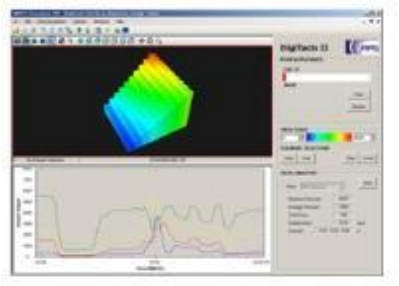
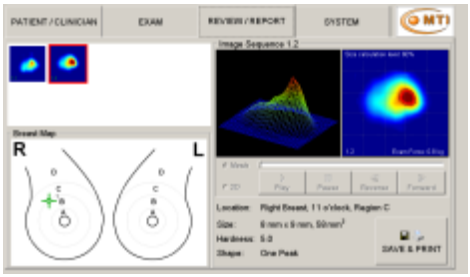
PPS Data Acquisition API Documentation





Document Number: 3812

Effective Date: 6/13/2013

Pressure Profile Systems, Inc.
5757 Century Blvd. Suite 600 ● Los Angeles, CA 90045
www.pressureprofile.com ● +1.310.641.8100

Table of Contents

Revision History

Revision	Date	By	Description
01	3/7/2013	DCA	Initial external release
02	6/13/2013	DCA	Updated for Unicode and VS2010 support

1 Introduction

The PPS Data Acquisition API provides access to sensor output data from any PPS system. It is built as a C-style Windows DLL which can be accessed using many different programming languages, including C, C++, C#, VisualBasic, LabVIEW, and MATLAB. Binary files are provided for creating native 32-bit and 64-bit applications.

1.1 API Contents

The root API folder contains the following files and directories. You may note that the setup\ folder and its contents are duplicated in multiple locations to accommodate default directory paths in Microsoft Visual Studio. These directories are automatically copied from the api\setup directory as post-build events in Visual Studio.

General Files for All Projects

- `PPS Data Acquisition API.pdf` ..... This file
- `bin\` ..... Binary files needed for projects
  - `x64\` ..... 64-bit Binary Files
    - `HwPlugins\` ..... 64-bit Hardware-access Libraries
    - `ProcPlugins\` ..... 64-bit Processing Libraries
    - `PPSDaqApi.dll` ..... 64-bit Main API Library
    - `setup\` ..... Setup folder with sample config file
      - `16x16_emulator.cfg` ..... Sample config file for emulated data
  - `x86\` ..... 32-bit Binary Files
    - `HwPlugins\` ..... 32-bit Hardware-access Libraries
    - `ProcPlugins\` ..... 32-bit Processing Libraries
    - `PPSDaqApi.dll` ..... 32-bit Main API Library
    - `setup\` ..... Setup folder with sample config file
      - `16x16_emulator.cfg` ..... Sample config file for emulated data
- `inc\` ..... Include files for C/C++ projects
  - `PPSDaq.h` ..... Universal C/C++ include file
- `lib\` ..... Import library files
  - `x64\` ..... 64-bit Import Library Files
    - `PPSDaqApi.lib` ..... 64-bit Import API Library
  - `x86\` ..... 32-bit Import Library Files
    - `PPSDaqApi.lib` ..... 32-bit Import API Library
- `setup\` ..... Setup folder (auto-copied to output directories)
  - `16x16_emulator.cfg` ..... Sample config file for emulated data

Sample Application files

- `PpsApiExamples.sln` ..... MSVS 2010 Solution for all Projects
- `C++Samples\` ..... Directory for C++ Sample Projects
  - `setup\` ..... Setup folder with sample config file
    - `16x16_emulator.cfg` ..... Sample config file for emulated data
  - `PPSDaqDyn.cpp` ..... Source file for dynamic linking
  - `PPSSampleApp.cpp` ..... Source file for sample C++ application
  - `PpsDynamicCpp.vcxproj` ..... MSVS 2010 Dynamic-link Project File
  - `PpsStaticCpp.vcxproj` ..... MSVS 2010 Static-link Project File
- `CSharpSample\` ..... Directory for C# Sample Project
  - `PPSDaq.cs` ..... Source file for API access
  - `Program.cs` ..... Source file for sample C# application
  - `CSharpSample.csproj` ..... MSVS 2010 Project File
  - `Properties\` ..... Required files for C# application build

2 Design Concepts

2.1 Directory Structure

Any application built using the PPS API must include the following files and folders (copied from the appropriate `bin\` directory for your system architecture) in the same directory as the application executable:

- `PPSDaqApi.dll`
- `HwPlugins\*.dll`
- `ProcPlugins\*.dll`

In addition to these binary files, your application will need access to the set of configuration files provided with your system. All configuration files should be located in the same directory, called "`setup\`" as the main configuration file identified by the "`.cfg`" extension. You can find your system's configuration files either on the installation CD provided by PPS or in the main Chameleon TVR folder:

```
C:\Program Files\Pressure Profile Systems\Chameleon TVR\setup\
```

The PPS API will also auto-create a `log\` directory in the same directory as the `setup\` folder described above, even if logging is not enabled.

The PPS API must have write-access to the `setup\` and `log\` directories, which is not given by default to some Windows folders when User Account Control (UAC) is enabled in Windows Vista and higher. It is the developer's responsibility to manage all relevant permissions for read- and write-access, such as by using a folder in the Windows hidden `%APPDATA%` directory.

2.2 Typical Usage

The PPS API is a minimalist design intended only to provide access to all data returned from the PPS hardware. It is up to your own application to perform any analysis, visualization, and/or storage of the data. A typical application workflow might be the following:

- Initialize connection to hardware for your configuration with `ppsInitialize()`
- Allocate memory transfer buffers based on `ppsRecordSize()`
- Start acquisition using `ppsStart()`
- Continue to acquire data, either in a while() loop or an appropriate Timer object
    - Call `ppsFramesRead()` to see if new data is available
    - If so, call `ppsReadData()` to extract the desired amount data
    - Perform processing, visualization, storage, etc.
    - Continue acquisition until your program is complete based on user input, timer, or other control flow
- Stop acquisition using `ppsStop()`

2.3 Data Format

Sensor data is grouped in frames consisting of a 32-bit integer representing the time (in milliseconds) since the API was initialized, and a collection of 32-bit floating-point values representing the sensor data. For calibrated output, sensor data is in units of either psi. or pounds, depending on the system type.

3 Function Reference

In the following function descriptions, "`bool`" is used to indicate the `bool` type in C# and the `BOOL` type in C/C++, and "`string`" is used to indicate the `string` type in C# and the C-style `wchar_t*` type redefined as `string_t` in `PPSDaq.h`. For sensor frames, "`int`" is used for timestamps and "`float`" for sensor data, though these are typedef'ed as `time_stamp_t` and `string_t`, respectively, in `PPSDaq.h`. In addition, the "`pps`" prefix is not used in C# function calls, but rather the static `pps` class is used, so calls will be in the form "`pps.FunctionName()`".

3.1 bool ppsInitialize(string configFile, int logLevel)

Description:
Initializes a connection to the PPS hardware using the provided configuration file and at the specified log level. NOTE: Logging can affect system stability and should not be used without first consulting PPS.

Arguments:

`configFile` (string)..... Path to the main configuration file for your system, identified by the extension "`.cfg`"

`logLevel` (int)..... Verbosity level for logs from 0 (disabled) to 5 (maximum verbosity). Note that this should not be set to a non-zero value without first consulting with PPS.

Returns:..... True on Success

3.2 bool ppsStart()

Description:
Starts the PPS hardware collecting and streaming data.

Arguments:..... None

Returns:..... True on Success

3.3 bool ppsStop()

Description:
Halts acquisition and disconnects from PPS hardware.

Arguments:..... None

Returns:..... True on Success

3.4 bool ppsGetRecordSize()

Description:
Retrieves the Record Size for one frame, which consists of one reading from each of the elements in all connected sensors.

Arguments:..... None

Returns:.....Size of a single frame of sensor data

3.5 bool ppsFramesReady()

Description:
Retrieves the number of frames available in the buffer to be read.

Arguments:.....None

Returns:.....Number of available frames

3.6 int ppsGetData(int nFrames, int* times, float* data)

Description:
Retrieves at most the specified number of frames into the provided vectors for time and data. The system will at most return the maximum number of available frames. It is the calling program's responsibility to ensure that sufficient memory is already allocated for the maximum number of frames to be returned.

Arguments:

`nFrames` (int) .....The maximum number of frames to read

`times` (int array) .....Destination for the list of times, one time per frame

`data` (float array).....Destination for the sensor data. The ordering of data within a frame matches the element ID used by the PPS Chameleon TVR software, or PPS can provide a diagram of the element ordering in your sensor. To access element with ID *`iElem`* in frame *`iFrame`* where each frame is of size *`recordSize`*, use `data[iFrame * recordSize + iElem]`.

Returns:.....Number of frames actually written to destination arrays

3.7 int ppsGetMaxSignal()

Description:
Retrieves the maximum raw signal value that may be returned by the attached hardware. Not applicable for calibrated systems.

Arguments:.....None

Returns:.....Maximum raw signal value

3.8 void ppsSetBaseline()

Description:
Resets the sensor zero output value based on the current sensor data.

Arguments:.....None

Returns:.....None

3.9 void ppsClearBaseline()

Description:
For calibrated systems, resets the baseline to the default value found during calibration. For uncalibrated systems, clears the baseline by setting all values to zero.

Arguments:.....None

Returns:.....None

3.10 bool ppsIsCalibrated()

Description:
Indicates whether the data being returned is calibrated or raw.

Arguments:.....None

Returns:.....True for calibrated data

3.11 string ppsGetLastError()

Description:
Retrieves the most recent error message generated by the attached PPS system and clears the error flag.

Arguments:.....None

Returns:.....Error message

3.12 string ppsDirectCommand(string command)

Description:
When applicable, used to provide hardware-specific commands for certain systems in advanced application. Requires additional documentation from PPS for your specific configuration and is not needed for normal operation.

Arguments:
`command` (string).....Command string for low-level command

Returns:.....Result of low-level command (empty on failure)

4 Sample Projects

Sample application projects for 32-bit and 64-bit systems are provided for Microsoft Visual Studio 2010. The provided code has also been successfully built using MSVS 2008 and MSVS 2012 for C++ and C# projects, and 32-bit C++ code has been built using Visual C 6.0, MSVS 2003, MSVS 2005, and Borland C++ Builder 5. While customers have reported successfully building applications using other languages and environments, such as VisualBasic, Java, LabVIEW, MATLAB, and gcc, PPS does not provide support for these platforms.

All three sample projects implement the same simple console app, which connects to the provided sample configuration file which emulates data for an uncalibrated 16x16 array sensor. The sample application will connect to the configuration, start and retrieve data for a time, and then disconnect.

When reading data, the sample app alternates between requesting as many frames of sensor data as its buffers will hold and requesting only as many frames as are available based on `ppsFramesReady()`. This is not necessary in practice, and only serves to demonstrate both means of reading data.

The sample app also sets the sensor baseline using `ppsSetBaseline()` halfway through its data simply to demonstrate the function call. An application would typically set the baseline at startup and/or when requested by the user.

4.1 C++ Projects

Two different C++ projects are provided as examples for different ways of connecting to the PPS API Library, static and dynamic. The two projects utilize the same source code (`PPSSampleApp.cpp`) and header file (`PPSDaq.h`).

4.1.1 Static Linking

For C++ projects using Microsoft Visual Studio, you can use the static import library `PPSDaqApi.lib`. This is merely an import library, so the `PPSDaqApi.dll` must still be present.

The `PPSDaq.h` header file includes summaries of the different functions, and also includes a number of declarations for use in dynamically-linked applications. These declarations are only compiled when the `PPS_DYNAMIC_LINK` constant is defined, which means they are omitted in the static example.

4.1.2 Dynamic Linking

The dynamically-linked C++ project is appropriate for use with other compilers for which the Visual Studio .lib file is not compatible. Its project configuration defines the preprocessor constant `PPS_DYNAMIC_LINK` to include a number of declarations specific to dynamic linking. It also must use the source file `PPSDaqDyn.cpp`, which defines the functions `ppsConnectToApi()` and `ppsDisconnectApi()`, which take care of loading in the exported library functions from the DLL. This allows the same function prototypes to be used, as well as the same sample code, as for the static application.

4.2 C# Project

The sample C# project is identical to the C++ projects in terms of functionality. The `Program.cs` file includes the sample app source code, and `PPSDaq.cs` provides a C# interface to the PPSDaqApi library that can be re-used in other applications. Simply add the `PPSDaq.cs` file to your project and add '`using PPSDaq;`' to any C# files needing access to the PPS API.