C++とC#でまなぶWiiRemoteプログラミング

Version 平成21年5月17日

白井暁彦＋小坂崇之
shirai at mail.com
http://akihiko.shirai.as/projects/BookWii/

目 次

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#を同時に並列して扱います。

また、本章のC#.NETによるプログラミングサンプルは金沢工業専門高等学校の小坂崇之先生 (http://www.kosaka-lab.com/) の協力によるものです。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することができるようになっています。

0.1 プログラミング環境のセットアップ

このセクションでは、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。既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や、C++、C#のいずれかしか必要でない方は、必要なところだけ参照してください。

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

Visual C# 2008 Express Editionのセットアップについては、マイクロソフトの公式サイトに丁寧に解説されております。

http://www.microsoft.com/japan/msdn/vstudio/express/beginners/2008/vcsharp.aspx

Express EditionのWebインストールをダウンロードし、時計を表示するアプリケーションを実際に作ってみることができます。C#はしばらく使う予定はないけれども、インストールしてみよう、という方は是非試してみてください。

0.1.2 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、サンプル開発について解説します。既に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。

まずマイクロソフト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」をダウンロードすることができま

Visual C++ 2008 Express Edition

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

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

す。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」を読んでみるとすると良いでしょう。

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。

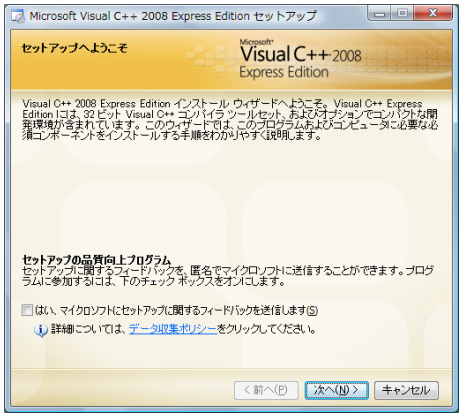


図 2: VC2008 インストール ウィザード (step1)

「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。

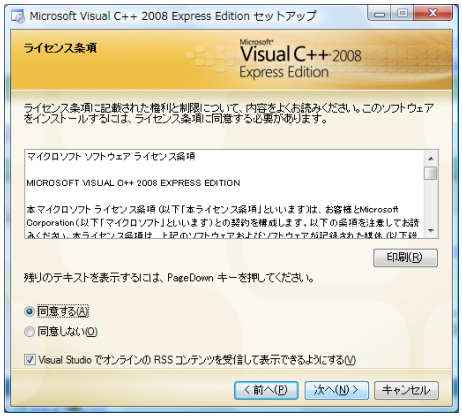


図 3: VC2008 インストール ウィザード (step2)

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（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）。「次へ」をクリックして進みます。

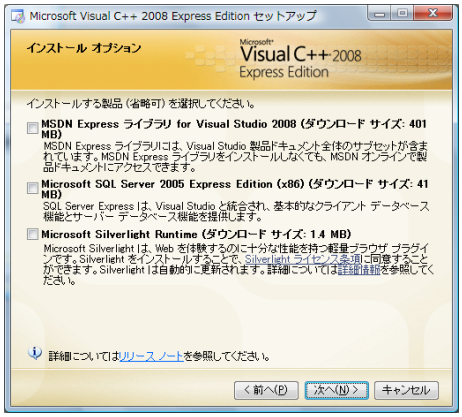


図 4: VC2008 インストール ウィザード (step3)

「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、オンライン版の方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良いでしょう。「次へ」をクリックして進みます。

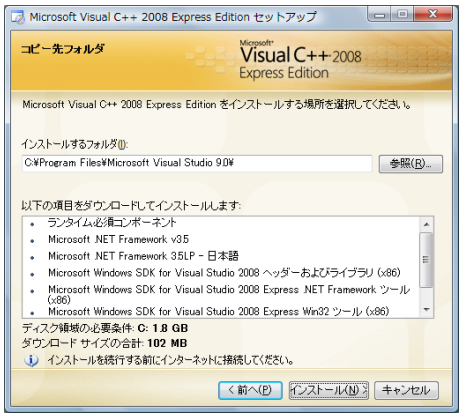


図 5: VC2008 インストール ウィザード (step4)

「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。ダウ

ンロードリストの中に「Windows SDK」が入っているのが助かります (以前の Express Edition では別途インストールする必要がありました)。「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。

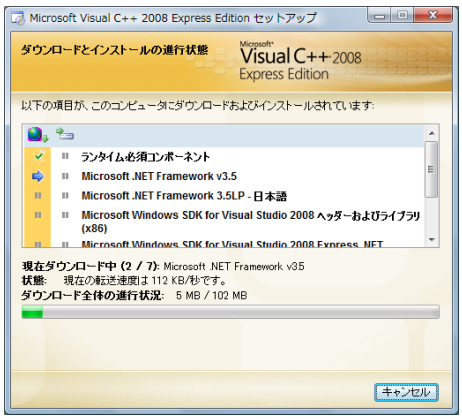


図 6: VC2008 インストール ウィザード (step5)

以上は WindowsXP シリーズにおける VC2008 セットアップの流れです (Windows Vista の場合は「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良いでしょう。

コラム：VC 混在環境を解決する Version Selector

もし、既に 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 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。

0.2 WiimoteLibの概要

WiimoteLib は Brian Peek(http://www.brianpeek.com/) による.NET 環境で利用できる API 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」(http://www.codeplex.com/) 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することができます。Version1.6.0.0 からは WiiRemote だけでなく Wii Fit バランスボードにも対応しています。

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、多言語間で問題なく利用できることがわかるでしょう。

C++言語にはさまざまな言語仕様が存在しますが、本章では特に.NET プログラミングとして扱いやすい「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ーにも、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、同じ基本的なプログラミングを C++と C#という 2 つの言語で解説するというわけです。

0.3 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、最も制限の緩いマイクロソフト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 (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)

0.4 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。

http://www.codeplex.com/WiimoteLib

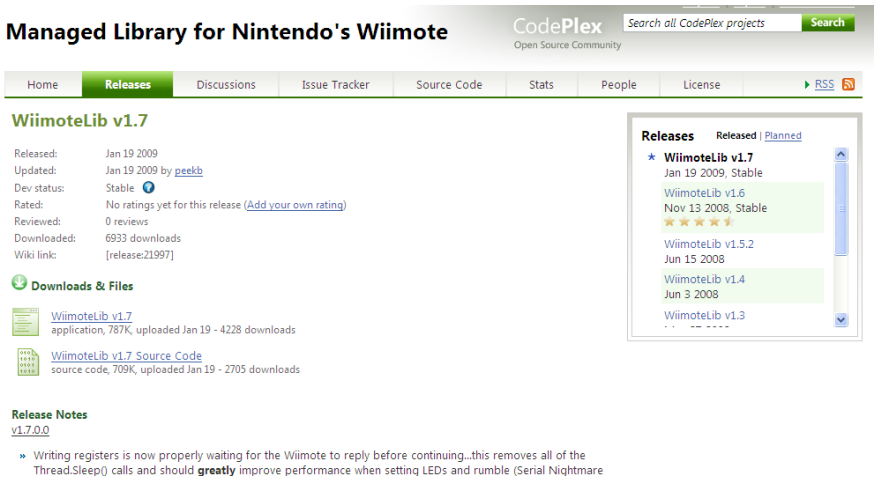


図 7: WiimoteLib のホームページ

ホームページ上部にある「Download」のリンクをクリックすると、ダウンロードページを閲覧できます。配布用のパッケージとソースコードが配布されていますが、ソースコードではないほう「WiimoteLib V1.7 (787K)」をダウンロードしてください。

クリックすると、ライセンス条文が英語で表示されますので (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 ファイルを展開したら、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。

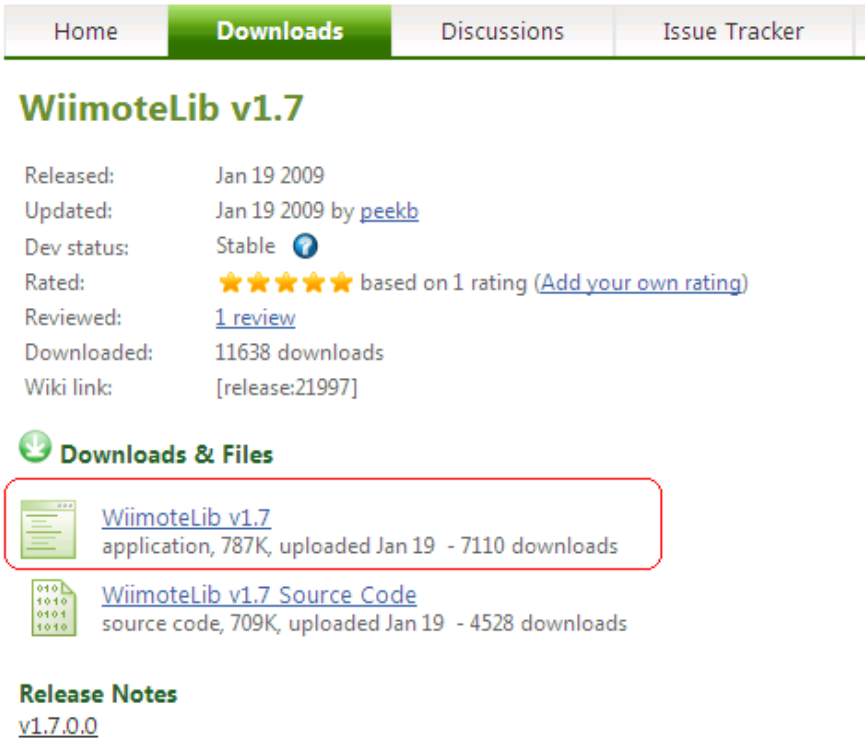


図 8: ダウンロードするファイル

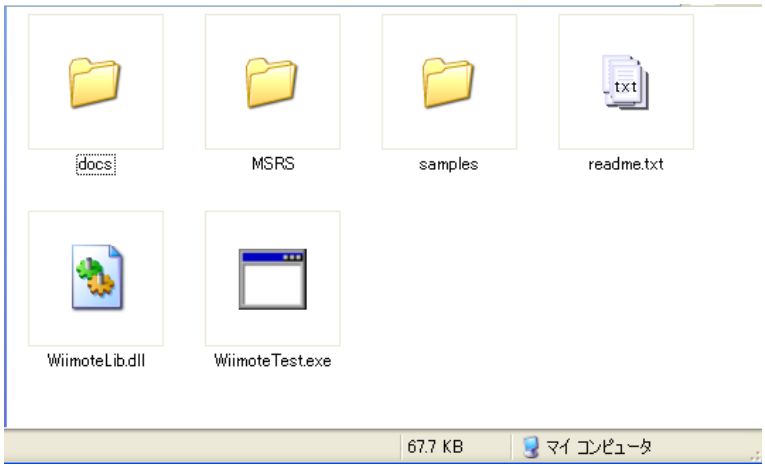


図 9: WiimoteLib1.7 の同梱ファイル

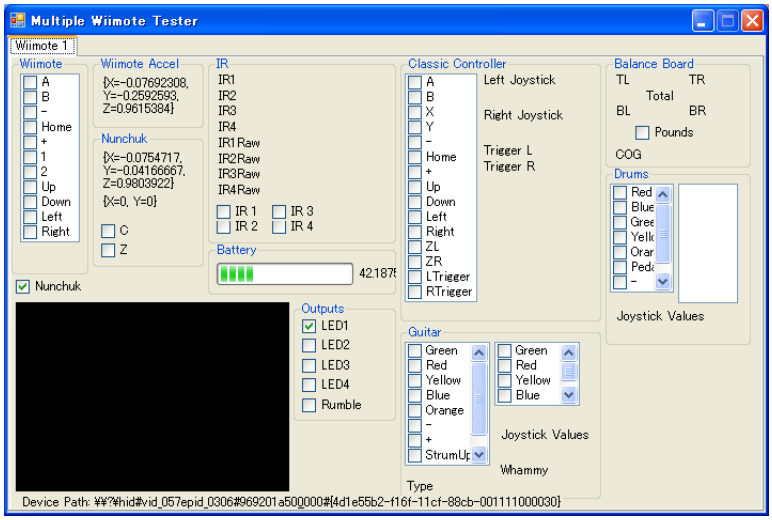


図 10: WiimoteLib のデモプログラム「WiimoteTest.exe」

フォームの右上「×」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、WiiFit バランスボードなどありましたら、ぜひ接続して動作を試してみてください。

これで WiimoteLib を使えることが確認できました。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 11: 「WiimoteTest.exe」の起動に失敗したら接続を確認しよう

さて、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 を開発で利用するにあたり、特にインストーラーなどはありませんので、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 が混在すると後々厄介です）。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!次の節からは実際に、まずは Visual C#を用いて WiiRemote を制御していきます。

0.4.1 WiimoteLib をプロジェクトに組み込む【C#編】

この節では、Visual C# 2008 Express(以後「C#」と標記) を使って、WiimoteLib でのプログラミングを体験していきます。

空のプロジェクトの作成

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

図 12: Visual C#を起動したところ

[ファイル (F)] → [新しいプロジェクト (P)] → [Windows フォームアプリケーション] を選択します。

「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。

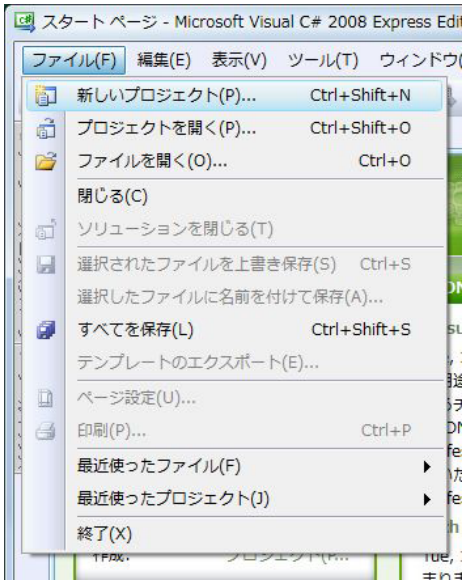


図 13: 新しいプロジェクトを作成

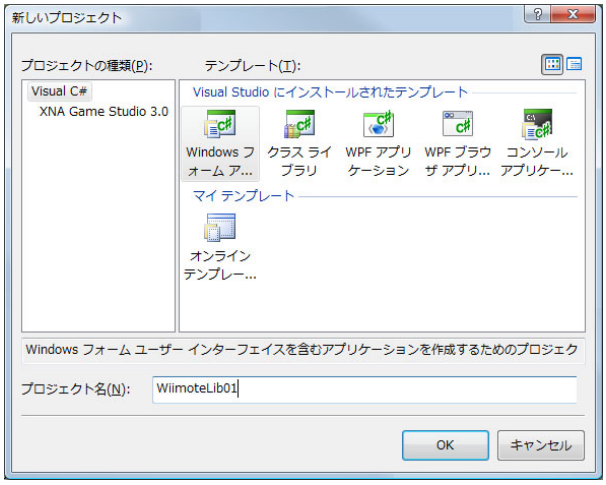


図 14: 「プロジェクト名」に「WiimoteLib01」という名前を付けて [OK]

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、実行してみると良いでしょう。

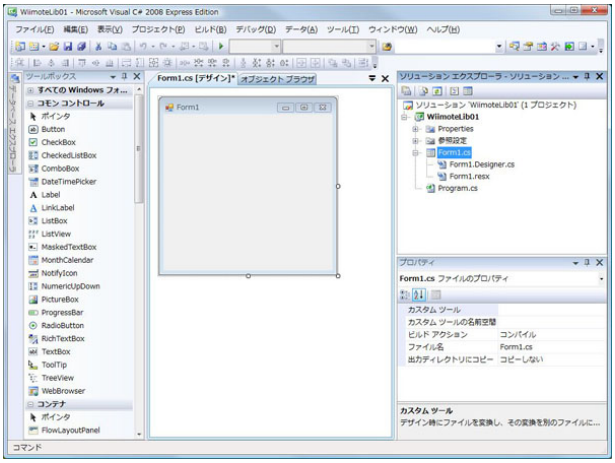


図 15: 新しいプロジェクトが作成されたところ

WiimoteLib の追加 (C#編)

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、[参照の追加 (R)...] を選択します。

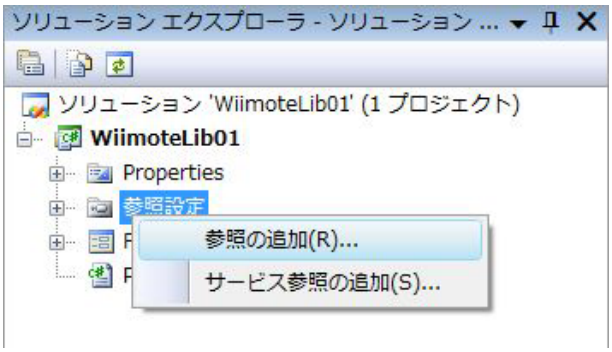


図 16: [参照設定] を右クリック

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

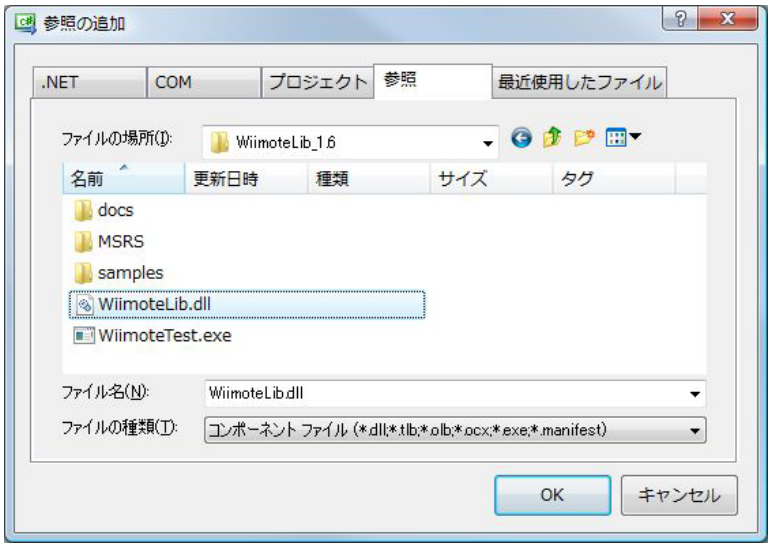


図 17: WiimoteLib.dll のファイルを指定する

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。

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。

```
  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;     //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();
    }

  }
}
```

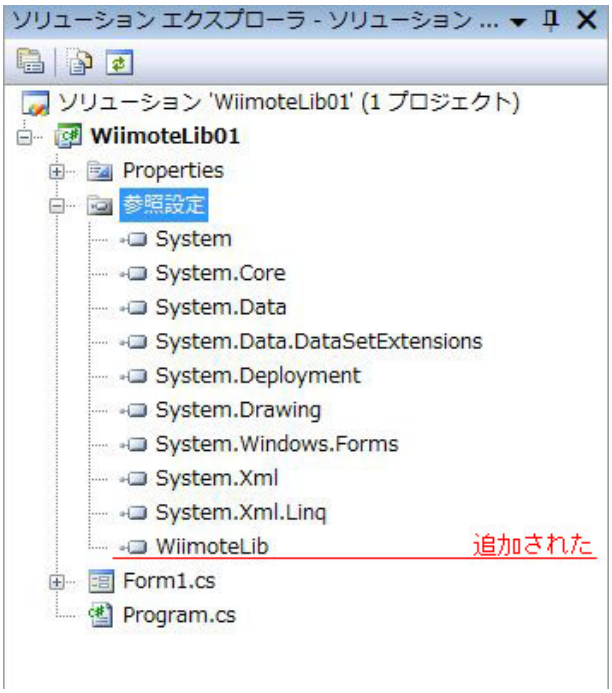


図 18: ソリューションエクスプローラーに現れた WiimoteLib

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、[デバック (D)] → [デバック開始 (F5)] 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下図のように表示されるはずです。

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。

0.4.2 WiimoteLib をプロジェクトに組み込む【C++編】

次は Visual C++ 2008 Express(以後「C++」と標記) を使って、WiimoteLib でのプログラミングを体験していきます。一度流れを覚えたら、以後は共通

図 19: [デバッグ (D)] → [デバッグ開始 (S)]

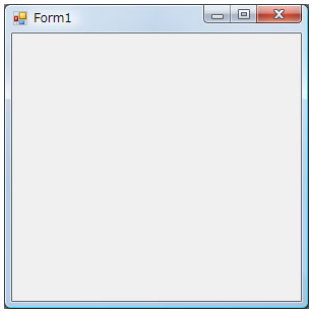


図 20: 何もないフォームが表示された

です。言語も C++か C#、どちらでもかまいません。ご自身が使いやすい言語を選んでみてください。

空のプロジェクトの作成

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

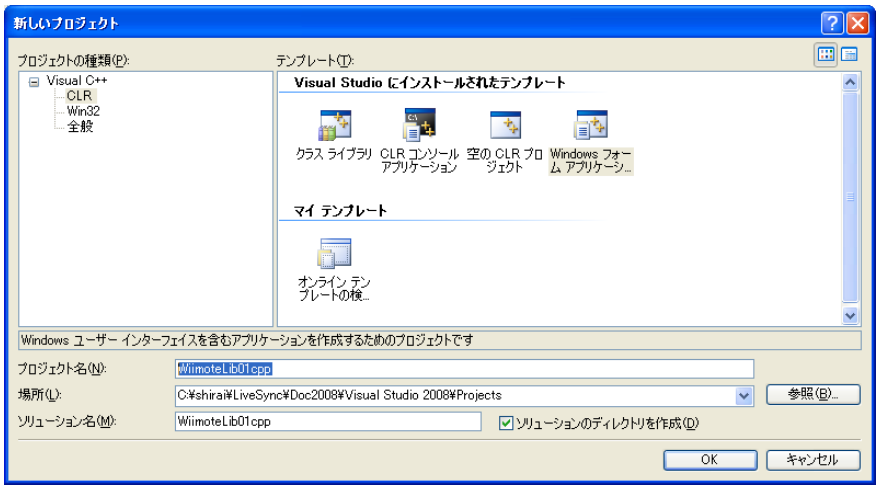


図 21: 新規プロジェクトの作成 [C++]

[ファイル (F)] → [新しいプロジェクト (P)] から、[CLR] の [Windows フォームアプリケーション] を選択します。

「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。

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、実行してみると良いでしょう。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 [C++]

先ほど作成したプロジェクトに WiimoteLib を組み込んでいきましょう。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 (N)...] 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」においた「WiimoteLib_1.7\WiimoteLib.dll」を選択し [OK] ボタンをクリックします。

図 22: プロジェクトを右クリックして [参照] 設定

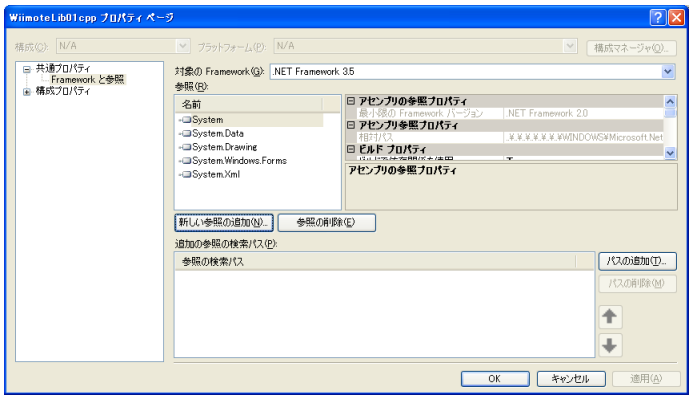


図 23: WiimoteLib を追加

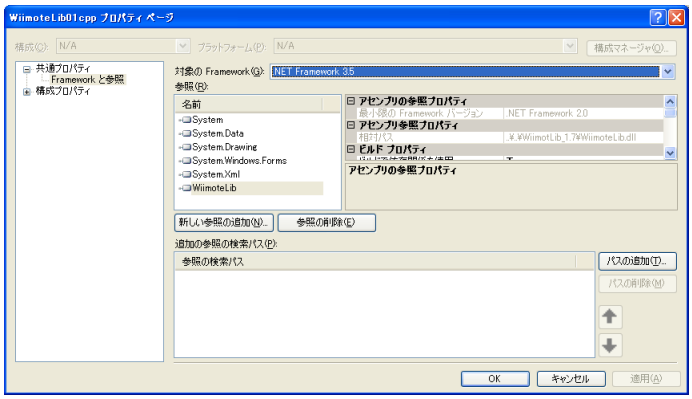


図 24: WiimoteLib が参照設定に追加された

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。

リスト 1: WiimoteLib クラスを宣言 (Form1.h)

```
#pragma once

namespace WiimoteLib01cpp { //←指定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;  //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。いまのところ、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。

0.5　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を行っていきます。解説は C++と C#を並列して解説していきますが、.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開発を進めることができます。

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) の動作をあやつるするプログラムを作ります。

0.5.1　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。

リスト 2: Form1.cs[C#]

```
    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>
```

リスト 3: Form1.h[C++]

```
#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分かりいただけたでしょうか？C++では wm という入れ物を Form1 クラスの Public メンバーとして用意しています。

0.5.2 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」がありますので、フォームの上にドラッグしてください。

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。

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ますので下記のように記述します。

リスト 4: Form1.cs[C#]

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

リスト 5: Form1.h[C++]

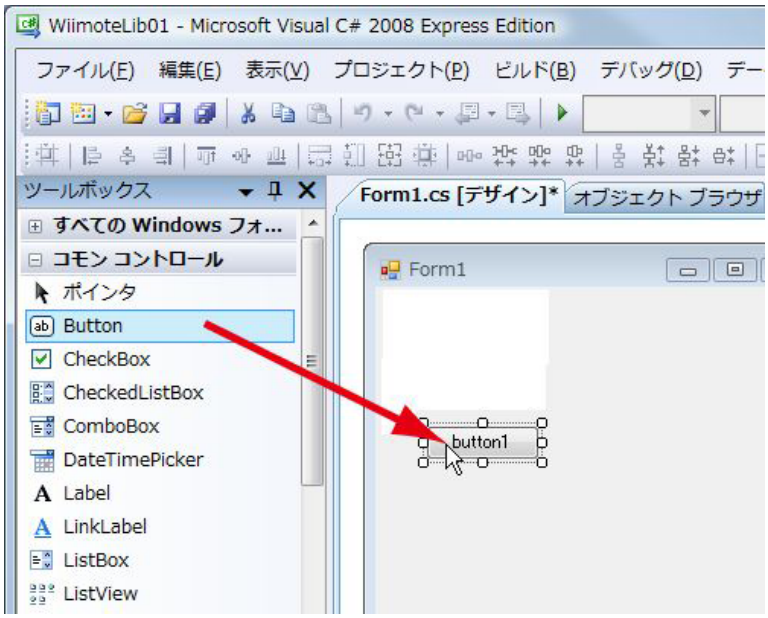


図 25: ツールボックスからボタンをドラッグして配置

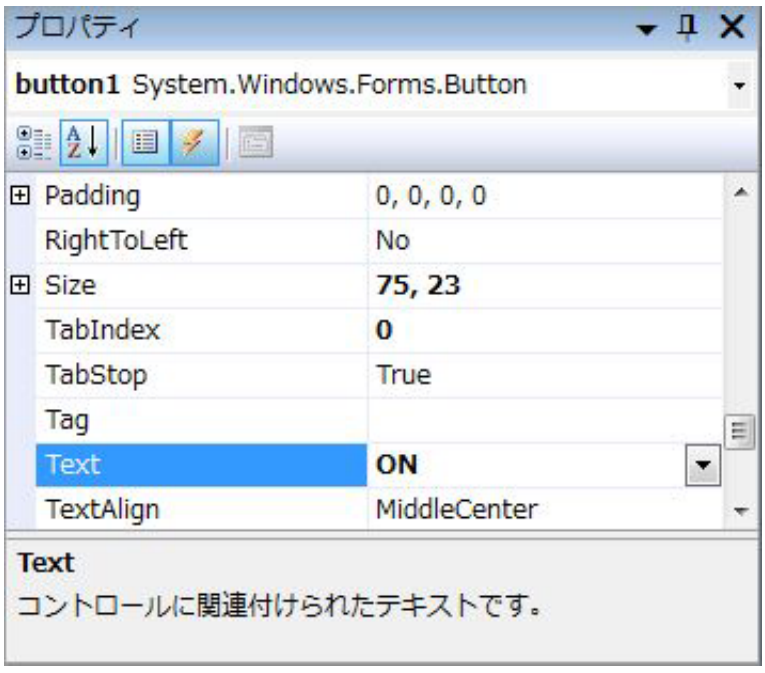


図 26: Text のプロパティを「button1」から「ON」に書き換える

```
#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困った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。

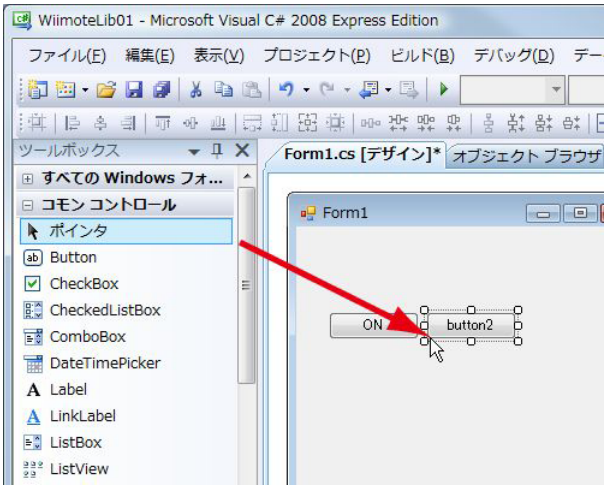


図 27: 「OFF」のためのボタンをドラッグで配置

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。

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

リスト 6: Form1.cs[C#]

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

リスト 7: Form1.cs[C#]

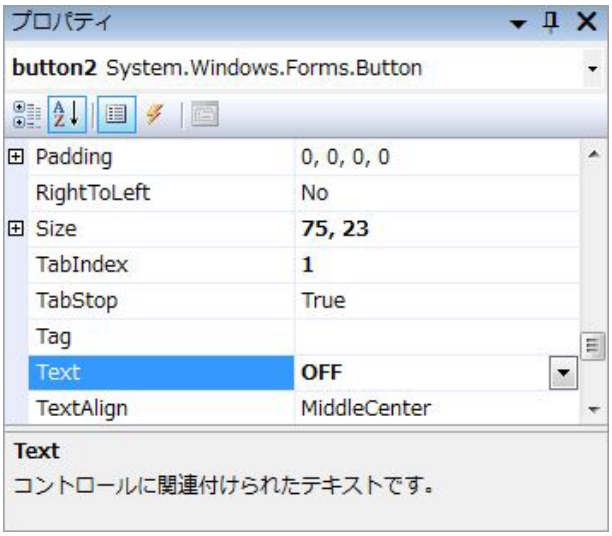


図 28: Text のプロパティを「button2」から「OFF」に書き換える

```
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。

リスト 8: Form1.cs の C#ソース (全体)

```
 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

 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
   }
  }
 }
```

リスト 9: 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
  }
private: System::Void button2_Click(System::Object^ sender, \
    System::EventArgs^ e) {
|     wm->SetRumble(false);  //バイブレーション OFF
  }
 };
}
```

0.5.3 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。

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。エラーがなければ下のようなフォームが起動するはずです。

図 29: Bluetooth 接続 (図は東芝製スタック)

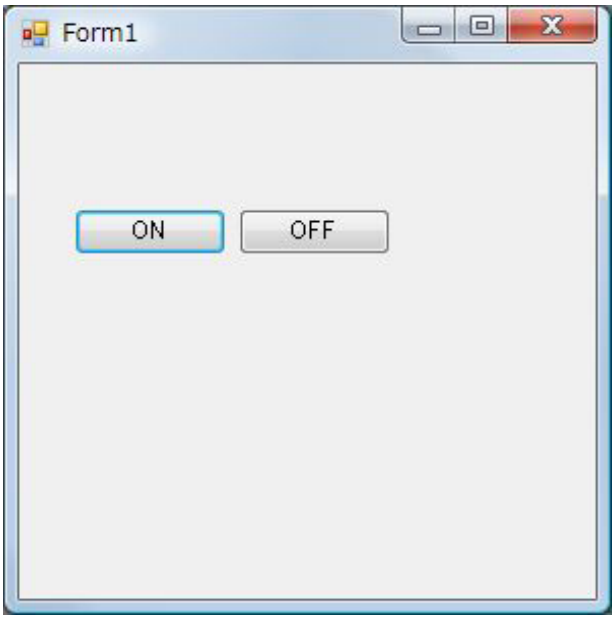


図 30: 「ON」と「OFF」をもつフォームが表示される

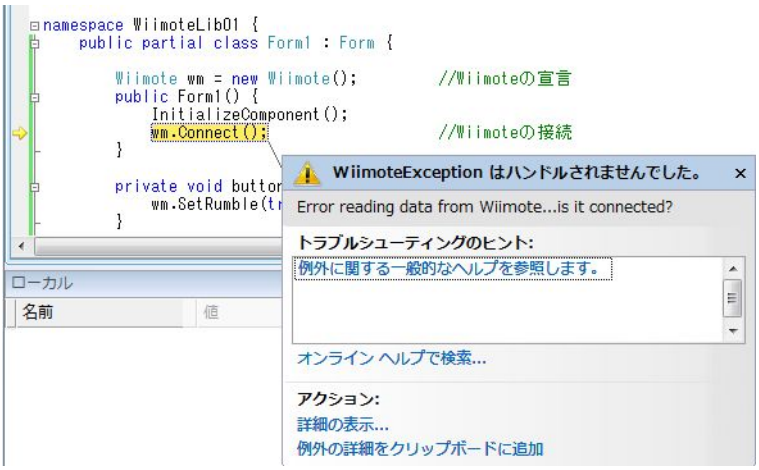


図 31: 接続失敗：Bluetooth 接続を確認しよう

もし下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。

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 の振動をとめましょう。

どうでしょうか？非常に簡単に WiiRemote のバイブレーション制御プログラムが作れました。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していくことができます。

WiimoteLib にはバイブレーターの制御以外にも WiiRemote が制御するための関数がたくさん揃っています。

簡単にプログラムの流れを解説しますと、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。今回は例外処理を行っていませんので、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。() の引数に「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。PWM 制御 (パルス幅変調:Pulse With Modulation) を用いることによってバイブレーションに強弱を付けることができます。簡単に説明すると高速に ON と OFF を繰り返すことによってバイブレーションに強弱をつける制御方法です。ここではあえて説明しませんが、余力のある人はチャレンジしてみてください。

0.5.4 解説：API 関数

以下、このプログラムで使った WiimoteLib の API 関数です。

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++では何ら代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りますので非常に快適にコーディングできます。次のステップでは同じ要領で、LED の点灯を制御していきます。

0.6 LED の点灯と消灯

次に WiiRemote の LED 制御についてプログラミングを行っていきます。ここでは、Form ボタンをクリックする毎に 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ていきます。

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、以下の初期化コードを書き足してください。

0.6.1 WiimoteLib の宣言と接続

リスト 10: Form1.cs に以下の部分を追加 [C#]

```
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
  }
 }
}
```

リスト 11: Form1.h に以下の部分を追加 [C++]

```
#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
| 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 の追加わすれなど、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までの流れ何度も行いますので、カラダで覚えてしまいましょう。

なお 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#と違いが出てきます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）。

0.6.2 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ますので、ボタンは少し大き目、横長にしておくと良いでしょう。

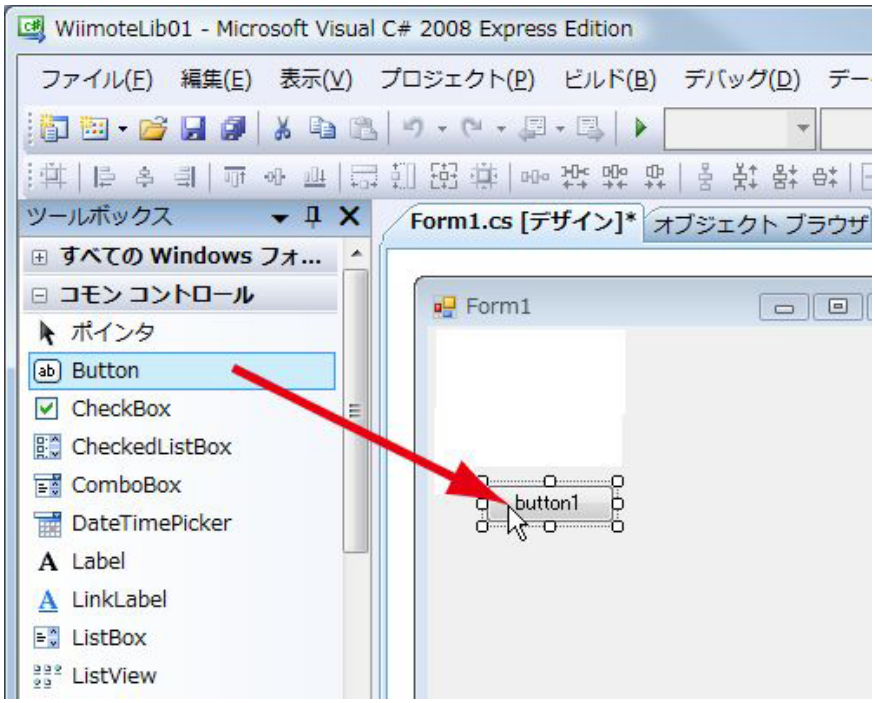


図 32: フォームに大きめなボタンを配置 [C#]

次に貼り付けた「button1」をダブルリックして、以下のコードを追加します。

リスト 12: ボタンクリック時の処理を追加 [C#]

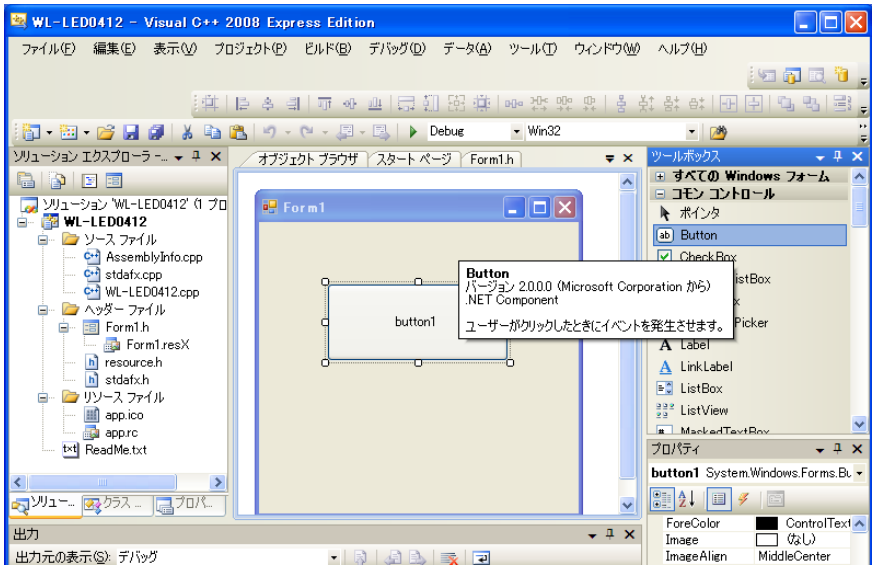


図 33: フォームに大きめなボタンを配置 [C++]

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

リスト 13: ボタンクリック時の処理を追加 [C++]

```
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。

リスト 14: 完成した Form1.cs[C#]

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

リスト 15: 完成した Form1.h[C++]

```
#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()
   {
   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。

0.6.3 実行してみよう

それでは早速実行してみましょう。

次に、Visual C#/C++の [F5] キーを押して実行してください。実行すると下図のようなアプリケーションが起動します。

図 34: 大きなボタンがひとつだけのフォームが表示される

もし下図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。

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。

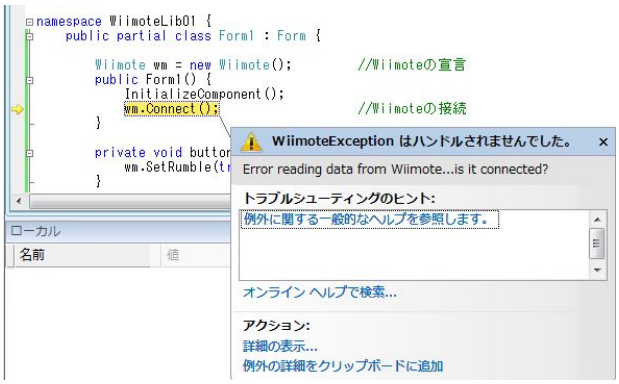


図 35: Bluetooth 接続を忘れるとエラーになる

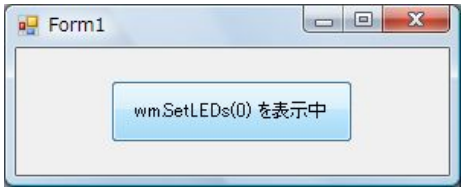


図 36: フォームに文字が表示される

「×」をクリックして終了します。

0.6.4　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 形式の数値ををを入れることで、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。

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（●が消灯、○が点灯です)。

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 は全て消えますが、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りますので、LED はカウントアップしつづけます。

表 1: LED の点灯でまなぶ、2 進数対応表

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

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)

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ますので、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。例えば、筆者は「SetLEDbyValue(int)」という関数を作って、4 段階の値を表示できるようにしました。「レースゲームでのシールド残量」や「受けたダメージの表現」、それから処理の段階を表すプログレスバーにも LED が使えます。

0.7 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。

0.7.1 ラベルの作成

さき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。

この Label の文字は、後ほどプログラム側から書き換えますので設定は不要です。

0.7.2 プログラム

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。

リスト 16: ボタン入力に対応させる (Form1.cs)[C#]

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。例えば、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 Wiimo

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プログラム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。

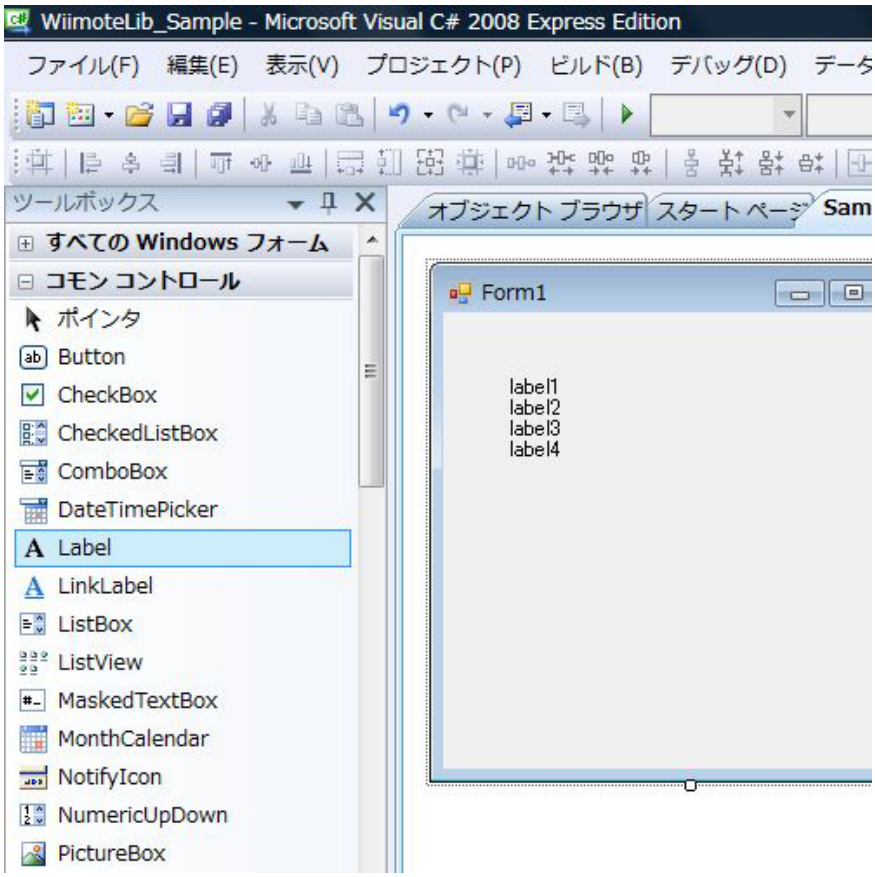


図 37: 新しいプロジェクトにラベルを 4 つ配置

```
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)
|   {
|    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
|   }
  }
}
```

リスト 17: ボタン入力に対応させる (Form1.h)[C++]

```
#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);
|   }
| 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>
```

0.7.3 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下図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、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。

WiiRemote の A ボタンを押し続けます。

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て、すべてのボタンを押さない状態にしたとき、フォーム上のボタンのステータスを表す表示が全て「False」になれば成功です。

図 38: 実行直後、ラベルの表示に注目

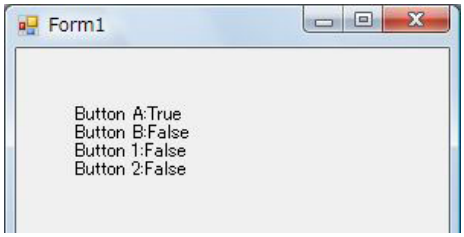


図 39: ラベルの表示が変わる

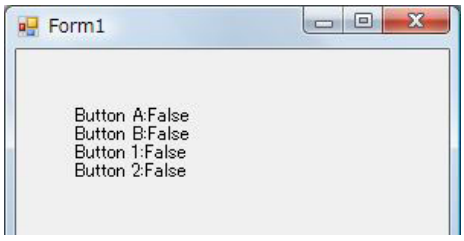


図 40: ボタンから手を離すと、すべて False になる

WiiRemote の A ボタンと B ボタンを同時に押しこんでみます。

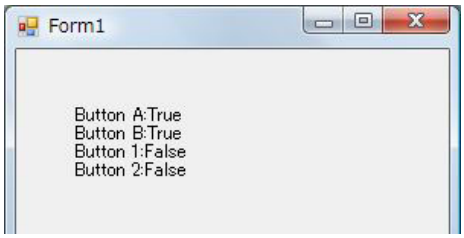


図 41: A,B が True になる

最後に、WiiRemote の 1 ボタンと 2 ボタンを同時に押しこんでみます。

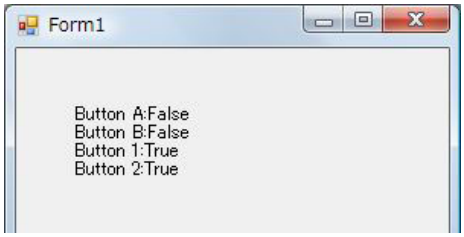


図 42: 1,2 が True になる

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

0.7.4 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。

リスト 18: イベント関数の登録

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

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登録しています。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決められた方

法で指定します。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ますので、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)。

リスト 19: WiimoteState の値を取得

```
[C#]  WiimoteState ws = args.WiimoteState;
[C++] WiimoteState^ ws;
      ws = args->WiimoteState;
```

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”にとりこんでいます。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。以下、使い方を見てみましょう。

リスト 20: ボタンの状態を取得

```
[C#]  this.label1.Text   = "Button A: " + (ws.ButtonState.A);
[C++] 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() する必要はない)。

同様に「ws.ButtonState.b」などとすることで WiiRemote の B ボタン、その他全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、様々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。

リスト 21: おまじない

```
[C#]  Control.CheckForIllegalCrossThreadCalls  = false;
[C++] 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 (スレッド;Thread　処理の流れ) 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。WiimoteLib の公式で紹介されている方法で、

リスト 22: 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 [C#]
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;
```

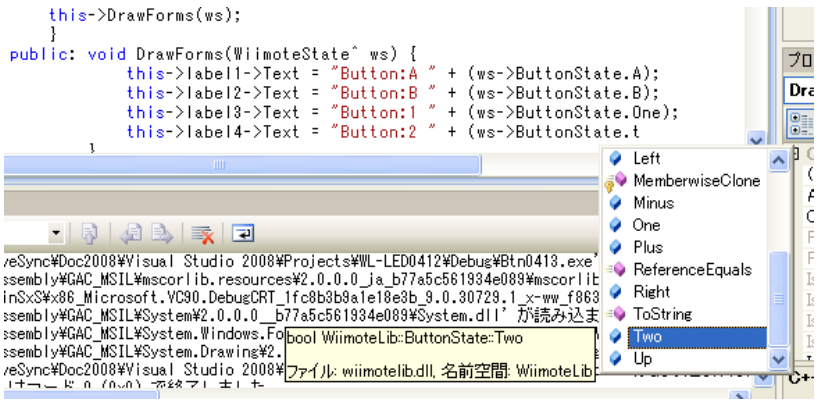
として、不正なスレッド間コールのチェックをしない、と宣言する方法もあるので、本章では「おまじない」として以後この方法を採用することにします。

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。

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 には実に様々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いられませんので、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。全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。



0.7.5　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良いでしょう。

リスト 23: ボタンイベントで外部プログラムを起動する (Form1.cs)[C#]

```
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);
|            }
|        }
    }
}
```

リスト 24: ボタン入力に対応させる (Form1.h)[C++]

```
#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");
|      }
|      //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 ボタンを数回おすと、押した回数だけ、電卓が起動します。

図 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。

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

「System.Diagnostics.Process.Start()」についてはいろいろな応用があります。テキストファイルなどを指定することで関連づけられたプログラムを使って開くことなども可能です。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良いでしょう。

以上は、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。

```
//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。

0.8　加速度センサーを使う

0.8.1　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。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。

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 の値を持ちます。このセクションでは、まず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。

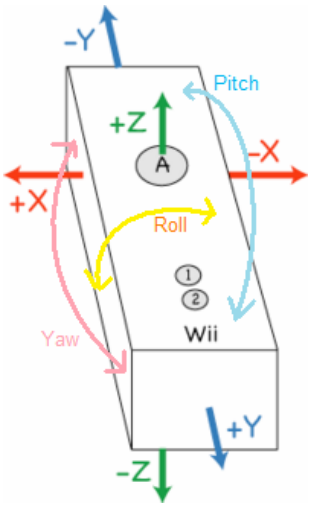


図 44: 加速度センサー (★ http://www.wiili.org より借用)

0.8.2 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 を右クリックして「コードを表示」して、以下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、最小限必要な物のみにしています。

リスト 25: 加速度センサーの値を表示する (Form1.cs)[C#]

```
  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);
|   }
  }
}
```

リスト 26: 加速度センサーの値を表示する (Form1.h)[C++]

```
#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) {
```

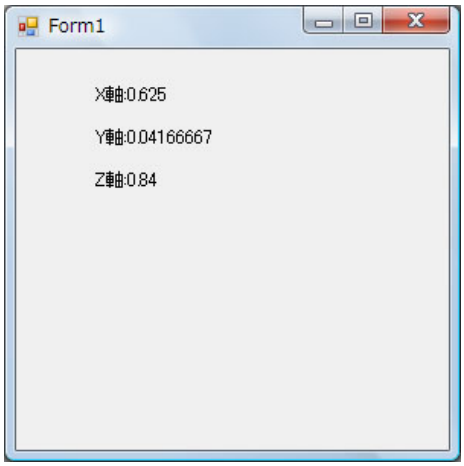
```
|     this->label1->Text = "X 軸:" + (ws->AccelState.Values.X);
|     this->label2->Text = "Y 軸:" + (ws->AccelState.Values.Y);
|     this->label3->Text = "Z 軸:" + (ws->AccelState.Values.Z);
|   }
```
＜以下略＞

0.8.3　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 を振りまわしてみてください。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。



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

【注意】起動時に 3 つの値がゼロのままだった場合、一旦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 45: Dispose でエラーがあるが今は無視

0.8.4 解説：レポートタイプと加速度センサー

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。

リスト 27: レポートタイプの設定

```
[C#]   wm.SetReportType(  InputReport.ButtonsAccel, true);
[C++]  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。

リスト 28: 加速度センサーの生値取得

```
[C#]  ws.AccelState.Values.X
[C++] ws->AccelState.Values.X
```

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

リスト 29: おまじない

```
[C#]  Control.CheckForIllegalCrossThreadCalls = false;
[C++] 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。そして、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 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、もうひとつの軸、たとえば Y 軸にはかならず ±0.9 程度の値が残ります。

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。

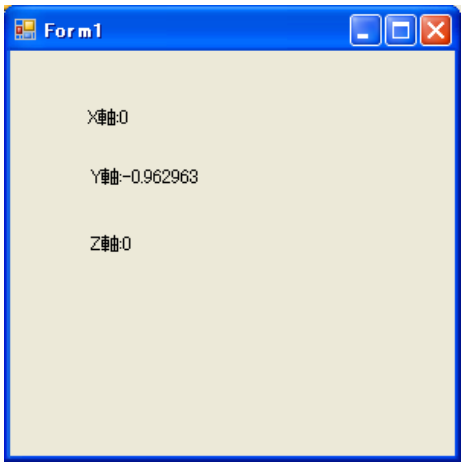


図 46: Y 軸に検出されているのは...「重力加速度」

0.8.5 加速度センサーで作る WiiRemote 太鼓

せっかく WiiRemote の特徴のひとつである加速度センサーの値が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。

応用：振ると WAV ファイルを再生

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良いでしょう。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。

リスト 30: 振ると WAV ファイルを再生 (Form1.cs)[C#]

```
  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();
```

コラム：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他にも、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 (WiimoteLib では 4 点のみサポートしています)。

```
|       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
|         }
|       }
|     }
    }
}
```

リスト 31: 振ると WAV ファイルを再生 (Form1.h)[C++]

```
#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
|         }
|       }
|     }
<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、なんだか音が聴いたことのある音が鳴ります。

解説

WiiRemote の 3 軸の加速度センサの値を取得して、指定した加速度を検出すると指定した WAV ファイルを鳴らします。

リスト 32: 再生する WAV ファイルを指定

```
[C#]   path = @"C:\WINDOWS\Media\achord.wav";
[C++]  path ="C:\\WINDOWS\\Media\\achord.wav";
```

この「@」は「@-quoted string」といって、これが先頭に着いている文字列は「\」を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標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良いでしょう。

リスト 33: 絶対値

```
[c#]  float AX = Math.Abs(ws.AccelState.Values.X); //X 軸の絶対値
[c++] 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。

リスト 34: 判定

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

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、音が再生中に 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)。

0.9 赤外線センサーを使う

さて、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 の視界にはいると、バイブレーターを鳴らすことができます。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、とても面白い情報を伝えるメディアになることを体感していきましょう！

0.9.1 赤外線探知機

早速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編してもよいのですが、新規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なく、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、プロジェクトができあがったら、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」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、一旦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 2 つ張り付けてください。

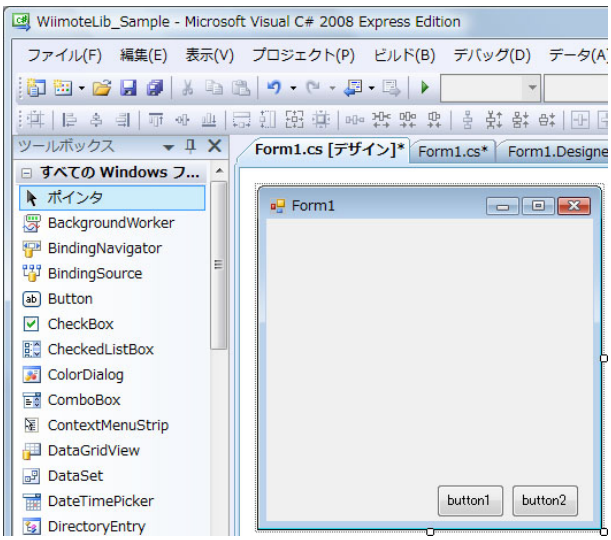


図 47: フォームにボタンを 2 つ配置する

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。

フォーム「Form1」を右クリックして「コードの表示」をして宣言と、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。

リスト 35: 赤外線探知機 (Form1.cs)[C#]

```
//不要な using 宣言は削除してかまいません
  using System;
  using System.Windows.Forms;
| using WiimoteLib;     //WimoteLib の使用を宣言

namespace IR1
{
  public partial class Form1 : Form
```

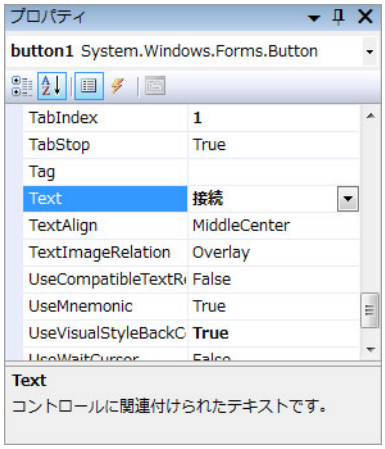


図 48: 「button1」の text プロパティを「接続」に

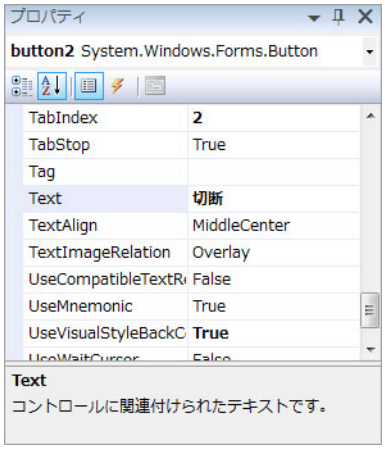


図 49: 「button2」の text プロパティを「切断」に

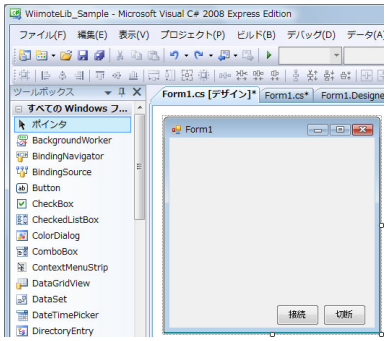


図 50: ボタンの完成

```
  {
|   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
|     }
|   }
  }
}
```

リスト 36: 赤外線探知機 (Form1.h)[C++]

```
#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
  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。上手く検出できると、バイブレーターが鳴ります。バイブレーターが鳴っていない状態で [切断] ボタンを押してから終了しましょう。

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ませんので、何度か連打して確認をおこなうとよいでしょう。

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物に利用されていることに気がつくことでしょう。

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ひとつでも見つかったら、バイブレーターを ON にします。

リスト 38: レポートタイプの設定

```
[C#]  wm.SetReportType ( InputReport.IRExtensionAccel, true);
[C++] 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。このコールはかならず 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。

リスト 39: 赤外線の検出

```
[C#]    ws.IRState.IRSensors[0].Found
[C++]  ws->IRState.IRSensors[0].Found
```

このプロパティは True/False を返しますので、if 文を使って赤外線を検出することができます。また WiimoteLib は、同時に 4 点まで赤外線光源を検出することができます。個々の光源を特定することはできませんが「IRSensors[3].Found」これが True なら 4 つの赤外線光源が見えている、ということです。

ボタンを押しても何も起きない時は？

上記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、上記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」を覗いてみましょう。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。

リスト 37: Form1.Designer.cs で自動生成されているコード [C#]

```
#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。大事なのは、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。

0.9.2　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(プレイヤーインジゲーター) をつかって、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。

リスト 40: 赤外線探知 LED 表示 (Form1.cs, 抜粋)[C#]

```
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);
}
```

リスト 41: 赤外線探知 LED 表示 (Form1.h, 抜粋)[C++]

```
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
  }
| //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「SetLED s ()」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共に LED インジゲーターを使って赤外線検出個数を表示します。

0.9.3　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。

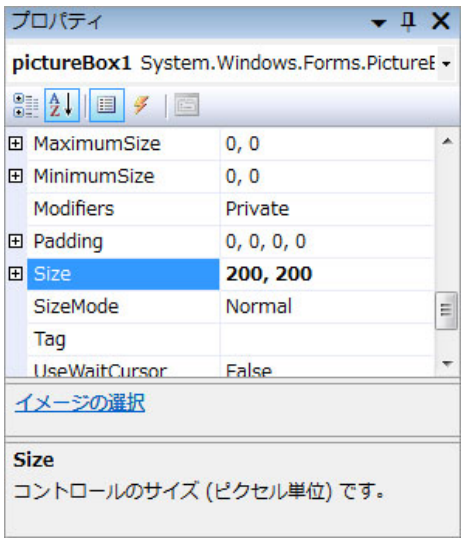


図 51: PictureBox の Size プロパティ

まず、フォームのデザインを変更します。はじ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加筆します。

リスト 42: 赤外線ポインタ描画 (Form1.cs)[C#]

```
  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) {
```

図 52: Form1 に配置した PictureBox とボタン

```
     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,
|       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
| }
 }
}
```

リスト 43: 赤外線ポインタ描画 (Form1.h)[C++]

```
 #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:
    /// <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 );
|       }
|       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。

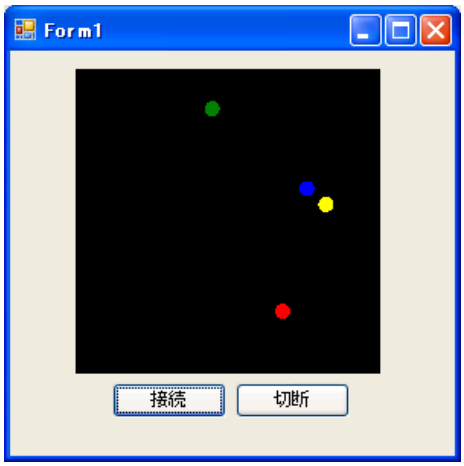


図 53: 赤外線に向けると 4 色のマーカーが動く

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。

赤外線が検出されると図のようにマーカ点が表示さます。マーカ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 2m〜3m まで離してください。距離が長いほど安定した動きを行うことができます。

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 4 点検出できるという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ひとつのグループとして誤認識されノイズの原因になりますので、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。上の図では、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

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。

リスト 44: 赤外線座標の取得

```
[C#]   ws.IRState.IRSensors[0].Position.X
[C++] 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 200x200 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等が 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です、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 のひとつで、非常に安定して動作します。.NET という環境から「C#専用？」と考えられていますが、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、具体的な開発例は 8 章でも扱っていきます。