# Getting Started

## Requirements

* Visual Studio 2013+
* .NET Framework 4.0+
* ProMetric 10.11+

## Dependencies

When creating an application that uses the ProMetric API, any supporting dependencies for the core API libraries (PMEngine.dll, RadiantCommon.dll, etc.) need to be located by the .NET Common Language Runtime (CLR) at runtime of your application. There are a couple of ways to ensure these dependencies are found:

* Copy all files from the ProMetric installation directory to the output directory of your application. By default, the .NET runtime will use this folder during its search for the supporting dependencies.
* (Recommended) Add a reference to your project to **Radiant.AssemblyLoader.dll**, located in the ProMetric installation folder, and then create an instance of the assembly loader class before accessing any of the types in the API.

Use of the assembly loader for locating supporting dependencies is preferred as it is automatic and allows the most flexibility.

**Assembly Loader Setup**

1. Add a reference to **Radiant.AssemblyLoader.dll** to your project.
2. Create an instance of the assembly loader before the application has a chance to load *any API libraries*. For VB.NET, a good place to do this is in the **ApplicationEvents.vb** file, within an event handler for the Startup event:

Private Sub MyApplication\_Startup**(**sender As Object**,** e As ApplicationServices.StartupEventArgs**)** Handles Me.Startup

'No need to call any methods, instantiation is sufficient

Dim loader As New Radiant.AssemblyLoader**()**

End Sub

* **Note**: You can specify a path to the ProMetric installation folder, which will cause the assembly loader to search for supporting dependencies in that location—if a path is not provided, the default installation path is used instead.

## Working with the API

**Initialization**

The API revolves around the core PMEngine type, which ties the camera and ProMetric software functionality together.

When using the API, before trying to access any functionality, you first need to perform initialization for PMEngine so that internal processes are configured correctly. This initialization only needs to happen once per session, i.e., it is *unnecessary* to initialize the engine before performing every major API operation, such as capturing a measurement.

To perform initialization, first create an instance of the **PMEngine** class, set your calibration and measurement databases, and then call the InitializeCamera method:

* **Note**: Only the camera calibration file needs to be set for initialization, since the camera relies on this database for proper setup. The measurement database can be set after initialization.

'Assuming mPMObj is an instance member

mPMObj **=** New PMEngine**()**

Dim calPath As String **=** "C:\Radiant Vision Systems Data\ProMetric\CalibrationData\PM Calibration Demo Camera.calx"

mPMObj.SetCalibrationDatabase**(**calPath**)**

mPMObj.InitializeCamera**()**

Dim measDbPath As String **=** "C:\Radiant Vision Systems Data\ProMetric\UserData\Sample.pmxm"

mPMObj.MeasurementDatabaseName **=** measDbPath

**Shutdown**

Along with initialization, performing Shutdown is also a necessary step that needs to occur at the end of the lifecycle of your application (or if you want to switch cameras, for example). Simply call Shutdown on the PMEngine instance:

mPMObj.Shutdown**()**

## ****Measurement Setups****

Before you start taking measurements, you need to set the measurement setup that you would like to use for the measurements, which contains all the information needed for the measurement conditions.

You can retrieve measurement setups from the database via the ReadMeasurementSetupfromDatabase method on an instance of PMEngine, which takes the measurement setup ID as its single parameter.

The measurement setup ID is a unique identifier for the measurement setup. A list of IDs for all the measurement setups stored in the measurement database can be obtained by calling GetMeasurementSetupList on an instance of PMEngine.

* **Note**: We recommend first creating your measurement setups in the ProMetric software, then saving and exiting the software, rather than creating the measurement setup programmatically. This is the most convienent and prevents configurational issues.

After retrieving the list of measurement setup IDs, iterate through the list to find the one you are interested in:

Dim measSetups**(-1)** As ListItem

mPMObj.GetMeasurementSetupList**(**measSetups**)**

Dim measSetup As MeasurementSetup

For Each **(**ms in measSetups**)**

If Not ms.Description **=** "MyMeasSetup" Then Continue

measSetup **=** mPMObj. ReadMeasurementSetupfromDatabase **(**ms.ID**)**

Next

You can update the properties on an instance of MeasurementSetup for changing the conditions of the measurement, such as the f-stop, subframe, and so forth:

'Set the f-stop

measSetup.LensfStop **=** **3.3;**

'Set the focus distance (default units: meters)

measSetup.LensDistance **=** **1.0;**

'Set subframe (cropping) region

measSetup.Subframe **=** new System.Drawing.Rectangle**(100,** **100,** **300,** **300);**

You can then save the measurement setup back to the database by calling WriteMeasurementSetupToDatabase on an instance of PMEngine:

mPMObj.WriteMeasurementSetupToDatabase**(**measSetup**)**

## MeasurementF

**Capturing Measurements**

To capture a measurement, call TakeMeasurementF on a PMEngine instance. This method takes a MeasurementSetup object for configuration of the measurement conditions and returns a MeasurementF object:

'Only the measurement setup and description are important parameters.

'The remaining arguments passed in can be discards.

Dim meas As MeasurementF **=** mPMObj.TakeMeasurementF**(**measSetup**,** "Description"**,** **0,** **0,** ""**,** ""**)**

**Retrieving Measurement Data**

You can retrieve either the tristimulus data or the calculated brightness and chromaticity values.

To get the tristimulus data, call the GetTristimulusF instance method, which returns a 2D array containing the tristimulus data for a specified Tristimulus value:

'Retrieve tris Y values for entire image

Dim trisY**(,)** As Single **=** meas.GetTristimulusArrayF**(**MeasurementBase.TristimulusType.TrisY**)**

To get the brightness and chromaticity values instead, you can iterate through each pixel in the image array and call the GetColor instance method:

'Retrieve lv, cx, cy for each pixel in the entire image

For i As Integer **=** **0** To meas.NbrCols **-** **1**

For j As Integer **=** **0** To meas.NbrRows **-** **1**

Dim color As CIEColor **=** meas.GetColor**(**i**,** j**)**

Dim lv As Float **=** color.Lv

Dim cx As Float **=** color.Cx

Dim cy As Float **=** color.Cy

Next

Next

### **Saving/Reading Measurements**

To read and write measurements to and from the database, call the shared methods ReadMeasurementFfromDatabase and WriteMeasurementFtoDatabase on the PMEngine class. ReadMeasurementF requires the measurement ID, which is obtained in the same manner as a MeasurementSetup object:

Dim items As New ListItem[**-1]**

PMEngine.GetMeasurementList**(**ref items**,** false**)**

Dim id As Integer**;**

For Each item As ListItem In items

If item.Description **=** "myMeas" Then

id **=** item.Id

End If

Next

Dim meas As PMMeasurementF **=** PMEngine.ReadMeasurementFfromDatabase**(**id**)**

# Next Steps

## ProMetric API Example

The best next step for gaining familiarity with our API is to open and run the example solution that was installed with the SDK package.

The example solution demonstrates the various concepts discussed in this guide and lets you run and modify the code to get a better understanding of how to use all the pieces together.

**ProMetric API Specification**

There are many operations and other types available to you through the API. For a detailed breakdown of these, refer to the ProMetric .NET API Specification document that was installed alongside this walkthrough guide.