PrairieLink

PrairieLink is an API (Application Programming Interface) which allows direct two-way communication with Prairie View without understanding the underlying communication protocols. Put simply, it uses script commands to communicate with Prairie View via TCP/IP. For more information please see the Script Command Reference.

Instantiating PrairieLink

Initializing PrairieLink will differ based on what language you are using, here are some examples:

VB.NET

First add PrairieLink.dll as a project reference, then call:
Dim pl As New PrairieLink.Application

MATLAB

pl = actxserver('PrairieLink.Application');

Python

import win32com.client
pl = win32com.client.Dispatch("PrairieLink.Application")

64-bit

PrairieLink64.Application can be substituted for **PrairieLink.Application** when applicable.

Using PrairieLink

Once a PrairieLink object has been instantiated you must connect to Prairie View using the Connect function:

pl.Connect()

Optionally include the IP address parameter to connect to Prairie View running on another computer:

pl.Connect("127.0.0.1")

If a remote access password is enabled (default) on the Edit Scripts dialog, accessible under the 'Tools' -> 'Scripts' menu, that password will need to be passed as the second optional parameter:

pl.Connect("127.0.0.1", "password")

The Connect function will return a Boolean value: true if the connection succeeded or false if the connection failed.

Once connected any of the methods described below can be used, without an active connection to Prairie View they not work.

For example, the following will return the current dwell time in microseconds:

pl.GetState("dwellTime")

When finished, use the Disconnect method to let Prairie View know so it can clean up afterwards; failure to disconnect properly could result in undesirable behavior.

pl.Disconnect()

PrairieLink Methods

Connect(*optional* IPAddress as String) as Boolean

This function connects to the currently running instance of Prairie View at the address provided and returns true if the connection was successful or false if the connection failed. If the optional IP address is not provided PrairieLink will connect to the currently running instance of Prairie View on the same machine. Note that some methods only work on the same machine, those exceptions will be mentioned in the descriptions of any affected methods.

Connected() as Boolean

This function returns true if a connection has been made or false if no connection is present.

Disconnect()

This method disconnects from Prairie View. Prairie View needs to do a little bit of cleanup when a communication channel isn't being used anymore, so failure to disconnect, particularly after numerous connections are made in the same session, could result in undesirable behavior.

SendScriptCommands(commands as String) as Boolean.

This function sends a string of script commands to Prairie View which will then be run. If the script commands run successfully the function will return true, otherwise it will return false. For a complete listing of available script commands please reference the script command reference. Some script commands can return values; each time a script command returns a value an event will be raised for which a handler can be registered to do something with the value. There are also specific functions to call a single script command which returns a value without using events. See the event handling section below for more details.

PixelsPerLine() as Integer

This function returns the number of image pixels in the width/X dimension as an integer.

LinesPerFrame() as Integer

This function returns the number of image pixels in the height/Y dimension as an integer.

GetState(key as String, *optional* index as String, *optional* subindex as String) as String

This function returns the value for the specified state key as a string. Some state keys are indexed or

subindexed which requires the optional parameters to be used. Use the PVStateShard section of the environment file as a reference.

GetMotorPosition(axis as String ('X', 'Y' or 'Z'), *optional* deviceIndex as Integer) as Double

This function returns the position of the specified axis as a double precision floating point value. On some systems there are multiple devices for an axis so the optional device index parameter can be used to differentiate between them. The device index parameter is zero index, so the first device is zero.

GetImage(channel as Integer) as Integer(,)

This function returns a two dimensional array containing the image data for the specified channel. In cases where there are multiple samples for a pixel the values are summed, not averaged. Note that this function will only work on the same computer that Prairie View is running on, it will return an array of zero's if run on a different computer.

GetImage_2(channel as Integer, pixelsPerLine as Integer, linesPerFrame as Integer) as Integer(,)

This function is a slightly more efficient version of the GetImage function to be used in cases where the image dimensions are known. This eliminates the need to poll Prairie View for the image dimensions, saving a little time. Note that this function will only work on the same computer that Prairie View is running on, it will return an array of zero's if run on a different computer.

DroppedData() as Boolean

This function returns true if the current acquisition has dropped data or false if all the data has been saved successfully.

ReadRawDataStream(*out* samplesRead as Integer) as Short()

In order to use this function raw data streaming needs to be enabled by calling the script command '–srd true', otherwise this function will never return any samples. This function will return an array of samples as they are read off of the acquisition card. This function will guarantee that the chunks of data returned will form contiguous whole frames. If frames of data are not read off fast enough, only the most recent frames will be kept and the stream will omit the frame(s) in between (unless the buffer frames parameter passed to the –srd command is non-zero which will return all data and stop streaming if the data is not read out fast enough to keep up). Making sense of the data stream requires knowledge of the acquisition being run, like how many pixels are in a line, how many lines are in a frame, how many samples are acquired for each pixel, and how many channels of data are being acquired. All of these things can be polled for using script commands. For example a galvo mode acquisition for two channels at a 4 µs dwell time will have two 16-bit values for each sample (one for each channel) and 20 samples for each pixel (4/0.2). The SamplesPerPixel function can help figure this out for other acquisition modes. Multiple channels for a camera based acquisition will not be interleaved. Using this function will definitely require some trial and error as every application will be different.

ReadRawDataStream_2(bufferSizeInSamples as Integer, *out* samplesRead as Integer) as

Short()
Same as above, but allows a buffer size other than the default 8 MB to be used. A smaller buffer will improve performance for getting individual chunks of data, but can decrease overall throughput if the goal is to keep up with the acquisition.

ReadRawDataStream_3(processId as Integer, bufferAddress as Int64, bufferSizeInSamples as Integer) as Integer
This is a lower level wrapper of the ReadRawDataStream script command which doesn't use a secondary memory buffer managed by PrairieLink. This option requires a bit more programming experience to correctly pass the buffer location/pointer, but provides a significant performance improvement by not requiring a second memory copy (or worse a slower data marshalling process). Returns the number of samples read into the buffer provided.

SamplesPerPixel() as Integer
This function returns how many samples are acquired for each pixel in the image. This is mainly useful when used in conjunction with the ReadRawDataStream function to figure out how to parse the raw data stream.

SetTimeoutAndRetries(timeout as Integer, retries as Integer)
This method sets the timeout and number of times Prairie Link will attempt to resend commands to Prairie View without acknowledgment. If Prairie Link has not received an acknowledgment from Prairie View within the timeout (in milliseconds), it will re-send the most recent command until it receives acknowledgment or reaches the maximum number of retries. If no acknowledgment is received after the maximum number of retries, Prairie Link will give up and return control to the calling program. Setting the timeout to 0 disables the timeout, and Prairie Link will wait until it receives acknowledgment from Prairie View no matter how long it takes. For most uses it is not necessary to set these values, as the default values of 2000 milliseconds and 5 retries are appropriate for all but very long commands (e.g. MarkPoints or SetCustomOutput scripts with many values).

PrairieLink Event Handling

There is currently only one event raise by PrairieLink which is used to handle script command responses asynchronously. There are specific methods to call a single script command and return the response synchronously in the methods section above. The following code will register for script command response events in VB .NET:

```
AddHandler pl.ScriptCommandResponse, AddressOf ScriptCommandResponseHandler
```

Where the handler subroutine looks something like:

```
Private Sub ScriptCommandResponseHandler(ByVal response As String)
…
End Sub
```

To register for the same event in MATLAB the code would look like this:

pl.registerevent({'ScriptCommandResponse' 'ScriptCommandResponseHandler'});

Where the handler subroutine would be in an M file with the same name, in the current working directory, and look something like:

function HandleScriptCommandRequest(source, eventId, response, args, type)
…
end

The response parameter in both cases contains Prairie View's response as a string.