PowerTool.Automation

QuickStart Guide

Last Updated for 5.0.0.0

Contents

About this Guide

The PowerTool software has a built-in (OLE) Automation interface that allows the programmer or engineer to programmatically control its operation.

What follows is a brief description of the PowerTool.Automation interface, what it is, how to install it, and how to use it.

It is intended to give the reader an acquaintance in using the interface, and guide him or her in their "first steps". It is not intended as a comprehensive reference.

Introduction

One can start or stop multiple PowerTool applications, connect to various Power Monitor devices, set trigger codes, start and stop sample runs, or save and load files via programmatic control, much like an interactive user can do sitting at his or her console.

This document assumes that the reader has at least some familiarity with programming concepts in general, and COM or the .NET Framework in particular.

Installation

First, install the PowerTool application via our standard installer. Typically, you will have already done that if you have gotten to this point. The Automation sample implementations can be found in the 'DeveloperExamples' subdirectory where you installed the Monsoon Power Tool applications. Executables and source code are both distributed for these examples. A Visual Studio solution file is present, as are individual project files for each example. Be sure to read our usage notes before building the samples. Your next steps will vary depending on whether you are accessing the library via .NET aware languages like C# or via COM aware languages like C++.

Note, if you are building the solution as a whole, you will need to follow the steps for COM and .NET or the C++ examples will not build. If you are only building specific C# sample projects, then you can skip the COM specific steps.

Interface Definition

Subject to change. Current as of 4.0.5.0. Note, the interface changed in 4.0.5.0!

```
[
    Guid("7DC013FC-D46C-4AD2-B01C-383176CBDDB8"),
    InterfaceType(ComInterfaceType.InterfaceIsDual),
    ComVisible(true),
]
public interface IAutomation
{
    #region IAutomation Properties

    // Whether the PowerTool application is open
    bool ApplicationIsOpen { get; }

    // Whether a Power Monitor device is connected
    bool DeviceIsConnected { get; }

    // Whether a sample is currently running
    bool SampleIsRunning { get; }

    // Serial number of the connected device
    ushort DeviceSerialNumber { get; }

    // Whether the application has any data captured or loaded from file
    bool HasData { get; }
```

```
        // Returns the summary statistics structure
        SelectionData SelectionData { get; }

        // Calibration status during sample run
        // Values: OK, Warning, Failed
        CalibrationStatus CalibrationStatus { get; }

        // Program exit code
        ExitCode ExitCode { get; }

        // Data file name
        string FileName { get; }

        // Application status
        PowerToolStatus PowerToolStatus { get; }

        // Enable/disable throwing exceptions on errors
        // (mostly during processing of properties)
        bool ExceptionsAreEnabled { get; set; }

        // Visibility state of the Power Tool application window
        bool Visible { get; set; }

        // Window state (min, max, normal)
        WindowState WindowState { get; set; }

        // Power-up time, in milliseconds.
        // Allowable range: 0-255
        byte PowerUpTime { get; set; }

        // Current limits, in amps.
        // Allowable range: 0-15.625
        float PowerUpCurrentLimit { get; set; }
        float RunTimeCurrentLimit { get; set; }

        // USB passthrough mode.
        // Allowable values: Auto, On, Off, Trigger, Sync
        UsbPassthroughMode UsbPassthroughMode { get; set; }

        // Directory for temporary files.
        // The directory must exist
        string TempDirectory { get; set; }

        // Main and USB resistor offsets, in ohms.
        // Depricated functions.  Use SCALE instead.
        float MainFineResistorOffset { get; set; }
        float MainCoarseResistorOffset { get; set; }
        float UsbFineResistorOffset { get; set; }
        float UsbCoarseResistorOffset { get; set; }

        // Aux resistor offsets, in ohms
        // Depricated functions, use SCALE instead.
        float AuxFineResistorOffset { get; set; }
        float AuxCoarseResistorOffset { get; set; }

//32-bit scale values.
```

```
//Current reported scales linearly with these values.
 UInt32 MainFineScale { get; set; }
 UInt32 MainCoarseScale { get; set; }
 UInt32 UsbFineScale { get; set; }
 UInt32 UsbCoarseScale { get; set; }
 UInt32 AuxFineScale { get; set; }
 UInt32 AuxCoarseScale { get; set; }

 //Manual offset.  Adds or subtracts a static amount from the ADC count.
 //Does not scale linearly.
 short MainFineZeroOffset { get; set; }
 short MainCoarseZeroOffset { get; set; }
 short UsbFineZeroOffset { get; set; }
 short UsbCoarseZeroOffset { get; set; }

 // Trigger setting code.
 // Same code as used on the PowerTool command line
 string TriggerSetting { get; set; }

 // Real-time voltage channel selection
 // Allowable values: Main, Aux. (USB not valid)
 Channel VoltageChannel { get; set; }

 // Enable/disable capture of currents
 bool CaptureMainCurrent { get; set; }
 bool CaptureUsbCurrent { get; set; }
 bool CaptureAuxCurrent { get; set; }

 // Set Main output voltage, in volts.
 // Allowable range: 2.0f < setting <= 4.55f
 // HVPM hardware allows a range of 0.8-156
 float MainOutputVoltageSetting { get; set; }

 // Whether Main output voltage is enabled
 bool EnableMainOutputVoltage { get; set; }

 // Battery size, in mAh
 // Allowable range: 1-9000
 uint BatterySize { get; set; }

 // Number of devices available
 uint DeviceCount { get; }

 // Incremental wait time, in milliseconds
 // Default = 1000
 uint WaitInterval { get; set; }

 // Version info
 HardwareRevision HardwareRevision { get; }
 byte FirmwareVersion { get; }
 byte ProtocolVersion { get; }
 SoftwareVersion SoftwareVersion { get; }

 // Number of samples captured
 ulong TotalSampleCount { get; }
 ulong MissingSampleCount { get; }
```

```
// Capture date of sample
DateTime CaptureDate { get; }

// Sample rate (always 5000, but provided for completeness)
uint SampleRate { get; }

// Form position and size, in pixels
int Height { get; set; }
int Width { get; set; }
int Left { get; set; }
int Top { get; set; }

// Log file name.  Empty or null string means no logging
string LogFileName { get; set; }

#endregion IAutomation Properties
#region IAutomation Methods

// Enumerate the Power Monitors which are available for connection.
// Outputs array of device serial numbers found, or null if none.
// Returns the number of devices found (zero if none).
uint EnumerateDevices(out ushort[] serialNumbers);

// Gets the serial number associated with a particular
// device instance. (See DeviceCount property).
// Argument must be in the range 0..(DeviceCount-1)
// Returns 0 if invalid.
ushort GetSerialNumber(uint deviceNumber);

// Open/close the Power Tool application window, with optional wait.
// Wait times are specified in multiplea of the WaitInterval
// property, listed above
bool OpenApplication(bool readIniFile, bool waitFlag);
bool OpenApplication(bool readIniFile, uint waitLimit);
bool CloseApplication(bool writeIniFile, bool waitFlag);
bool CloseApplication(bool writeIniFile, uint waitLimit);

// Connect/disconnect a Power Monitor device
bool ConnectDevice(ushort serialNumber);
bool DisconnectDevice();

// Refresh the application display
bool RefreshDisplay();

// Start/stop sampling
bool StartSampling(bool waitFlag;
bool StartSampling(uint waitLimit); // in multiples of WaitInterval
bool StopSampling(bool waitFlag);
bool StopSampling(uint waitLimit);  // see above

// Load a previously-captured file
bool LoadFile(string fileName);

// Save captured data to a file
bool SaveFile(  string fileName,        // File name
                bool overwriteFile,     // Overwrite file?
                bool createDirectory);  // Create directory?
```

```
        bool ExportCSV( ulong lowIndex,          // 0..(TotalSampleCount-1)
                        ulong highIndex,         // 0..(TotalSampleCount-1)
                        string fileName,         // File name
                        uint granularity,        // 1,10,100,1000,10000 the
                                                 //modulus used to determine
                                                 //which samples to export
                        bool overwriteFile,      // Overwrite file?
                        bool createDirectory);   // Create directory?

        // Sample retrieval
        bool GetSample( ulong sampleIndex,       // 0..(TotalSampleCount-1)
                        out Sample sample);      // A Sample structure

        bool GetSamples(ulong startIndex,        // 0..(TotalSampleCount-1)
                        uint sampleCount,        // Number of samples desired
                        out Sample[] samples);   // Array of Sample structures


       //Resets the connection to the currently connected power monitor
        //device and disables VOut.
        bool ResetPowerMonitor();

        #endregion IAutomation Methods
    }
```

Usage

Approach 1: .NET

All projects built for version 5.0.0.0 require 32-bit compatibility.

.Net 4.5 is specifically required for version 4.0.5.0. and above

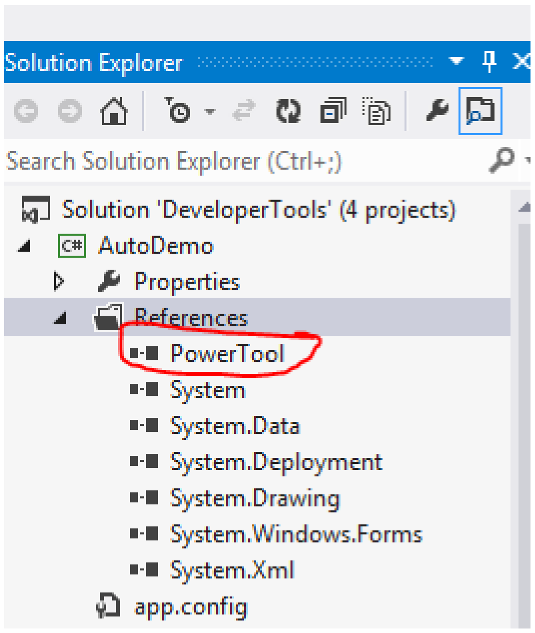
As of 4.0.5.0, all included projects are built under Visual Studio 2015. If you wish to build under Visual Studio 2012 and are only using C#, disable the CPPClient project. Otherwise, you must set your PlatformToolset to VisualStudio 2012 for the CPPClient project or you will have build errors when you build from the Solution as a whole.

.Net 4.0 is specifically required for version 4.0.4.11.

.NET 3.5 is specifically required for versions 4.0.4.0-4.0.4.10.

Assuming that you are writing your code in a .NET-compatible language, such as C#, you simply need to fix the reference to PowerTool.exe in your project to point to wherever it was installed on your computer.

The simplest way to do this is remove PowerTool .exe from your project references, and then re-add it.



No additional assembly registration or steps are necessary. If you did register an earlier assembly in the GAC, .NET will pull that assembly first, and you must unregister it to compile against the new assembly.

Approach 2: COM

Alternatively, if you want to do it the old-fashioned way (COM-style) with a language such as C++, follow the following steps. You will need to redo these steps anytime there is a new version of the PowerTool executable or an interface change.

Note, COM support requires version 4.0.4.9 or later of the PowerTool application. Also note that the COM API is not as full featured as the .NET API and does not expose all of the same functionality.

As of 4.0.5.0, all included projects are built under Visual Studio 2013. If you wish to build under Visual Studio 2012, you must set your PlatformToolset to VisualStudio 2012 for the CPPClient project or you will have build errors.

1. Register the assembly and generate a TLB file.

This step requires that you have installed the free .NET 3.5 (or higher) SDK from Microsoft.

Open a command prompt as an administrator and change to the .NET directory where regasm is installed (for example: c:\Windows\Microsoft.NET\Framework\v4.0.30319). Run the following command:

```
RegAsm.exe [path to powertool.exe] /tlb:PowerTool.tlb
```

For example, with the default PowerTool install location, you would run

```
RegAsm.exe “C:\Program Files (x86)\Monsoon Solutions Inc\Power Monitor \PowerTool.exe”
/tlb:PowerTool.tlb
```

The above command, once completed successfully, registers the PowerTool.exe application as the server for the PowerTool.Automation object and generates the tlb file you will need to import in your code.

You should see an output along the lines of:

```
Types registered successfully

Assembly exported to 'c:\Program Files (x86)\Monsoon Solutions Inc\Power Monitor\PowerTool.tlb', and the type
library was registered successfully
```

2. Register the assembly with the GAC.

This step requires that you have installed either Visual Studio(2010 or later) or the free Windows SDK(version 6 or later) from Microsoft.

Open a Visual Studio Command Prompt as administrator or a Windows SDK Command Prompt as administrator. If you have previously registered a PowerTool assembly be sure to unregister first and then register the new assembly.

If you need to uninstall the assembly from the GAC, you can issue the command:

```
gacutil /u PowerTool
```

Register your new assembly:

```
gacutil /i "C:\Program Files (x86)\Monsoon Solutions Inc\Power Monitor \PowerTool.exe"
```

You should see an output along the lines of:

```
Assembly successfully added to the cache.
```

You can query the GAC to see the PowerTool assembly is installed with the following command. Make sure the PowerTool assembly version shown is the one you expect.

```
gacutil /l PowerTool
```

3. Import the TLB file and initialize the COM objects

The following sample calls are written in C++. To see a reference sample implementation, look at our C++ samples.

Create a C++ project type of your choice, for example, a command line project.

Specify your includes and imports, including the tlb file

-This example uses the ATL COM libraries

```
#include <atlbase.h>
#include <atlsafe.h>
```

-IMPORTANT! Fix this path to match your current location!

```
#import "C:\Program Files (x86)\Monsoon Solutions Inc\Power Monitor\PowerTool.tlb"
```

Specify the namespace

```
using namespace PowerTool;
```

Initialize the COM library

```
::CoInitialize(NULL);
```

Get the powertool instance

```
IAutomation *pAutomation = NULL;
CComPtr<IAutomation> autoPtr;
HRESULT hr = autoPtr.CoCreateInstance(_T("PowerTool.Automation"));
```

Do whatever you want to do here with the interface.

Cleanup

```
if (pAutomation)
        pAutomation->Release();
CoUninitialize();
```

Overview

The following sample calls are written in C#. To see a complete implementation, look at our samples.

Start out with:

```
PowerTool.Automation myPowerTool = new PowerTool.Automation();
```

If you get a non-null result, you are now in programmatic control of the PowerTool. You can open more than one PowerTool if you wish. Each is represented in your space by the object you instantiated, as above.

So, now that you have a PowerTool instance, what can you do with it? You might start out by running it:

```
bool powerToolRunning = myPowerTool.OpenApplication(false, 30);
```

The "30" specifies the number of wait intervals we're willing to wait for the app to start up. The default interval is 1000 milliseconds, so in the above call, we said we're willing to wait 30 intervals, or 30 seconds. The "false" tells it not to read its INI file, we'll take the defaults.

Once the PowerTool app is running, it might be convenient to know what Power Monitor devices are attached to the host computer:

```
ushort[] serialNumbers = new ushort[1];           // just temporary… gets replaced shortly
uint deviceCount  = myPowerTool.EnumerateDevices(out serialNumbers);
```

If there are no Power Monitors, the EnumerateDevices call returns zero, and nulls out the output array. Otherwise, it gives us a new array of serial numbers, and returns the count of devices.

If we see a device we like, we can connect to it by specifying its serial number. So, say EnumerateDevices() reported back at least one device. We can then issue the following call to connect to the first device:

```
bool deviceConnected = myPowerTool.ConnectDevice(serialNumbers[0]);
```

Now that we're connected, the device is ours and will stay that way until we disconnect, or someone pulls the plug.

How about we take some samples?

```
myPowerTool.Visible = true;                         // Let's watch it run
myPowerTool.MainOutputVoltageSetting = 3.7;         // Our device likes 3.7 volts
myPowerTool.EnableMainOutputVoltage = true;         // Give it the juice
bool started = myPowerTool.StartSampling(10);       // Same timing rules as above
// …
// … We do other stuff while sampling occurs on a background thread, and then…
// …
bool stopped = myPowerTool.StopSampling(10);        // Stop sampling… same timing rules
myPowerTool.EnableMainOutputVoltage = false;        // Leave the device powered off
```

Okay, now we have sample data sitting in PowerTool's working memory. Exactly 5,000 data points for each second that the sample was running. What do we do with this data? Well, we could save it in a file…

```
bool dataSaved = myPowerTool.SaveFile ("C:\\MyFolder\\MyFile.pt5", true, true);
```

The "true" values merely tell it to overwrite any like-named file it may find in that folder, and to create the folder if it doesn't already exist.

Now that our data is saved, let's clean up…

```
myPowerTool.DisconnectDevice();                     // Disconnect
myPowerTool.CloseApplication(false, true);          // Don't write INI file, wait for closure
myPowerTool = null;                                 // Dispose of the interface
```

The above is intended as a simple reference and is not a complete guide. See the sample implementations that are included for additional examples.

Sample Implementations

A Visual Studio Solution file is provided for your convenience. "DeveloperTools.sln" contains the projects referenced below.

C# Samples

IMPORTANT NOTE: To build any of these projects, change the project reference to PowerTool.exe to reflect its installed location on your system.

Console Examples

Several simple examples are provided to demonstrate different capabilities.

"SimpleSamplingExample" shows how to connect to a device, sample, access realtime sampling data, and export data to CSV and PT5 files.

"IterativeCallingExample" takes the functionality of SimpleSamplingExample, but runs it iteratively, showing how to properly shutdown and startup your connections each time.

"ResetExample" shows how to use the new to 4.0.5.0 ResetPowerTool method to cycle the power to the connected Power Monitor device.

"IniTriggerExample" shows how to use the Automation interface to work with trigger codes in ini files.

"PT4Reader" shows how to read in a pt4 file to extract the stored data.

"PT5Reader" shows how to read in a pt5 file to extract the stored data.

"RecalibrateExample" shows how to programmatically recalibrate the Power Monitor.

Windows Application Example

A sample application "AutoDemo" is included, both as an executable and with full source code. It contains a full-featured Windows GUI app that demonstrates a more involved implementation of the PowerTool .Automation interface.

When you run the executable, click OPEN to connect to an available PowerTool device. This will refresh the list of devices available to connect via the AutoDemo application. Make sure your desired device is selected, and then click CONNECT to activate control of the PowerTool instance via the AutoDemo application.

C++ Samples

IMPORTANT NOTE: Be sure you have fully followed the directions listed for using COM, before using these samples. Also, be sure to edit the path to your TLB file in your code.

Console Example

A simple application “CPPClient” is included, both as an executable and with full source code. It contains a console application that uses the COM interface. It demonstrates connecting to the PowerTool application and to a specific PowerMonitor device, taking a short sample, and writing the output to a pt5 file.