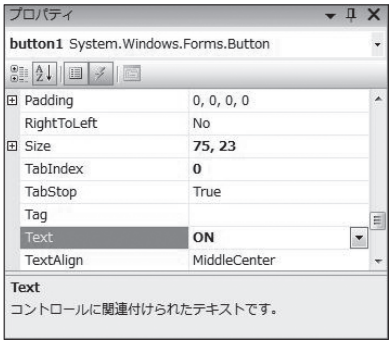
まず、フォームにボタンを貼り付けてください。C#では、ツールボックスからドラッグしてForm1の好きな位置に配置します（C++でも同様です）。表示されていない場合、［表示］から［デザイナ］としてForm1のデザインを表示し、再度、［表示］から［ツールボックス］を選ぶことで右側にツールボックスウィンドウが現れます（C++は左側）。「コモンコントロール」に「Button」があるので、フォームの上にドラッグしてください。

図4-22　ツールボックスからボタンをドラッグして配置



次に、貼り付けた「button1」のプロパティシートの「Text」を「button1」から「ON」に変更します。これでフォーム上のボタンに書かれているテキストが「ON」に変わるはずです。

図4-23　Textのプロパティを「button1」から「ON」に書き換える



Form1 上に配置したボタン「ON」をダブルクリックすると、ボタンクリック時のイベントを指定するコードが自動的に表示されるので、次のように記述します。

コード4-5　C# Form1.cs

```
private void button1_Click(object sender, EventArgs e) {
 wm.SetRumble(true);  //バイブーションON
}
```

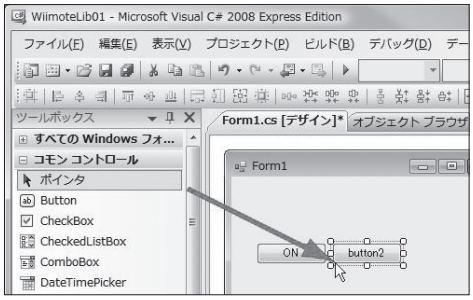
コード4-6 C++ Form1.h

```
#pragma endregion
private: System::Void button1_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
    wm->SetRumble(true);  //バイブーションON
  }
 };
}
```

気がはやる方はここで「F5」キーを押したくなるかもしれませんが、試すのは次のステップまで進んでからにしましょう！ このままでは、Bluetooth接続されていませんし、バイブレーターを駆動してもまだ止める方法を実装していませんので、ブルブル鳴りっぱなしの暴走状態になってしまいます。

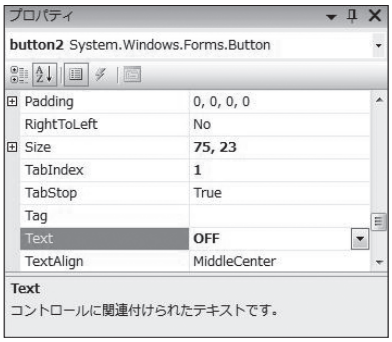
バイブレーターを停止させる「OFF」ボタンを作成します。先ほどと同様に、ツールボックスからボタンを配置します。

図4-24 「OFF」のためのボタンをドラッグで配置



先ほどと同じく、貼り付けた「button2」のプロパティのTextを「button2」から「OFF」に変更します。

図4-25　Textのプロパティを「button2」から「OFF」に書き換える



最後に、貼り付けた「button2」をダブルクリックし、次のようなコードを記述します。

コード4-7　C# Form1.cs

```
 private void button2_Click(object sender, EventArgs e) {
   wm.SetRumble(false);  //バイブーションOFF
 }
```

コード4-8　C++ Form1.cs

```
private: System::Void button2_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
   wm->SetRumble(false);       //バイブレーションOFF
 }
```

以上で終了です。これだけのプログラムでWiiRemoteのバイブレーション機能のON/OFF制御が可能になります。

コード4-9　C# Form1.csのC#ソース

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WiimoteLib;      //WiimoteLibの読み込み
```

次ページにつづく

```
namespace WiimoteLib01  {
 public partial class Form1 : Form {
   Wiimote wm = new Wiimote();   //Wiimoteの宣言
  public Form1() {
   InitializeComponent();
    wm.Connect();       //Wiimoteの接続
  }

  private void button1_Click(object sender, EventArgs e) {
    wm.SetRumble(true);     //バイブーションON
  }

  private void button2_Click(object sender, EventArgs e) {
    wm.SetRumble(false);   //バイブーションOFF
  }
 }
}
```

コード4-10　C++ Form1.hのC++ソース(変更点のみ抜粋)

```
#pragma once

<略>
        using namespace WiimoteLib; //WiimoteLibを宣言
<略>
  public ref class Form1 : public System::Windows::Forms::Form
  {
  public: Wiimote^ wm;         //Wiimoteクラスの入れ物
  public:
    Form1(void)
    {
      wm = gcnew Wiimote();   //Wiimoteクラスの作成;
      InitializeComponent();
      //
      //TODO: ここにコンストラクタ コードを追加します
      //
      wm->Connect();    //Wiimoteに接続

    }
<略>
#pragma endregion
private: System::Void button1_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
      wm->SetRumble(true);   //バイブーションON
  }
```

次ページにつづく

```
private: System::Void button2_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
      wm->SetRumble(false);  //バイブレーションOFF
  }
 };
}
```

簡単にプログラムの流れを解説します。

まず、「Connect() 関数」でWiiRemoteとの接続を行います。このときWiiRemoteが正しく接続・認識されていなかった場合、例外（Exception）が発生します。ただし、今回は例外処理を行っていません。実際のアプリケーションでは必要に応じて例外処理を追加してください。

ボタンを押したときに、SetRumble() 関数でWiiRemoteのバイブレーションを制御します。引数に「true」を入れるとバイブレーションが振動し、「false」を入れるとバイブレーションが停止します。

実行してみよう

まず、お使いのBluetoothスタックからWiiRemoteを接続します。

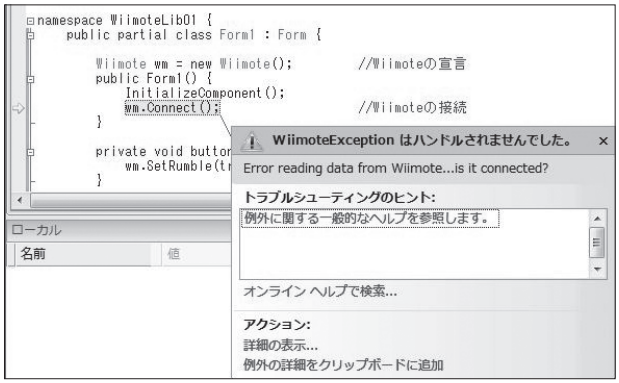
図4-26　Bluetooth接続（図は東芝製スタック）

続いて、Visual C#/C++から「F5」キーを押してプログラムを起動します。エラーがなければ、次のようなフォームが起動するはずです。

図 4-27 「ON」と「OFF」を持つフォームが表示される

もし、図 4-28 のようなエラーが発生する場合は WiiRemote が正しく接続されているか確認してください。

図 4-28 接続失敗：Bluetooth 接続を確認しよう



無事に起動できた場合、「ON」ボタンをクリックすると、バイブレーションが ON になり WiiRemote が振動します。あわてず騒がず「OFF」ボタンをクリックして、WiiRemote の振動を止めましょう。

どうでしょうか？ とても簡単に WiiRemote のバイブレーション制御プログラムを作ることができました。

解説：WiimoteLib の基本 API

表 4-2 は、このプログラムで使った WiimoteLib の API です。

表4-2　このプログラムで使用したWiimoteLibの基本API

C#	C++	解説
using WiimoteLib;	using namespace WiimoteLib;	ネームスペース宣言
public Wiimote wm;	public: Wiimote^ wm;	クラスの宣言
wm = new Wiimote();	Wiimote^ wm = gcnew Wiimote();	クラス新規作成
wm.Connect();	wm->Connect();	WiiRemoteに接続
wm.SetRumble(true);	wm->SetRumble(true);	バイブレーター作動
wm.SetRumble(false);	wm->SetRumble(false);	バイブレーター停止

いかがでしょう。.NET環境において、C#とC++では何ら変わりのないことがよくわかります。GUIによるフォーム作成も、マウスドラッグとプロパティの設定、ダブルクリックによる該当コードの自動生成があるので、非常に快適にコーディングできます。

次の節では同じ要領で、LEDの点灯を制御していきます。

POINT ▸▸▸ このようにWiimoteLibとC++/CLIやC#.NETの組み合わせで、簡単にアプリケーションを開発できます。WiimoteLibにはバイブレーターの制御以外にも、WiiRemoteを制御するための関数（メソッド）がたくさん揃っています。

4.6 LEDの点灯と消灯

次に、WiiRemoteのLEDを制御します。ここでは、Formボタンをクリックするごとに WiiRemoteの「プレイヤーインジケーター」と呼ばれる青色LEDをカウントアップさせます。

WiimoteLibの宣言と接続

準備にあたっての基本的なプログラミングの流れは前節のバイブレーターの制御の場合と同じ

です。新しいプロジェクトを作成し、プロジェクトのクラスの参照設定にWiimoteLibを追加し、次の初期化コードを書き足してください。

コード4-11　C# Form1.csに以下の部分を追加

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WiimoteLib;  //WiimoteLibの使用を宣言

namespace WindowsFormsApplication1 { //プロジェクト名によって異なる
 public partial class Form1 : Form {

Wiimote wm = new Wiimote();   //Wiimoteの宣言
int count=0;                  //カウントの宣言

  public Form1() {
   InitializeComponent();
        wm.Connect();     //WiiRemoteへ接続
  }
 }
}
```

コード4-12　C++ Form1.hに以下の部分を追加

```
#pragma once
namespace WLCLED { //プロジェクト名によって異なる

  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;

  using namespace WiimoteLib;     //WiimoteLibの使用を宣言
<略>
  public ref class Form1 : public System::Windows::Forms::Form
  {
  public: Wiimote^ wm;  //Wiimoteオブジェクトwmの宣言
```

次ページにつづく

```
  public: int count;     //LEDカウント用の変数countの宣言
  public:
    Form1(void)
    {
      wm = gcnew Wiimote(); //Wiimoteインスタンスの作成
      InitializeComponent();
      //
      //TODO: ここにコンストラクタ コードを追加します
      //
      wm->Connect();     //WiiRemoteへ接続
    }

  protected:
<略>
```

確認のために、ここでWiiRemoteのBluetooth接続を行い、「F5」キーで実行を試してみることをお勧めします。コンパイルエラーやWiimoteLib.dllの追加を忘れるなど、この段階でミスはチェックしておきましょう。実行しても、ただの空白のフォームが表示されるだけの状態ですが、確認は大事です。今後も、この初期化コード作成は何度も繰り返すので、カラダで覚えてしまいましょう。

POINT ▸▸▸ C++では、int型の変数countや、Wiimoteオブジェクトを格納するwmの宣言を、Form1のインスタンスとは別に行う必要があります。そうしなければ他のメソッドから扱うことができません。今後、細かいところでC#と違いが出てくるので、注意してください（興味のある人は、わざと間違えてみるのも勉強になっていいかもしれませんが）。

LEDカウントアップボタンの作成

次は、C#/C++で共通の作業です。先ほどのバイブレーターの例と同様にフォーム「Form1」にボタンを貼り付けてください。ボタンを押すたび表示が変わる仕組みも取り入れるので、ボタンは少し大きめ、横長にしておくとよいでしょう。

図 4-29　フォームに大きめなボタンを配置 [C#]

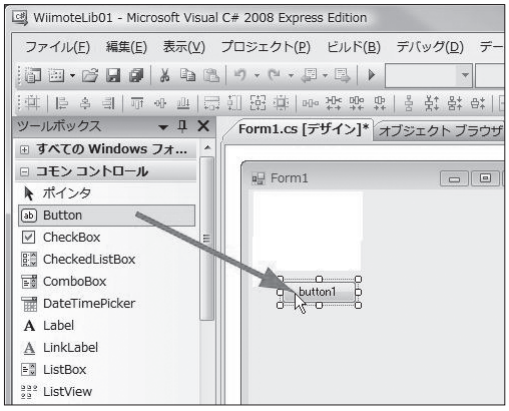
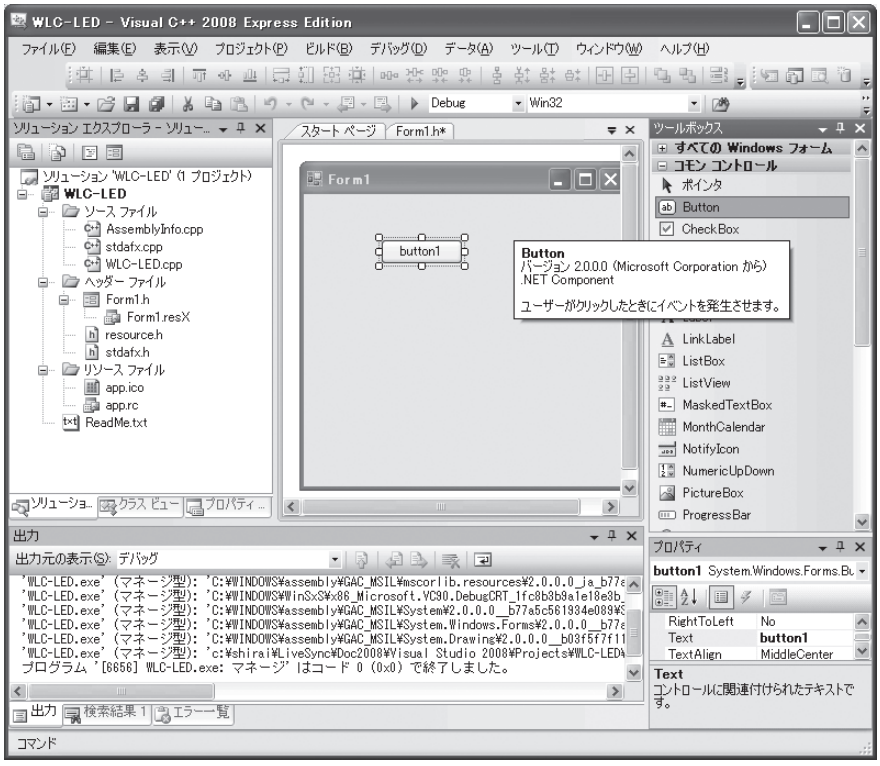


図 4-30　フォームに大きめなボタンを配置 [C++]



貼り付けた「button1」をダブルリックして、次のコードを追加します。

コード4-13 C# ボタンクリック時の処理を追加

```
private void button1_Click(object sender, EventArgs e) {
   this.button1.Text ="wm.SetLEDs("+ count+") を表示中";
   this.wm.SetLEDs(count);
   count++;
}
```

コード4-14 C++ ボタンクリック時の処理を追加

```
private: System::Void button1_Click(System::Object^ sender,
    System::EventArgs^ e) {
        button1->Text = "wm->SetLEDs("+ count +")を表示中";
        wm->SetLEDs(count);
        count++;
}
```

以上でコーディングは終了です。たったこれだけのプログラムでWiiRemoteのLED制御が可能になります。

コード4-15 C# 完成したForm1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WiimoteLib;

namespace WL_LED
{
    public partial class Form1 : Form
    {
        Wiimote wm = new Wiimote();
        int count = 0;
        public Form1()
        {
            InitializeComponent();
```

次ページにつづく

```
            wm.Connect();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            this.button1.Text = "wm.SetLEDs(" + count + ")を表示中";
            wm.SetLEDs(count);
            count++;
        }
    }
}
```

コード4-16　C++ 完成したForm1.h

```
#pragma once
namespace WLCLED { //作成したプロジェクト名、自由。
  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;
  using namespace WiimoteLib;
  /// <summary>
<略>
  /// </summary>
  public ref class Form1 : public System::Windows::Forms::Form
  {
  public: Wiimote^ wm; //Wiimoteオブジェクトwmの宣言
  public: int count;    //LEDカウント用の変数countの宣言
  public:
    Form1(void)
    {
     wm = gcnew Wiimote();
     InitializeComponent();
      //
      //TODO: ここにコンストラクタ コードを追加します
      //
      wm->Connect();    //WiiRemoteへ接続
    }
  protected:
    /// <summary>
    /// 使用中のリソースをすべてクリーンアップします。
    /// </summary>
   ~Form1()
    {
```

次ページにつづく

```
    if (components)
     {
      delete components;
     }
    }
     private: System::Windows::Forms::Button^  button1;
     protected:
<略>
#pragma endregion
private: System::Void button1_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
     button1->Text = "wm->SetLEDs("+ count +")を表示中";
     wm->SetLEDs(count);
     count++;
    }
   };
}
```

若干C++のほうがコードが長くなりますが、自動で追加された以外の箇所の意味合いはC#でもC++でも、ほぼ同じであることがわかります。

実行してみよう

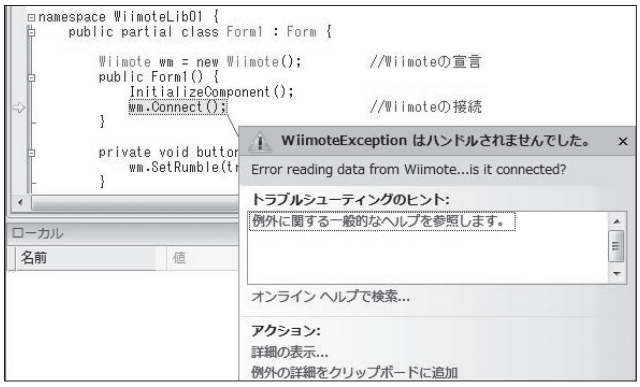
それでは、さっそく実行してみましょう。

Visual C#/C++の「F5」キーを押して実行してください。実行すると、図4-31のようなアプリケーションが起動します。

図4-31　大きなボタンが1つだけのフォームが表示される

もし図4-32のようなエラーが発生する場合はWiiRemoteが正しく接続されているか確認してください。

図 4-32　Bluetooth 接続を忘れるとエラーになる



フォームに表示されるボタンをクリックしていくと、WiiRemote 下部の LED が次々と光っていきます。

図 4-33　フォームに文字が表示される

図 4-34　WiiRemote 下部の LED

「×」をクリックして終了します。

解説：LED の点灯制御

LED の点灯と消灯はこの API を利用します。

```
SetLEDs(int32 leds);
```

この関数の引数「leds」にint32形式の数値を入れることで、対応するLEDが変化します。このプログラムでは変数countの値を入れ「SetLEDs(count);」としています。Formに配置されたボタンをクリックすると、count値が+1されていきます。

図4-35　LEDの点灯

引数は整数(int32形式)で与えますが、これはWiiRemoteを逆さまにした状態の各LEDを4ビットの2進数で表現して、各ビットを0から15まで足していったものです。2進数に馴染みがない方のために、表で表現してみました。

表4-3　LEDの点灯で学ぶ2進数―10進数対応表（●が消灯、○が点灯です）

10進数(int)	LED4	LED3	LED2	LED1
0	●	●	●	●
1	●	●	●	○
2	●	●	○	●
3	●	●	○	○
4	●	○	●	●
5	●	○	●	○
6	●	○	○	●
7	●	○	○	○
8	○	●	●	●
9	○	●	●	○
10	○	●	○	●
11	○	●	○	○
12	○	○	●	●
13	○	○	●	○
14	○	○	○	●
15	○	○	○	○

フォーム上のボタンを押し続けて、countに10進数の16になると、LED0～LED4の桁はそれぞれ0になりLEDはすべて消えますが、それ以上の値（17,18,19,...）が入っても、また下位ビットに値が入るので、LEDはカウントアップし続けます。

プログラムによっては2進数→10進数ではなく、個々のLEDを指定して光らせたいときもあるでしょう。そういったときは、関数フォーマットが異なる以下の形式を利用します。

```
SetLEDs(bool led1, bool led2, bool led3, bool led4);
```

同じ関数でも、引数を4つ指定することで、各々のLEDを制御することが可能になります（プログラミング用語では、このような関数の利用の仕方をオーバーロードといいます）。

ヘルプファイルを活用しよう

上記の「SetLEDs」のようなWiimoteLibに実装されているAPI関数それぞれの機能は、WiimoteLibの「docs」フォルダにあるヘルプファイル「WiimoteLib.chm」を参照することで探すことができます。

たとえば、このAPI関数の場合は以下のように記載されています。

```
<Wiimote.SetLEDs Method>Overload List

Name             Description
SetLEDs(Int32)   Set the LEDs on the Wiimote
SetLEDs(Boolean  Boolean Boolean   Boolean)
                 Set the LEDs on the Wiimote

See Also Wiimote Class Wiimote Members WiimoteLib Namespace
```

つまり関数「SetLEDs()」には、今回のようにInt32の値1つで指定する方式と、Booleanつまり点灯するかどうかの真偽(true/false)の4つで指定する方法の2種類が用意されているということです（どちらも同じ結果ではあるのですが）。

このようにして、WiimoteLibなどのAPIを作った人は便利にアクセスできるように、たくさんの気の利いた関数を開発しているということですね。下の「See Also」には所属しているクラスやメンバー関数などへのリンクがあります。

わからないことがあったり「こんな機能ないかな？」と思ったときは、このヘルプファイルを活用しましょう。このヘルプファイルはソースコードから自動生成されているようですが、検索機能も備えており、C#とVBのコードも含まれていて、勉強になります。

以上でLEDの制御は終わります。非常にシンプルですが、アイディア次第でいろいろなことができるので、ぜひ発想を豊かにして何に使えるか考えてみてください。また、その表現に合わせた便利な出力用関数も作ってみるとよいでしょう。

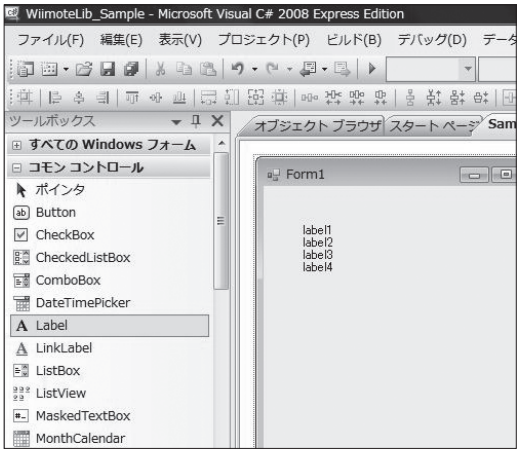
4.7 ボタンイベントの取得

次のステップではWiiRemoteのボタン入力について学びます。ボタンのON/OFFを取得して、フォームに表示するシンプルなプログラムを作成します。

ラベルの作成

先ほどまでのプログラムと同様に、新しいプロジェクトを作成し、参照設定にWiimoteLibを追加して準備ができたら、「Form1」にラベル（Labelコントロール）を4つ貼り付けてください。

図4-36　新しいプロジェクトにラベルを4つ配置



このラベルの文字は、後ほどプログラム側から書き換えるので設定は不要です。

ボタン入力に対応してラベルの表示を変える

WiiRemoteのボタンを押したときに発生するイベントを利用して、このラベルを表示する値を変更することで、現在のボタン入力の状態を表示するという設計でプログラムを作っていきます。

Form1.cs（C#）もしくはForm1.h（C++）に以下の部分を追加します。

コード4-17　C# ボタン入力に対応させる(Form1.cs)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WiimoteLib;     //WimoteLibの使用を宣言

namespace WiimoteLib_Sample  //作成したプロジェクト名
{
  public partial class Form1 : Form
  {
    Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
    public Form1()
    {
     Control.CheckForIllegalCrossThreadCalls = false; //おまじない
     InitializeComponent();
     wm.WiimoteChanged += wm_WiimoteChanged;  //イベント関数の登録
     wm.Connect();                            //WiiRemoteと接続
    }
    //Wiiリモコンの値が変化したときに呼ばれる関数
    void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args)
    {
     WiimoteState ws = args.WiimoteState;     //WiimoteStateの値を取得
     this.DrawForms(ws);                      //フォーム描写関数へ
    }
    //フォーム描写関数
    public void DrawForms(WiimoteState ws)
    {
```

次ページにつづく

```
      this.label1.Text = "Button A:" + (ws.ButtonState.A);  //ボタンA
      this.label2.Text = "Button B:" + (ws.ButtonState.B);  //ボタンB
      this.label3.Text = "Button 1:" + (ws.ButtonState.One);//ボタン1
      this.label4.Text = "Button 2:" + (ws.ButtonState.Two);//ボタン2
     }
   }
}
```

コード4-18 C++ ボタン入力に対応させる(Form1.h)

```
#pragma once
namespace WiimoteLib_Sample {      //作成したプロジェクト名
  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;
  using namespace WiimoteLib;  //WimoteLibの使用を宣言
        /// <summary>
<略>
        /// </summary>
  public ref class Form1 : public System::Windows::Forms::Form
  {
   public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
   public:
   Form1(void)
    {
     Control::CheckForIllegalCrossThreadCalls = false; //おまじない
     wm = gcnew Wiimote();  //Wiimoteクラスを作成
     InitializeComponent();
     //イベント関数の登録
     wm->WiimoteChanged +=
     gcnew System::EventHandler<WiimoteChangedEventArgs^>(
      this, &Form1::wm_WiimoteChanged);
     wm->SetReportType(InputReport::Buttons, true); //レポートタイプの設定
     wm->Connect(); //WiiRemoteと接続
    }
  public:
  void wm_WiimoteChanged(Object^
    sender,WiimoteLib::WiimoteChangedEventArgs^ args){
     WiimoteState^ ws;
     ws = args->WiimoteState;
     this->DrawForms(ws);
    }
```

次ページにつづく

```
  public: void DrawForms(WiimoteState^ ws) {
     this->label1->Text = "Button A:" + (ws->ButtonState.A);
     this->label2->Text = "Button B:" + (ws->ButtonState.B);
     this->label3->Text = "Button 1:" + (ws->ButtonState.One);
     this->label4->Text = "Button 2:" + (ws->ButtonState.Two);
  }
 protected:
<略>
```

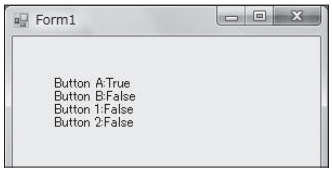
実行してみよう

さて、実行してみましょう。まずお使いの Bluetooth スタックから、WiiRemote を Bluetooth で PC に接続します。接続が確認できたら、Visual C#/C++ 上で「F5」キーを押して実行します。コンパイルがとおり、正しく実行されると、図 4-38 のようなアプリケーションが起動します。もしここでエラーが発生する場合、そのほとんどが Bluetooth 接続がうまく接続されていないケースです。WiiRemote が正しく接続されているか確認してください。

図 4-37　実行直後、ラベルの表示に注目

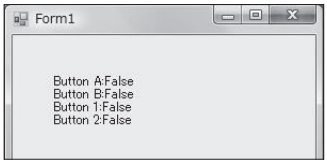
WiiRemote の A ボタンを押し続けます。

図 4-38　ラベルの表示が変わる



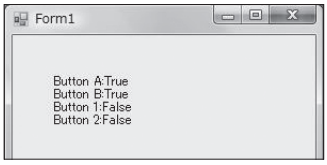
フォーム上の「Label1」の箇所に「Button A: True」と表示されれば成功です。さらにWiiRemoteのすべてのボタンから手を離したとき、フォーム上のボタンのステータスを表す表示がすべて「False」になれば成功です。

図4-39　ボタンから手を離すと、すべてFalseになる



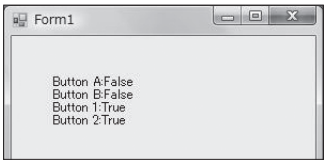
WiiRemoteのAボタンとBボタンを同時に押します。

図4-40　A、Bが Trueになる



最後に、WiiRemoteの1ボタンと2ボタンを同時に押します。

図4-41　1、2がTrueになる



一通りの実験が終わったら、マウスで「x」をクリックするか、キーボードから「Alt+F4」を押して、プログラムを終了させます。

解説：ボタンイベントの取得

WiiリモコンのボタンのON/OFFによってFormのラベルを変化させています。FalseがOFF（手を離した状態）、TrueがON（押下）に対応しています。以下、利用したAPI関数を解説します。

イベント関数の登録

WiiRemote のボタンが押された、加速度センサーの値が変わったなど、状態に変化があったときに呼ばれる関数を Form 内で登録しています。これをプログラミング用語で「コールバック関数」といい、関数名を決められた方法で登録することで、そのイベントが発生したときに自動的にその関数が実行されるようになります。

コード 4-19　C# イベント関数の登録

```
wm.WiimoteChanged += wm_WiimoteChanged;
```

コード 4-20　C++ イベント関数の登録

```
wm->WiimoteChanged +=
            gcnew System::EventHandler<WiimoteChangedEventArgs^>(
            this, &Form1::wm_WiimoteChanged);
```

ここでは、WiiRemote のボタン状態に変化があった場合「wm_WiimoteChanged」という関数が呼ばれるように設定しています。カッコや引数などはあらかじめ規定された形式に沿っているので、関数名を渡すだけでだけでよいのです（C++ のコードが少し長いのはそのためです）。

WiimoteState の値を取得

ここで Wii リモコンのステータス（状態）を ws という名前の " イレモノ " に取り込んでいます。正式にはここで使っている WiimoteState はクラスですが、その名前から想像できるように、イベントで発生したボタンなどの値が取り込まれます。

コード 4-21　C# WiimoteState の値を取得

```
WiimoteState ws = args.WiimoteState;
```

コード 4-22　C++ WiimoteState の値を取得

```
WiimoteState^ ws;
      ws = args->WiimoteState;
```

以下、使い方を見てみましょう。

コード 4-23　C# ボタンの状態を取得

```
this.label1.Text   = "Button A: " + (ws.ButtonState.A);
```

コード 4-23　C++ ボタンの状態を取得

```
this->label1->Text = "Button A:" + (ws->ButtonState.A);
```

WiiRemote の A ボタンの値を label1 に表示しています。ボタンが押されていたら、True を表示します。ボタンが離されていたら、False を表示します。実際には「ws.ButtonState.A」が意味する値は True か False という真偽の値ですが、左側が「label1.Text」なので自動的に文字列に変換されています（.ToString() する必要はありません）。

同様に「ws.ButtonState.b」などとすることで WiiRemote の B ボタン、その他すべてのボタンの状態を取得することができます。

どうでしょう？ とっても簡単ですね！ このイベントのコールバック関数でステートを取得する方法は他のいろいろな入力に応用できます。しかし、この完成したコールバックの仕組みを簡単に利用できる背景には、さまざまな複雑なプログラミングの内側の技術があります。実はフォームのコンストラクタで、1 つだけ、おまじないをしていました。

コード 4-25　C# おまじない

```
Control.CheckForIllegalCrossThreadCalls  = false;
```

コード 4-26　C++ おまじない

```
Control::CheckForIllegalCrossThreadCalls = false;
```

余裕のある人は、この行をコメントアウトして、実行してみてください。なぜか実行時にボタンを押すとエラーが出てしまいます。これは、マルチスレッド（複数の処理を並行で進める仕組み）に関わる問題です。上のボタンの状態を読み込む WiimoteLib のスレッドと、フォームの書き換えを行うスレッドがそれぞれ異なるので「スレッドセーフでない」つまり、複数のスレッドにおける処理の順序などが保証できないため実行時エラーになってしまいます。

回避する方法として、WiimoteLib の公式 HP で紹介されている、次のような方法があります。

WiimoteState のメンバー

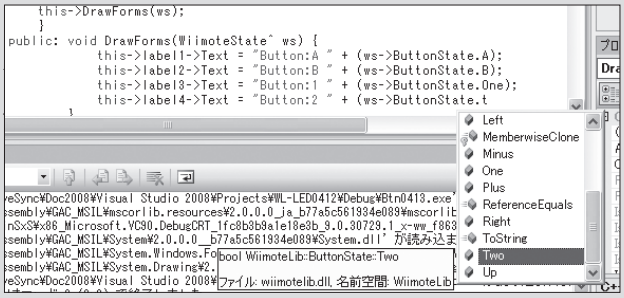
ここで WiimoteState で参照できるメンバーを表で紹介しておきます。

表 4-4　WiimoteState のメンバー（WiimoteLib ver.1.7）

名称	解説
AccelCalibrationInfo	現在の加速度センサーのキャリブレーション情報
AccelState	現在の加速度センサーの状態
BalanceBoardState	現在の WiiFit バランスボードの状態
Battery	算出された現在のバッテリーレベル
BatteryRaw	現在のバッテリーレベルの計算前の値（生値）
ButtonState	現在のボタンの状態
ClassicControllerState	現在の拡張クラシックコントローラーの状態
DrumsState	現在の拡張ドラムコントローラーの状態
Extension	拡張コントローラーが挿入されているか
ExtensionType	拡張コントローラーが挿入されている場合その種類
GuitarState	現在の GuitarHero 拡張ギターコントローラーの状態
IRState	現在の赤外線センサーの状態
LEDState	現在の LED の状態
NunchukState	現在の拡張ヌンチャクコントローラーの状態
Rumble	現在のバイブレーターの状態

WiimoteLib には実にさまざまな拡張コントローラーが実装されており、このメンバーから状態を取得できることがわかります。これらの値やメソッドなどは、Visual Studio の Intellisense 機能を使ってどんどん効率化していきましょう。先ほどのボタンの例なども、「ws.ButtonState.」と「.」を押した瞬間には正しい標記の選択肢が現れます。ボタンの名称などはいちいち覚えてられませんので、非常に便利です。なお Intellisense は Ctrl+Space でいつでもどこでも呼び出せます。すべてのグローバルなオブジェクトが表示されます。辞書代わりに使うとよいでしょう。

図 4-42　Intellisense による補完機能を使いこなそう



コード 4-27　C# Invoke(),delegate() を使う方法

```
//Wiiリモコンの値が変化したときに呼ばれる関数
void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args){
  WiimoteState ws = args.WiimoteState;     //WiimoteStateの値を取得
  if (this.IsHandleCreated) {
    this.Invoke( (MethodInvoker)delegate() {
    this.DrawForms( ws );              //フォーム描写関数へ
  });
 }
}
```

このように、Invoke() というメソッドを使う方法もありますが、ちょっと初心者には不明瞭な書き方です。何が起きていて、どんなリスクがあるか（フォームの書き換えがスレッドセーフでなく上書きされる）ということがわかっているなら、

```
Control.CheckForIllegalCrossThreadCalls=false;
```

として、「不正なスレッド間コールのチェックをしない」と宣言する方法もあるので、本章では「おまじない」として、以後この方法を採用することにします。

4.8 ランチャーを作る

さて、これまでWiimoteLibを使って、基本的な入出力を学んできました。このあたりで、実用的なプログラムの例として「ランチャー」を作成してみましょう。ボタンを押すたびに、Windowsのアクセサリ「メモ帳」や「電卓」など外部プログラムが起動するプログラムです。

外部プログラムの起動

前節の「ボタンイベントの取得」と基本は同じです。WiimoteLib の宣言を行い、コールバック関数内で外部プログラムを起動したり、アプリケーション自身を終了させたりします。

新しいプロジェクトを作成し、WiimoteLib を参照に追加し、Form1 を右クリックしてコードを表示し、下のコードを記述します。前節のプログラムの改造から始めてもよいでしょう。

コード 4-28　C# ボタンイベントで外部プログラムを起動する (Form1.cs)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using WiimoteLib;    //WimoteLibの使用を宣言

namespace WL_Launcher {
    public partial class Form1 : Form {
         Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
        public Form1() {
             Control.CheckForIllegalCrossThreadCalls = false; //おまじない
            InitializeComponent();
             wm.WiimoteChanged += wm_WiimoteChanged;  //イベント関数の登録
             wm.Connect();                             //WiiRemoteと接続
        }
         //Wiiリモコンの値が変化したときに呼ばれる関数
  void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args)
         {
             WiimoteState ws = args.WiimoteState;    //WiimoteStateの値を取得
             //Aボタンが押されたらメモ帳を起動
             if (ws.ButtonState.A == true) {
                 System.Diagnostics.Process.Start("notepad.exe");
             }
             //Bボタンが押されたら電卓を起動
             if (ws.ButtonState.B == true) {
                 System.Diagnostics.Process.Start("calc.exe");
             }
             //HOMEボタンが押されたらこのアプリを終了
             if (ws.ButtonState.Home == true) {
                 Environment.Exit(0);
```

次ページにつづく

```
            }
        }
    }
}
```

コード 4-29 C++ ボタン入力に対応させる(Form1.h)

```
#pragma once

namespace WLCLauncher {
  using namespace System;
  using namespace System::ComponentModel;
  using namespace System::Collections;
  using namespace System::Windows::Forms;
  using namespace System::Data;
  using namespace System::Drawing;
  using namespace WiimoteLib;  //WimoteLibの使用を宣言

  public ref class Form1 : public System::Windows::Forms::Form
  {
  public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
  public:
   Form1(void){
     Control::CheckForIllegalCrossThreadCalls = false; //おまじない
     wm = gcnew Wiimote();  //Wiimoteクラスを作成
     InitializeComponent();
     //イベント関数の登録
     wm->WiimoteChanged +=
       gcnew System::EventHandler<WiimoteChangedEventArgs^>(
         this, &Form1::wm_WiimoteChanged);
     wm->SetReportType(InputReport::Buttons, true); //レポートタイプの設定
     wm->Connect(); //WiiRemoteと接続
    }
  public:
void wm_WiimoteChanged(Object^ sender,WiimoteLib::
                                  WiimoteChangedEventArgs^args){
      WiimoteState^ ws;
      ws = args->WiimoteState;
      //Aボタンが押されたらメモ帳を起動
      if (ws->ButtonState.A) {
        System::Diagnostics::Process::Start("notepad.exe");
      }
      //Bボタンが押されたら電卓を起動
      if (ws->ButtonState.B) {
        System::Diagnostics::Process::Start("calc.exe");
```

次ページにつづく

```
        }
        //HOMEボタンが押されたらこのアプリを終了
        if (ws->ButtonState.Home) {
          Environment::Exit(0);
        }
      }
<以下略>
```

実行してみよう

まずは、いつも通りWiiRemoteを接続してください。そしてVisual Studioの「F5」キーを押して、作成したプログラムを実行します。もしここでエラーが発生する場合は、WiiRemoteが正しく接続されているか確認してください。

WiiRemoteのAボタンを押すとメモ帳が起動し、Bボタンを押すと電卓が起動します。WiiRemoteのBボタンを数回押すと、押した回数だけ、電卓が起動します。

図4-43　Bボタンを押した回数だけ電卓が起動する

WiiRemoteのHomeボタンを押すと、ランチャープログラムが終了します。

このプログラムの「notepad.exe」や「calc.exe」を好きな外部プログラムに書き換えれば、何でも起動できるというわけですね。なんだかいろいろなことができそうです。楽しくなってきませんか？

解説：ボタンイベントによるアプリ起動

WiiRemoteのボタンが押されることによって設定したアプリケーションが起動するためには、先ほどのプログラムのDrawForm()で処理したような、ラベルのテキストを書き換える代わりに、.NETで用意されている仕組みを利用して、外部プログラムを起動します。

コード4-30　C# ボタンが押されたら外部プログラムを起動する

```
if (ws.ButtonState.A == true) {
     //Aボタンが押されたらメモ帳を起動
      System.Diagnostics.Process.Start("notepad.exe");
}
```

「System.Diagnostics.Process.Start()」はいろいろな応用が可能です。テキストファイルなどを指定すれば関連づけられたプログラムを使って開くことができます。詳しくはインターネットで公開されている.NET Frameworkクラスライブラリのマニュアルや「Process.Start」をキーワードに検索してみるとよいでしょう。

コード4-30はC#での記述ですが、C++/CLIでも全く違和感なく互換性が保たれています。C++でHomeボタンを押して終了する箇所のコードを見てみましょう。

コード4-31　C++ Homeボタンが押されたら外部プログラムを終了する

```
//HOMEボタンが押されたらこのアプリを終了
if (ws->ButtonState.Home) {
  Environment::Exit(0);
}
```

以上のように「Environment::Exit(0);」で、自分自身のアプリケーションを終了できます。

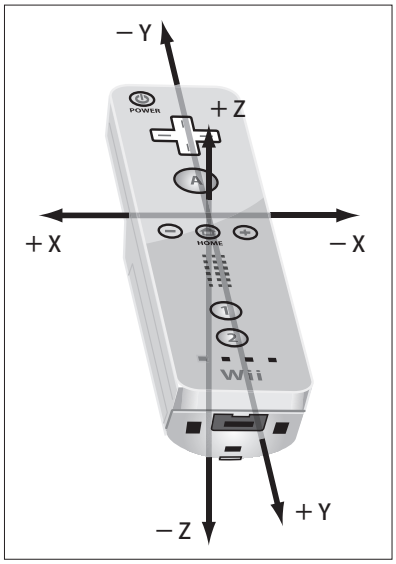
さて、こんな便利なコールを覚えると、ランチャーで起動したプログラムの終了などもやってみたくなると思います。この先の赤外線ポインタを用いてマウスを作成する例を学習してから、さらに高機能に改造してみるとよいでしょう。

4.9 加速度センサーを使う

加速度センサーについて

次は、WiiRemote の加速度センサーを使ったプログラミングを実験していきます。第２章で解説したとおり、WiiRemote には X 軸、Y 軸、Z 軸に対応した３軸の加速度センサーが内蔵されています。

図 4-44　加速度センサー



WiiRemote に内蔵された３軸マイクロ加速度センサーは、それぞれの軸に対して 8bit、つまり０～255 レベルの値を持ちます。このセクションでは、まず加速度センサーの値がとれるプログラムを作成し、その後、応用アプリケーションの開発を通して、加速度センサーの基本的な利用に挑戦します。

加速度センサーの値を表示

ここでは、まず手始めに、WiiRemoteの加速度を取得するプログラムを作成します。先ほどまでと同様、Visual Studio（C#/C++どちらでもかまいません！）で、新しいプロジェクトを作成し、参照に「WiimoteLib」を追加します。自動的に生成されているフォーム「Form1」に、ツールボックスからラベル（Label）を3つ貼り付けてください。ここに加速度センサーX、Y、Zのリアルタイム測定値を表示します。

続いてコーディングです。Form1の右クリックメニューの「コードの表示」でコードを表示し、次のプログラムのコメントアウトしている箇所を書き足していきます(自動生成されたコメント行は割愛しています)。また冒頭のusing句によるクラスの宣言ですが、最小限必要なもののみにしています。

コード4-32　C# 加速度センサーの値を表示する(Form1.cs)

```
  using System;
  using System.Windows.Forms;
  using WiimoteLib;    //WimoteLibの使用を宣言

namespace WL_Accel{
  public partial class Form1 : Form{
    Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
    public Form1(){
      Control.CheckForIllegalCrossThreadCalls = false;  //おまじない
      InitializeComponent();
      wm.WiimoteChanged += wm_WiimoteChanged;           //イベント関数の登録
      wm.SetReportType(InputReport.ButtonsAccel, true); //レポートタイプの設定
      wm.Connect();                                     //WiiRemoteと接続
    }
    //Wiiリモコンの値が変化したときに呼ばれる関数
    void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args) {
      WiimoteState ws = args.WiimoteState;     //WiimoteStateの値を取得
      this.DrawForms(ws); //フォーム描画関数へ
    }
    //フォーム描画関数
    public void DrawForms(WiimoteState ws)  {
      this.label1.Text = "X軸:" + (ws.AccelState.Values.X);
      this.label2.Text = "Y軸:" + (ws.AccelState.Values.Y);
      this.label3.Text = "Z軸:" + (ws.AccelState.Values.Z);
    }
  }
}
```

コード4-33 C++ 加速度センサーの値を表示する(Form1.h)

```
#pragma once
namespace WLCAccel {
  using namespace System;
  using namespace System::Windows::Forms;
  using namespace WiimoteLib;  //WimoteLibの使用を宣言

  public ref class Form1 : public System::Windows::Forms::Form{
  public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
  public:
    Form1(void) {
      Control::CheckForIllegalCrossThreadCalls = false; //おまじない
      wm = gcnew Wiimote();  //Wiimoteクラスを作成
      InitializeComponent();
      //イベント関数の登録
      wm->WiimoteChanged +=
        gcnew System::EventHandler<WiimoteChangedEventArgs^>(
        this, &Form1::wm_WiimoteChanged);
      wm->SetReportType(InputReport::ButtonsAccel, true); //レポートタイプの設定
      wm->Connect(); //WiiRemoteと接続
    }
  public:
  void wm_WiimoteChanged(Object^
    sender,WiimoteLib::WiimoteChangedEventArgs^ args){
      WiimoteState^ ws;
      ws = args->WiimoteState;
      this->DrawForms(ws);
    }
  public:
    void DrawForms(WiimoteState^ ws) {
      this->label1->Text = "X軸:" + (ws->AccelState.Values.X);
      this->label2->Text = "Y軸:" + (ws->AccelState.Values.Y);
      this->label3->Text = "Z軸:" + (ws->AccelState.Values.Z);
    }
<以下略>
```

実行してみよう

それでは実験してみましょう。まずWiiRemoteをお使いのBluetoothスタックで接続してください。

次にVisual Studio上で「F5」キーを押して実行してください。正しくプログラムが書かれてお

らず、エラーなどが出る場合はよく確認して、修正してください。

フォームが表示されたら、WiiRemoteを振り回してみてください。このとき、調子に乗って振り回しすぎて飛んでいくと危険なので、大振りするときは必ずストラップをしてください。

図4-45　加速度のX、Y、Zの値が変化する

フォームに張り付けた、加速度のX、Y、Zの値がすばやく動いていることがわかります。

POINT▶▶▶　起動時に3つの値がゼロのままだった場合、いったん作成したアプリケーションを終了させて、WiimoteLibのサンプル「WiimoteTest.exe」を実行してみてください。一度このサンプルを起動してから、今回作成したアプリケーションを起動すると値がとれることがあります(エラー処理や初期化を丁寧にしていないからかもしれません)。不具合があったときは試してみてください。

また、終了時に「Dispose」に関するエラーが出るときがありますが、これもいまのところ無視してかまいません。

図4-46　Disposeでエラーがあっても今は無視

解説：レポートタイプと加速度センサー

このサンプルでは、WiiRemote の３軸の加速度センサーのリアルタイム値を表示しました。

コード 4-33　C# レポートタイプの設定

```
wm.SetReportType(  InputReport.ButtonsAccel, true);
```

コード 4-34　C++ レポートタイプの設定

```
wm->SetReportType(InputReport::ButtonsAccel, true);
```

レポートタイプ、すなわちイベントが起きたときに報告するように WiiRemote にお願いする「種類」をここで設定しています。「ButtonsAccel」は加速度センサーとボタンイベントを取得しています。

コード 4-35　C# 加速度センサーの値取得

```
ws.AccelState.Values.X
```

コード 4-36　C++ 加速度センサーの値取得

```
ws->AccelState.Values.X
```

WiiRemote に内蔵された加速度センサー各軸の値を float で取得します。

コード 4-37　C# おまじない

```
Control.CheckForIllegalCrossThreadCalls = false;
```

コード 4-38　C++ おまじない

```
Control::CheckForIllegalCrossThreadCalls = false;
```

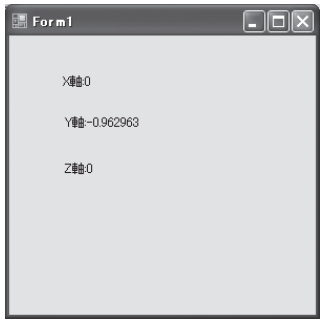
前回と同じく、別のスレッドからフォームを書き換えることを許可します。

レポートタイプに「ButtonsAccel」を指定しているので、この状態でボタンイベントなども取得できます。余力のある人は試してみましょう。

実際にどれだけの値が出力されるか実験してみましょう。ブンブン振ってみると、実測でだい

たい－５～＋５程度の値が計測されます。WiiRemoteを直立させると、X、Zなど２つの値はゼロになりますが、もう１つの軸、たとえばY軸には必ず－0.9～＋0.9程度の値が残ります。

図4-47　Y軸に検出されているのは……「重力加速度」



これは何でしょう……？ そうです。重力加速度です！ 普段は目に見えない重力加速度を目で見ることができます。

レポートタイプとは？

「レポートタイプ」とはWiiRemoteに問い合わせするときのモードのことで、このレポートタイプによって、WiiRemoteが返す返事が異なります。

WiimoteLib1.7ではInputReport内で以下のレポートタイプが定義されているようです。

- Buttons
- ButtonsAccel
- IRAccel
- ButtonsExtension
- ExtensionAccel
- IRExtensionAccel

レポートタイプは、データのフォーマットを設定する目的のほかにも、限りある通信帯域や処理速度を最適に設定する目的があるようです。上記のWiimoteLibで実装されているレポートタイプ以外にも、わかっているだけでも、ボタンのみの入出力から、加速度センサー３種、ヌンチャク付き６種、赤外線付きかどうかといったより高速でシンプルな入出力モードから、たくさんの値をやりとりするモードまで、各種揃っています。

また、赤外線センサーについても、最大４点まで扱えるモードに対して２点高速モードなど、隠しモード的なレポートタイプも存在するようです。

加速度センサーで作るWiiRemote太鼓

せっかくWiiRemoteの特徴の1つである加速度センサーの値を取得できるようになったので、「太鼓もどき」を作ってみましょう。太鼓と呼ぶには大げさかもしれませんが、加速度センサーに入力された強さが一定より強くなると……たとえば先ほどの実験で±5程度の値が測定できましたので、この値を超えた場合にWAVファイルを再生することにします。

先ほどの加速度センサーを使うプログラムの続きから始めるとよいでしょう。フォーム「Form1」のコードを表示して、以下のように追記します。

コード4-39　C# 振るとWAVファイルを再生(Form1.cs)

```
  using System;
  using System.Windows.Forms;
  using WiimoteLib;    //WimoteLibの使用を宣言
  using System.Media;  //System.Mediaの宣言

namespace WL_Taiko{
  public partial class Form1 : Form{
      Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
      string path = null;         //Wavファイル名
      SoundPlayer wavePlayer;     //SoundPlayerを宣言

      public Form1() {
        InitializeComponent();
        wm.WiimoteChanged += wm_WiimoteChanged;  //イベント関数の登録
        wm.SetReportType(InputReport.ButtonsAccel, true); //レポートタイプの設定
        wm.Connect();                            //WiiRemoteと接続
        path = @"C:¥WINDOWS¥Media¥chord.wav";    //再生するWAVファイルを指定
        wavePlayer = new SoundPlayer(path);      //プレイヤーにWAVファイルを渡す
      }
      //Wiiリモコンの値が変化したときに呼ばれる関数
      void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args) {
        WiimoteState ws = args.WiimoteState;    //WiimoteStateの値を取得
        //WAVファイルが読み込まれているか確認
        if (this.path != null) {
          float AX = Math.Abs(ws.AccelState.Values.X); //X軸の絶対値
          float AY = Math.Abs(ws.AccelState.Values.Y); //Y軸の絶対値
          float AZ = Math.Abs(ws.AccelState.Values.Z); //Z軸の絶対値
          //X,Y,Z軸の加速度センサーの絶対値の合計が5を超える時に、振ったと判定
          if ((AX+AY+AZ) >= 5) {
            wavePlayer.PlaySync();  //音を鳴らす
          }
```

次ページにつづく

```
        }
      }
    }
}
```

コード 4-40　C++ 振るとWAVファイルを再生 (Form1.h)

```
#pragma once
namespace WLCTaiko {
  using namespace System;
  using namespace System::Windows::Forms;
  using namespace WiimoteLib;  //WimoteLibの使用を宣言
  using namespace System::Media;  //System.Mediaの宣言

  public ref class Form1 : public System::Windows::Forms::Form{
    public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
    public: String^ path;          //WAVファイルパス格納用
    public: SoundPlayer^ wavePlayer;  //SoundPlayerを宣言
    public:
      Form1(void) {
        wm = gcnew Wiimote();  //Wiimoteクラスを作成
        InitializeComponent();
        //イベント関数の登録
        wm->WiimoteChanged +=
          gcnew System::EventHandler<WiimoteChangedEventArgs^>(
            this, &Form1::wm_WiimoteChanged);
        wm->SetReportType(InputReport::ButtonsAccel, true);
                                                  //レポートタイプの設定
        wm->Connect(); //WiiRemoteと接続
        path ="C:¥¥WINDOWS¥¥Media¥¥achord.wav";    //再生するWAVファイルを指定
        wavePlayer = gcnew SoundPlayer(path);      //プレイヤーにWAVファイルを渡す
      }
  public:
void wm_WiimoteChanged(Object^ sender,WiimoteLib::
                                  WiimoteChangedEventArgs^args){
        WiimoteState^ ws;
        float AX, AY, AZ;
        ws = args->WiimoteState;
        if (this->path!=nullptr) {
          AX = Math::Abs(ws->AccelState.Values.X); //X軸の絶対値
          AY = Math::Abs(ws->AccelState.Values.Y); //Y軸の絶対値
          AZ = Math::Abs(ws->AccelState.Values.Z); //Z軸の絶対値
          //X,Y,Z軸の加速度センサーの絶対値の合計が5を超える時に、振ったと判定
          if ((AX+AY+AZ)>=5) {
            wavePlayer->PlaySync();              //音を鳴らす
```

次ページにつづく

```
            }
          }
        }
<以下略>
```

Visual C#は［F6］、Visual C++は［F7］キーを押してコンパイルエラーがないことなどを確認したら、Bluetoothスタックから WiiRemoteを接続してください(切断されていなければそのまま続行してかまいません)。

Visual Studioの［F5］キーを押して実行してください。無事にエラーなく実行されると、フォームが表示されます。このフォームは今回使用しませんが、WiiRemoteを振ってみると、振りに合わせて、なんだか聴いたことのある音が鳴ります。

解説：WiiRemote太鼓

WiiRemoteの3軸の加速度センサーの値を取得して、指定した強さ以上の加速度を検出すると、指定したWAVファイルを鳴らします。

コード4-41　C# 再生するWAVファイルを指定

```
path = @"C:¥WINDOWS¥Media¥achord.wav";
```

コード4-42　C++ 再生するWAVファイルを指定

```
path ="C:¥¥WINDOWS¥¥Media¥¥achord.wav";
```

この「@」は「@-quoted string」といって、これが先頭に付いている文字列は「¥」(バックスラッシュ)を特殊な文字としてではなく、ファイルパスとして簡単に処理できます。C++にはそれに該当する表記がないようなので「¥¥」として「¥」を特別な1文字として扱っています。

さて、ここで再生するファイルを指定しています。鳴らしたいWAVファイルを指定してください。このプログラムではWindowsに用意されたWAVファイルとして、Windowsのシステムに最初から入っているWAVを指定しましたが、ご自身で用意された音楽や効果音を指定してもよいでしょう。

コード4-43　C# 絶対値

```
float AX = Math.Abs(ws.AccelState.Values.X); //X軸の絶対値
```

コード4-44 C++ 絶対値

```
AY = Math::Abs(ws->AccelState.Values.Y);      //Y軸の絶対値
```

WiiRemoteのX、Y、Z軸の値を取得し、その絶対値をとります。

コード4-45 C# 判定

```
//X,Y,Z軸の加速度センサーの絶対値の合計が5を超える時に、振ったと判定
if ((AX+AY+AZ)>= 5) {
        wavePlayer.PlaySync();
}
```

コード4-46 C++ 判定

```
if ((AX+AY+AZ)>=5) {
 wavePlayer->PlaySync(); //音を鳴らす
 }
```

X、Y、Z軸の加速度の絶対値の和が「5」を超えると音を鳴らすとは、いかにも簡単です。この「5」という値を、小さくすれば少ない動作で反応します。反対に増やせば、大きい動作で反応します。自分の好みの数字に置き換えてみて、調整してみてください。なお今回は、音の再生中にWiiRemoteを振っても反応しません(再生中にも音を連続再生したい場合はスレッド処理などを用いる必要があります)。

4.10 赤外線センサーを使う

加速度センサーをひと通り使いこなしたあとは、赤外線センサーに挑戦してみましょう。赤外線は人間の目で見ることができません。しかし最初に作成するプログラム「赤外線探知機」は、赤外線がWiiRemoteの視界に入ると、バイブレーターを鳴らすことができます。そして次に、最大4点の赤外線光源をカウントし、その結果をLEDに表示するプログラムを作ります。さらにそれ

を応用し、グラフィックスに組み込む基礎を学び、最後に赤外線センサーを使ったマウス操作プログラムをステップを追って開発していきます。目に見えない赤外線が、情報を伝えるメディアになることを体感してください！

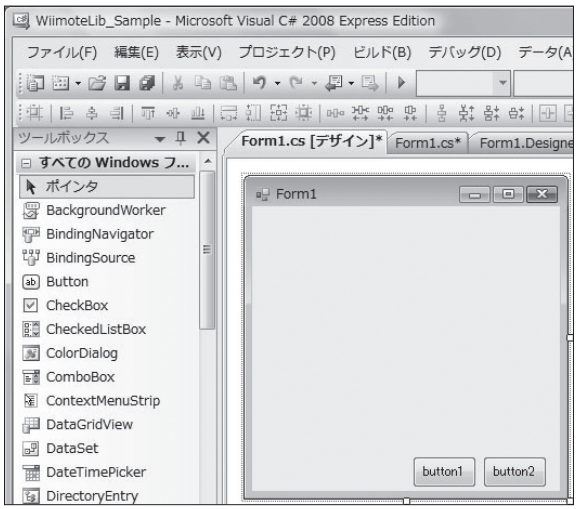
赤外線探知機

さっそくプログラミングを始めましょう。今回も前回に作成した加速度センサーのプログラムを改変してもよいのですが、新規プロジェクトで作るほうが勉強になってよいでしょう。

ソリューションを新しく作成する必要はありません。ソリューションエクスプローラーのソリューションを右クリックして［追加］→［新しいプロジェクト］で新しいプロジェクト名（ここでは「IR1」）を与えて、プロジェクトを追加します。できあがったプロジェクトを右クリックして「スタートアッププロジェクトに設定」を選択し、参照設定に WiimoteLib を追加します。他のプロジェクトのコードやフォームを間違えて編集しないよう、いったん開いているソースコードのウィンドウをすべて閉じます。これで準備はできあがりです。動作確認のために「F5」キーを押して実行してみてもよいでしょう。

このプロジェクトではまず、WiiRemote への「接続」「切断」ボタンを作成します。フォームにボタンを２つ貼り付けてください。

図 4-48　フォームにボタンを２つ配置する



貼り付けたら、button1のプロパティのTextを「接続」に、button2のプロパティのTextを「切断」に設定します。

図4-49 「button1」のtextプロパティを「接続」に

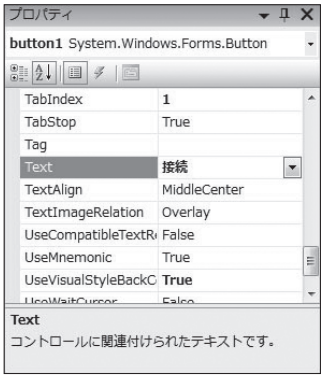
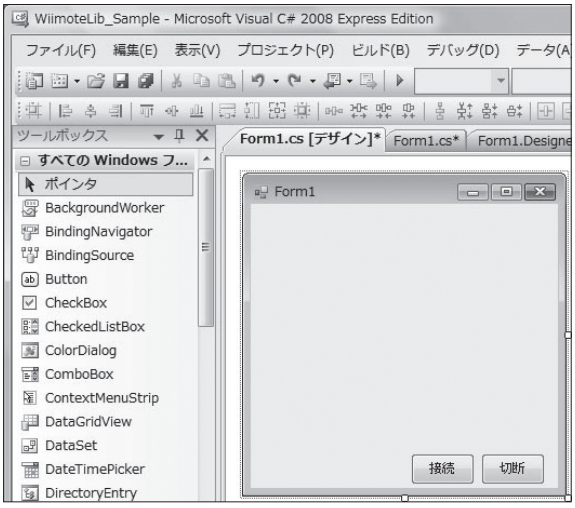


図4-50 ボタンの完成



フォーム「Form1」の右クリックメニューから「コードの表示」を選択し、コードを表示します。そして、宣言と接続時の処理、赤外線が見えたときのバイブレーターの処理を書き足しましょう。

コード4-47 C# 赤外線探知機(Form1.cs)

```
//不要なusing宣言は整理してかまいません
  using System;
  using System.Windows.Forms;
```

次ページにつづく

```
  using WiimoteLib;     //WimoteLibの使用を宣言

namespace IR1
{
  public partial class Form1 : Form{
    Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
    public Form1(){
      InitializeComponent();
      //他スレッドからのコントロール呼び出し許可
      Control.CheckForIllegalCrossThreadCalls = false;}
    //接続ボタンが押されたら
    private void button1_Click(object sender, EventArgs e) {
      wm.Connect();                              //Wiimoteの接続
      wm.WiimoteChanged += wm_WiimoteChanged;  //イベント関数の登録
      wm.SetReportType(InputReport.IRExtensionAccel, true);
                                                 //レポートタイプの設定
    }
    //切断ボタンが押されたら
    private void button2_Click(object sender, EventArgs e) {
      wm.WiimoteChanged -= wm_WiimoteChanged; //イベント関数の登録解除
      wm.Disconnect();  //Wiimote切断
      wm.Dispose();     //オブジェクトの破棄
    }
    //Wiiリモコンの値が変化する度に呼ばれる
    void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args)  {
      WiimoteState ws = args.WiimoteState;    //WiimoteStateの値を取得
      //もし赤外線を1つ発見したら
      if (ws.IRState.IRSensors[0].Found)  {
        wm.SetRumble(true);     //バイブレータON
      } else {
        wm.SetRumble(false);    //バイブレータOFF
      }
    }
  }
}
```

コード4-48　C++ 赤外線探知機(Form1.h)

```
#pragma once
namespace IR1 {
  using namespace System;
  using namespace System::Windows::Forms;
  using namespace WiimoteLib;  //WimoteLibの使用を宣言
  public ref class Form1 : public System::Windows::Forms::Form{
  public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
```

次ページにつづく

```
  public:
    Form1(void){
      wm = gcnew Wiimote();  //Wiimoteクラスを作成
      InitializeComponent();
      //他スレッドからのコントロール呼び出し許可
      Control::CheckForIllegalCrossThreadCalls = false;
    }
  protected:
    ~Form1()
<省略>
#pragma endregion
  //接続ボタンが押されたら
private: System::Void button1_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
    wm->Connect(); //WiiRemoteと接続
    wm->WiimoteChanged +=
      gcnew System::EventHandler<WiimoteChangedEventArgs^>(
        this, &Form1::wm_WiimoteChanged);
    wm->SetReportType(InputReport::IRExtensionAccel, true);
                                          //レポートタイプの設定
  }
    //切断ボタンが押されたら
private: System::Void button2_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
    wm->WiimoteChanged -=
      gcnew System::EventHandler<WiimoteChangedEventArgs^>(
         this, &Form1::wm_WiimoteChanged); //イベント関数の登録解除
      wm->Disconnect();  //Wiimote切断
      wm->Dispose();     //オブジェクトの破棄
  }

  //WiiRemoteの値が変化する度に呼ばれる
  public:
  void wm_WiimoteChanged(Object^
                   sender,WiimoteLib::WiimoteChangedEventArgs^ args){
      WiimoteState^ ws = args->WiimoteState; //WiimoteStateの値を取得
      //もし赤外線を１つ発見したら
      if (ws->IRState.IRSensors[0].Found) {
        wm->SetRumble(true);     //バイブレータON
      } else  {
        wm->SetRumble(false);    //バイブレータOFF
      }
    }
  };
}
```

実験：赤外線を見てみよう

さて、実行してみます。まず周囲に赤外線光源を用意してください。Wii本体付属のセンサーバーがあれば手っ取り早いのですが、ない場合は周りにある照明、太陽光などにあたりをつけて

ボタンを押しても何も起きないときは？

前述のコードをそのままコピーしていませんか？ ボタンを押したときの処理は、Form1の上にあるボタンをダブルクリックして、Windowsフォームデザイナが自動で生成したコードを使って書いていくのが確実です。もし単に前述のコードをコピーすると、接続ボタンを押しても、適切な処理にプログラムが流れていきません。

この仕組みに「どうして!?」と思った人は、プロジェクトの中にある「Form1.Designer.cs」をのぞいてみましょう。ここに「#region Windowsフォームデザイナで生成されたコード」というデフォルトで非表示になっているパートがあります。

コード4-49 C# Form1.Designer.csで自動生成されているコード

```
#region Windows フォーム デザイナで生成されたコード
…
this.button1.Click += new System.EventHandler(this.button1_Click_1);
…
this.button2.Click += new System.EventHandler(this.button2_Click);
```

ここには、GUIで作成したフォームについての位置や大きさなどのプロパティが記載されています。大事な部分は、ボタンを押したときのイベントの発生です。

```
this.button1.Click+=new System.EventHandler(this.button1_Click_1);
```

まさにWiiRemoteのイベントの追加と同じように、クリック時のイベントを追加しています。ただし、上の例では「button1_Click_1」という関数になっています。「_1」の部分は、他の既存の関数と名前が衝突しないよう、自動で生成されます。つまり、勝手に「button1_Click」という関数を書いていたとしても、ボタンを押したときのイベントとしてコールされることはありません。

「なんだ、わざわざGUIでダブルクリックしないといけないのか！」と思われるかもしれませんが、管理が面倒なイベント渡しなども自動で生成・管理してくれる、.NETスタイルの開発を「裏側まで理解して」使いこなす、というのがカッコイイのではないでしょうか。

みましょう。

Bluetoothスタックから WiiRemoteを接続して、Visual Studioの「F5」キーを押して実行してください。フォームが表示されたら、「接続」ボタンを押してください。WiiRemoteをセンサーバーなど赤外線光源に向けてください。センサーバーがない場合は、太陽、ハロゲンランプなどの熱源照明、ライターの火、テレビのリモコン、携帯電話の赤外線通信などに向けてみてください。

うまく検出できると、バイブレーターが振動します。いろいろな方向に向けてみましょう。普段は見えない赤外線ですが、身の回りにある様々なものに利用されていることに気がつくことでしょう。終了するには、バイブレーターが鳴っていない状態で「切断」ボタンを押します。

ライターやロウソクなどを用いて赤外線光源にする場合は、火事や火傷などに十分気をつけて実験を行ってください。また、テレビリモコンを用いる場合は、ボタンが押されたときにしか赤外線を送信しないので、何度か連打して確認を行うとよいでしょう。

解説:レポートの設定／赤外線4点検出

このプログラムの仕組みは単純です。コールバック関数を設定して、WiiRemoteの赤外線センサーが1つでも見つかったら、バイブレーターをONにします。

コード4-50 C# レポートタイプの設定

```
wm.SetReportType ( InputReport.IRExtensionAccel, true);
```

コード4-51 C++ レポートタイプの設定

```
wm->SetReportType(InputReport::IRExtensionAccel, true);
```

レポートタイプをIRExtensionAccel（赤外線＋拡張＋加速度）取得モードに設定します。このコールは必ずwm.Connect();の後に記述してください。wm.Connect();より前に記述すると赤外線センサーが正しく動作しません。

コード4-52 C# 赤外線の検出

```
ws.IRState.IRSensors[0].Found
```

コード4-53 C++ 赤外線の検出

```
ws->IRState.IRSensors[0].Found
```

このプロパティはTrue/Falseを返すので、if文を使って赤外線を検出できます。またWiimoteLibは、同時に4点まで赤外線光源を検出できます。個々の光源がインデックスのどの値にあたるのかを特定することはできませんが、IRSensors[3].Foundの値がTrueなら4つの赤外線光源が見えている、ということになります。

赤外線を数える

続いて、作成した基本的なプログラムを応用して、赤外線の個数を数えるプログラムに拡張します。WiimoteLibには同時に4点までの赤外線を計測することができます。ここまでのプログラムでは1点でも赤外線光源がセンサーの視界に入ると、バイブレーターが振動するようになっていましたが、青色LED（プレイヤーインジケーター）を使って、何点検出しているかを表示するプログラムを追加します。

コード4-54 C# 赤外線探知LED表示(Form1.cs, 抜粋)

```
void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args){
  WiimoteState ws = args.WiimoteState;  //WiimoteStateの値を取得
  //もし赤外線を1つ発見したら
  if (ws.IRState.IRSensors[0].Found)  {
     wm.SetRumble(true);     //バイブレータON
  }  else  {
     wm.SetRumble(false);    //バイブレータOFF
  }
  //検出された赤外線個数をWiiリモコンのLEDに表示する
  wm.SetLEDs(ws.IRState.IRSensors[0].Found, ws.IRState.IRSensors[1].Found,
             ws.IRState.IRSensors[2].Found, ws.IRState.IRSensors[3].Found);
}
```

コード4-55 C++ 赤外線探知LED表示(Form1.h, 抜粋)

```
public:
void wm_WiimoteChanged(Object^ sender,WiimoteLib::
                                  WiimoteChangedEventArgs^args){
```

次ページにつづく

```
 WiimoteState^ ws = args->WiimoteState;  //WiimoteStateの値を取得

  //もし赤外線を１つ発見したら
  if (ws->IRState.IRSensors[0].Found) {
    wm->SetRumble(true);     //バイブレータON
  } else {
    wm->SetRumble(false);    //バイブレータOFF
  }
  //検出された赤外線個数をWiiリモコンのLEDに表示する
  wm->SetLEDs(ws->IRState.IRSensors[0].Found,
  ws->IRState.IRSensors[1].Found,
  ws->IRState.IRSensors[2].Found, ws->IRState.IRSensors[3].Found );
}
```

ここでは先ほどLEDの点灯制御で使ったSetLEDs()関数のうち、4引数のものを使っています。WiiRemoteの赤外線センサーに複数の赤外線が発見されると、バイブレーターの振動とともにプレイヤーインジケーターの青色LEDを使って赤外線検出個数を表示します。

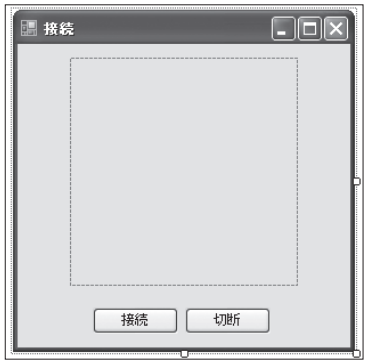
座標を描画

さて、赤外線光源の有無や、そのカウントができるようになったので、次は赤外線センサーによる光源座標の取得を行い、フォーム内にグラフィックス機能を使って描画します。先ほどのプロジェクトをそのまま改良して開発することにしましょう。

まずフォームのデザインを変更します。初めて使う新しいコントロールを配置します。「ツールボックス」の「コモンコントロール」から「PictureBox」をForm1に張り付けます。プロパティの「Size」を「200, 200」にします。他のボタンやフォームとのバランスをとって配置します。

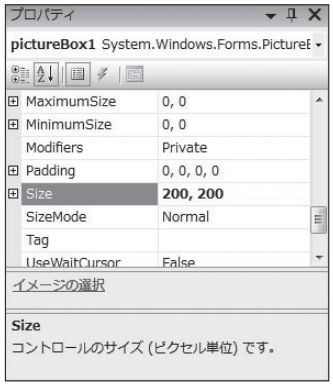
プログラムのほうは、まず冒頭のusing宣言で「System.Drawing」が宣言されていることを確認

図4-51　Form1に配置したPictureBoxとボタン



してください。初期化やボタンイベントはそのままで、WiiRemoteの状態が変化したときに呼ばれる関数「wm_WiimoteChanged()」とフォーム描画関数「DrawForms(ws)」に、描画のためのコードを追加します。

図4-52　PictureBoxのSizeプロパティ



コード4-56　C# 赤外線ポインタ描画(Form1.cs)

```
  using System;
  using System.Windows.Forms;
  using System.Drawing; //描画のために必要
  using WiimoteLib;     //WimoteLibの使用を宣言

  namespace IR4 {       //作成したプロジェクト名称
    public partial class Form1 Form {
      Wiimote wm = new Wiimote(); //Wiimoteクラスを作成
    public Form1) {
     InitializeComponent();
     //他スレッドからのコントロール呼び出し許可
     Control.CheckForIllegalCrossThreadCalls = false;
    }

    //WiiRemoteの状態が変化したときに呼ばれる関数
    void wm_WiimoteChanged(object sender, WiimoteChangedEventArgs args) {
      WiimoteState ws = args.WiimoteState;  //WiimoteStateの値を取得
      DrawForms(ws);                        //フォーム描写関数へ
    }
<略：ボタンイベント関係>
    //フォーム描写関数
    public void DrawForms(WiimoteState ws) {
      Graphics g =this.pictureBox1.CreateGraphics(); //グラフィックス取得
      g.Clear(Color.Black);                          //画面を黒色にクリア
      //もし赤外線を１つ発見したら
      if (ws.IRState.IRSensors[0].Found) {
        //赤色でマーカを描写
        g.FillEllipse(Brushes.Red,
        ws.IRState.IRSensors[0].Position.X * 200,
```

次ページにつづく

```
        ws.IRState.IRSensors[0].Position.Y * 200, 10, 10);
      }
      //もし赤外線を２つ発見したら
      if (ws.IRState.IRSensors[1].Found) {
        //青色でマーカを描写
        g.FillEllipse(Brushes.Blue,
        ws.IRState.IRSensors[1].Position.X * 200,
        ws.IRState.IRSensors[1].Position.Y * 200, 10, 10);
      }
      //もし赤外線を３つ発見したら
      if (ws.IRState.IRSensors[2].Found) {
        //黄色でマーカを描写
        g.FillEllipse(Brushes.Yellow,
        ws.IRState.IRSensors[2].Position.X * 200,
        ws.IRState.IRSensors[2].Position.Y * 200, 10, 10);
      }
      //もし赤外線を４つ発見したら
      if (ws.IRState.IRSensors[3].Found) {
        //緑色でマーカを描写
        g.FillEllipse(Brushes.Green,
        ws.IRState.IRSensors[3].Position.X * 200,
        ws.IRState.IRSensors[3].Position.Y * 200, 10, 10);
      }
      g.Dispose();//グラフィックスの解放
  }
 }
}
```

コード4-57　C++ 赤外線ポインタ描画(Form1.h)

```
#pragma once
namespace IR4 {  //作成したプロジェクト名称
 using namespace System::Windows::Forms;
 using namespace System::Drawing;
 using namespace WiimoteLib;  //WimoteLibの使用を宣言
 public ref class Form1 : public System::Windows::Forms::Form{
   public: Wiimote^ wm; //Wiimoteオブジェクトwmを作成
   public:
    Form1(void) {
      wm = gcnew Wiimote();  //Wiimoteクラスを作成
      InitializeComponent();
      //他スレッドからのコントロール呼び出し許可
      Control::CheckForIllegalCrossThreadCalls = false;
    }
   protected:
```

次ページにつづく

```
    /// <summary>
<略>
#pragma endregion
    //接続ボタンが押されたら
private: System::Void button1_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
      wm->Connect(); //WiiRemoteと接続
      wm->WiimoteChanged +=
        gcnew System::EventHandler<WiimoteChangedEventArgs^>(
          this, &Form1::wm_WiimoteChanged);
        //レポートタイプの設定
        wm->SetReportType(InputReport::IRExtensionAccel, true);
    }
    //切断ボタンが押されたら
private: System::Void button2_Click(System::Object^ sender,
                                    System::EventArgs^ e) {
      wm->WiimoteChanged -=
        gcnew System::EventHandler<WiimoteChangedEventArgs^>(
          this, &Form1::wm_WiimoteChanged); //イベント関数の登録解除
        wm->Disconnect();  //Wiimote切断
    }
    //WiiRemoteの値が変化する度に呼ばれる
    public:
  void wm_WiimoteChanged(Object^
                    sender,WiimoteLib::WiimoteChangedEventArgs^ args){
        WiimoteState^ ws = args->WiimoteState;  //WiimoteStateを取得
        DrawForms(ws);
    }
   public:
      void DrawForms(WiimoteState^ ws) {
        //グラフィックスを取得
        Graphics^ g = this->pictureBox1->CreateGraphics();
        g->Clear(Color::Black);//画面を黒色にクリア

        if (ws->IRState.IRSensors[0].Found) {  //赤外線を1つ発見したら
          //赤色でマーカを描写
          g->FillEllipse( Brushes::Red ,
            (float)ws->IRState.IRSensors[0].Position.X * 200.0f ,
  (float)ws->IRState.IRSensors[0].Position.Y * 200.0f , 10.0f , 10.0f );
        }
        if (ws->IRState.IRSensors[1].Found) {  //赤外線を2つ発見したら
          //青色でマーカを描写
          g->FillEllipse( Brushes::Blue ,
            (float)ws->IRState.IRSensors[1].Position.X * 200.0f ,
  (float)ws->IRState.IRSensors[1].Position.Y * 200.0f , 10.0f , 10.0f );
        }
```

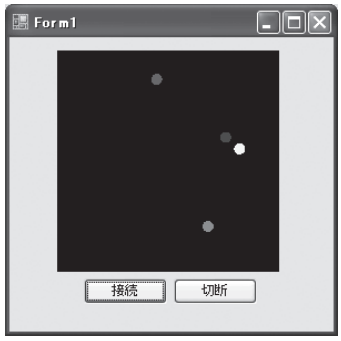
次ページにつづく

```
        if (ws->IRState.IRSensors[2].Found) {  //赤外線を3つ発見したら
          //黄色でマーカを描写
          g->FillEllipse( Brushes::Yellow ,
            (float)ws->IRState.IRSensors[2].Position.X * 200.0f ,
  (float)ws->IRState.IRSensors[2].Position.Y * 200.0f , 10.0f , 10.0f );
        }
        if (ws->IRState.IRSensors[3].Found) {  //赤外線を4つ発見したら
          //緑色でマーカを描写
          g->FillEllipse( Brushes::Green ,
            (float)ws->IRState.IRSensors[3].Position.X * 200.0f ,
  (float)ws->IRState.IRSensors[3].Position.Y * 200.0f , 10.0f , 10.0f );
        }
      }
  };
}
```

実験しよう

まず、赤外線光源を用意してBluetoothスタックからWiiRemoteを接続してください。Visual Studioから「F5」キーを押してプログラムを実行します。表示されたフォームの「接続」ボタンをクリックして、WiiRemoteをセンサーバーや電球などの赤外線光源に向けてください。

図4-53　赤外線に向けると4色のマーカーが動く



赤外線が検出されると図4-53のようにマーカー点が表示されます。マーカーの動きが激しすぎる場合は、WiiRemoteとセンサーバーとの距離を2～3mまで離してください。

なお、Wii本体付属の標準のセンサーバーには赤外線LEDが左右2グループしかありません（そもそもWiiRemoteに4点検出できる機能が存在するところが驚きです！）。複数の赤外線が見つからない場合は、日中（太陽に向けるなど）に窓の外に向けると複数の赤外線を検出できると思います。太陽の光を乱反射している様子などでも複数点を取得できることがありますが、赤外線光源同士が近すぎると1つのグループとして誤認識されノイズの原因になるで、ある程度安定して取得できる条件や距離を調べてみるのもよいでしょう。

図4-53は、とある店舗の天井に吊られている4個のハロゲンランプの様子です。終了する場合

は、切断ボタンを押してから終了させてください。

解説：赤外線座標の取得

このサンプルでは、WiiRemote の赤外線カメラの値を取得して、赤外線を発見したら画面に描画しています。

コード 4-58　C# 赤外線座標の取得

```
ws.IRState.IRSensors[0].Position.X
```

コード 4-58　C++ 赤外線座標の取得

```
ws->IRState.IRSensors[0].Position.X
```

赤外線の座標（x,y）の位置（Position）は、{0.0 ～ 1.0} の値域をとります。グラフィックスとの組み合わせも意外と簡単だったのではないでしょうか。本プログラムでは、その値に PictureBox の横幅として設定した 200×200 に合うように、200 を掛けて出力していますが、フォームの Size を変更したりして、お好みの画面デザインにしてみるとよいでしょう。

なお、WiiRemote の赤外線センサーは非常に高性能です。ここでは 4 点の検出を行っていますが、実際に赤外線光源座標が送られてくるスピードは非常に速いことに注目です。通常のビデオカメラが 1 秒に 15-30 回程度の撮影を行っているのに対し、WiiRemote は 200 回程度の座標取得処理を行っているようです。さすがゲーム用入力デバイス、速度が大切です！

本章では比較的初心者の読者に向けて、Visual Studio を使い、C# と C++ という複数の言語環境を使って .NET で開発された WiimoteLib を通して、基本的な WiiRemote プログラミングを学びました。WiimoteLib は現状最も完成度の高い API の 1 つで、非常に安定して動作します。.NET という環境から C# 専用ではないかと誤解されていますが、本書に向けて C++/CLI における解説を充実させました（おそらく世界初！です）。

WiimoteLib を使ったサンプルや具体的な開発例は第 8 章、第 9 章でも扱います。