JVA Perimeter Patrol™
High Level Interface
Programmers Reference Guide

Programmers' reference guide to developing custom applications using JVA Perimeter Patrol HLI

Version: 5.1
Date: October 2012
By: Benjamin Boyle

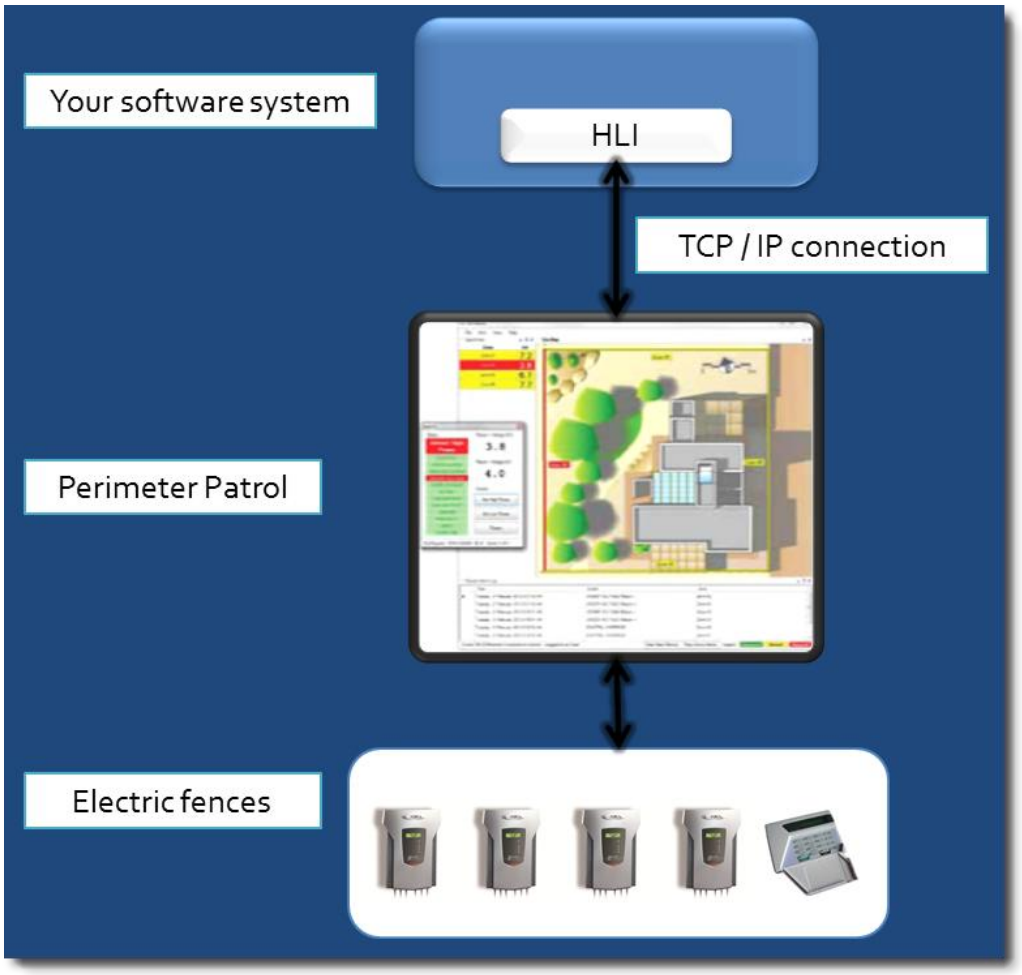


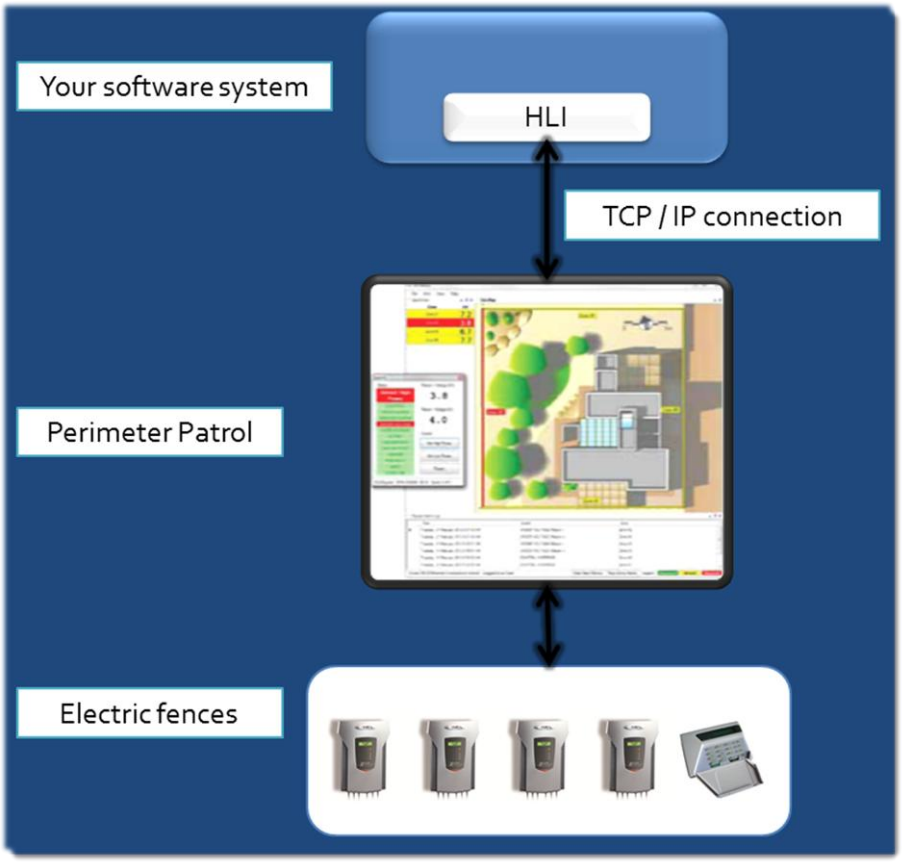
Table of Contents

1 Introduction

JVA Perimeter Patrol, a software system for monitoring and commanding JVA Security Electric Fence Devices, is designed exclusively for **JVA Technologies Pty Ltd** by **Pakton Technologies Pty Ltd** of Brisbane, Australia.

The **JVA Perimeter Patrol High Level Interface** (HLI) allows you to monitor and command JVA Security Electric Fence systems from within your own software systems.

Your software connects to the JVA Perimeter Patrol program over an encrypted TCP/IP connection. The HLI component that we provide to your system programmers makes this easy. Everything your programmers need to know is included in this document and the documents named in the **Required reading and setup** section below.



1.1 Required reading and setup

This document assumes that you have already read and understood the following documentation:

- JVA Perimeter Patrol User Manual
- JVA Perimeter Patrol Configuration Manual

1.2 Scope

Read this document if you want to create custom software programs that connect to JVA Perimeter Patrol using the High Level Interface (HLI).

If you want to connect two instances of Perimeter Patrol together in Server / Client configuration, please read section **Connecting two instances of Perimeter Patrol in Server / Client configuration using the HLI** in the **JVA Perimeter Patrol Configuration Manual.**

1.3 Contact details

Position	Name	Email
Author of this document	Benjamin Boyle	bboyle@pakton.com.au
Pakton Technologies CEO	Paul Thompson	sales@pakton.com.au
Pakton Technologies General Manager	Kayleen Thompson	sales@pakton.com.au

2 Getting started

This document assumes that you have a Security Electric Fence installation contractor who knows how to set up the electric fence systems as well as configure and command them using the JVA Perimeter Patrol software package.

To begin, you will have an electric fence set up already, or you will have a small testing rig in your office with a short section of fence and a number of JVA electric fence energisers or other devices. You will have a computer with JVA Perimeter Patrol connected to the devices and you will have already sent commands to the devices from that computer.

2.1 Configuring Perimeter Patrol

Since you will be connecting to Perimeter Patrol with the HLI, you need to configure it to accept the connections. There are two small jobs you need to do:

- Configure Perimeter Patrol to accept HLI connections
- Configure the user accounts in Perimeter Patrol.

2.1.1 Configuring Perimeter Patrol to accept HLI connections

- Click **Setup -> System Configuration**
- Open the **HLI Server** tab
- Choose **HLI Server Enabled**
- Set the **TCP Listening Port** to a value between 1,000 and 10,000.
- Set a value for the **Authentication Key**. It doesn't matter what value you use for the Authentication Key during testing, but you should use the "Generate" button to make a complicated key for your production sites, because this key is like a password code that prevents unauthorized software from connecting to your Perimeter Patrol.
- Shut down and restart Perimeter Patrol as this is required for the new settings to take effect.

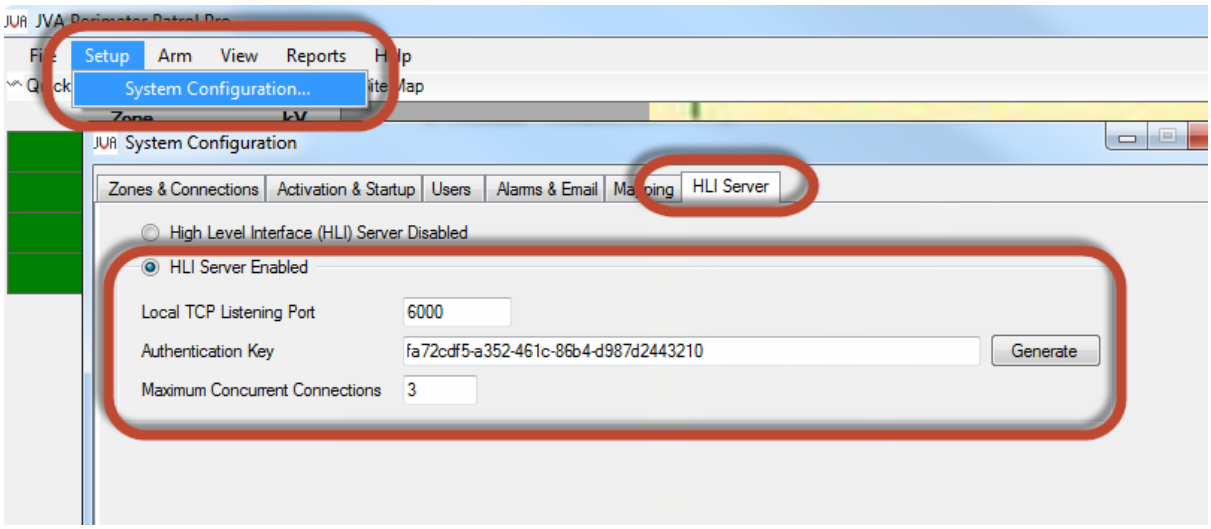


Figure 1 - Enabling HLI Server to accept HLI connections

2.1.1.1 Use a static (fixed) IP Address for Perimeter Patrol

Ask your system administrator to configure your network router so that it assigns a static IP Address to the computer running Perimeter Patrol. Your system administrator should be able to tell you the new IP Address, but if he does not, you will need to find out. Here's how:

- On the computer's start menu, click **Start -> Run** and enter **cmd.exe**.
- If that doesn't work, enter **C:/windows/system32/cmd.exe**
- If you don't have "Run" on your start menu, you can find C:/windows/system32/cmd.exe in explorer and double-click the file to start it.
- The window pictured below will open.
- Type **ipconfig** and press Enter.
- Write down the IPv4 address. This is the IP Address that your client software needs to know.
- Sometimes your computer will have more than one IPv4 address, if it has more than one network adapter device. Choose the address corresponding to the correct network adapter device.

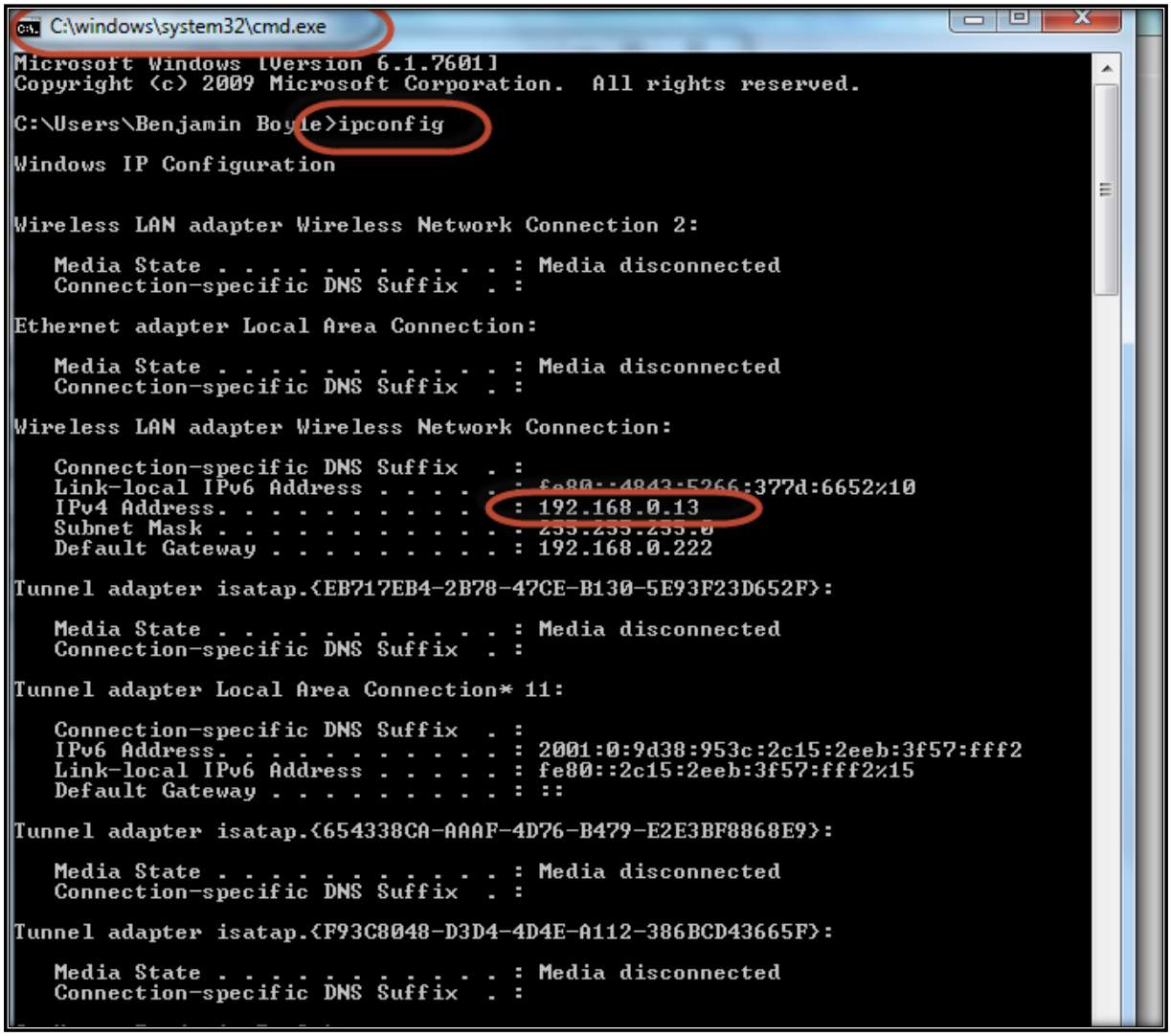


Figure 2 - Finding out the computer's static IP Address

2.1.2 Setup Perimeter Patrol users

Perimeter Patrol will not allow any HLI connection to receive information or receive commands unless the HLI connection provides a username and password that matches one those configured in Perimeter Patrol.

Your software will have to use these usernames and passwords to access Perimeter Patrol over the HLI. You can hard-code the usernames and passwords in your software system if you like, but we recommend you require your users to type them in manually.

You can find out how to setup Perimeter Patrol users in the **JVA Perimeter Patrol Configuration Manual.**

Remember that there are three levels of security access permission:

1. Users – may view information
    a. Accessible by your software via HLI
2. Supervisors – may clear alarms and send commands
    a. Accessible by your software via HLI
3. Administrators – may configure the JVA Perimeter Patrol software
    a. Not accessible to your software. JVA Perimeter Patrol configuration must be done on the JVA Perimeter Patrol computer by an administrator.

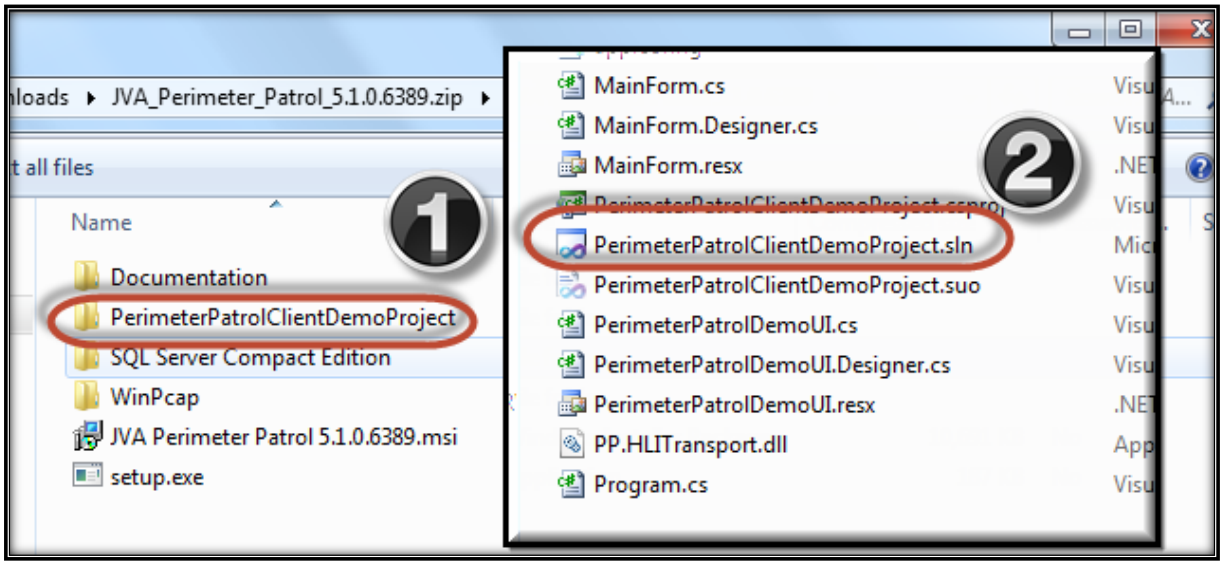
Your HLI software will be given permissions according to the level of authentication it can provide.

3 Using the HLI Demo Project

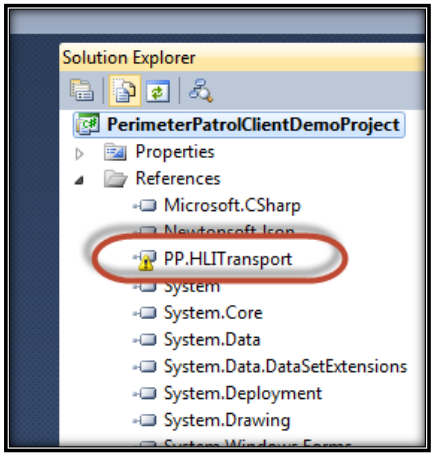
Now that you have Perimeter Patrol setup and waiting to accept connections, it's probably going to be easiest for you if you use the demo HLI project to make your first connection, since all the work is done for you.

You need to have Visual Studio 2010. You can download it from http://download.microsoft.com/download/B/1/7/B17C731C-3161-45C0-AC16-56C81BAAF85C/vs_premiumweb.exe if you don't already have it on your computer.

Using the installation cd, open the **PerimeterPatrolClientDemoProject** folder and copy it to a place your computer's hard disk. In the place you have saved it, double-click **PerimeterPatrolClientDemoProject.sln**

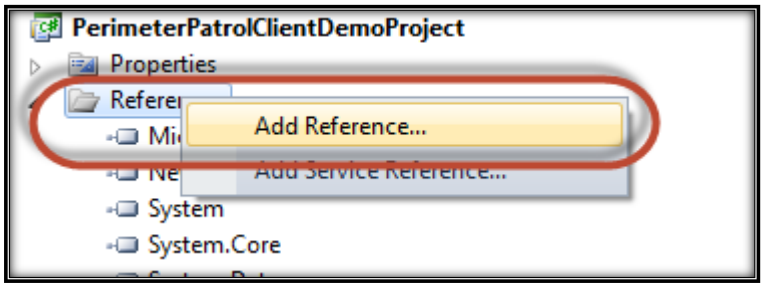


After the project loads, you can click References in the Solution Explorer, and you will see that there is a reference error:

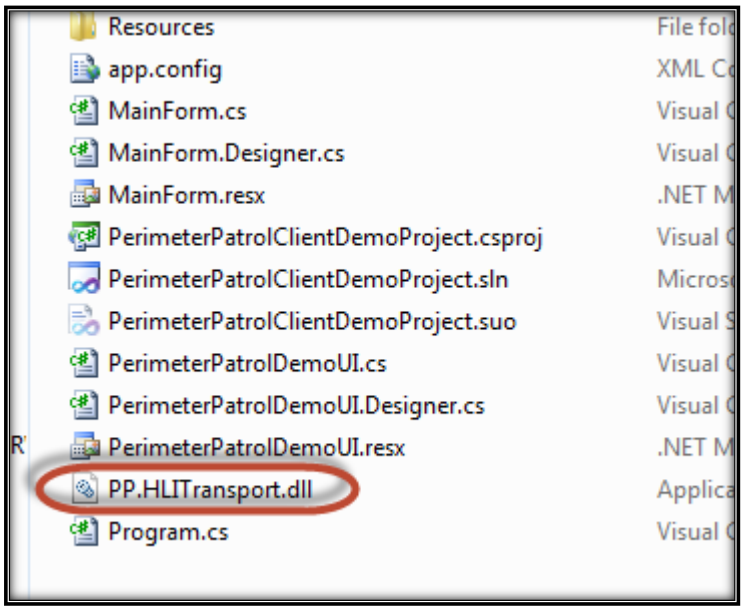


This was done on purpose so that you can learn how to reference the .dll file that is provided for you. Right-click on the **PP.HLITransport** reference and choose **Remove**.

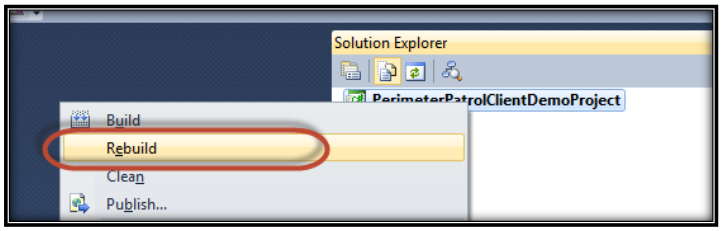
Now right-click on **References** and choose **Add Reference**



Choose **Browse** and navigate to the place where you saved the PP.HLITransport.dll file.



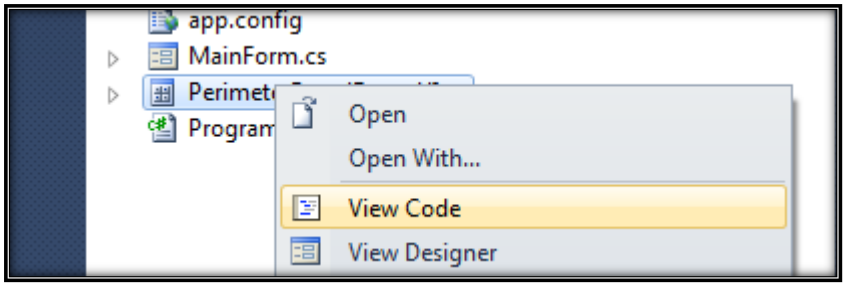
Now that you have added PP.HLITransport.dll as a reference to the project, it should compile successfully. Right-click on **PerimeterPatrolClientDemoProject** and choose **Rebuild**.



The project should compile successfully.

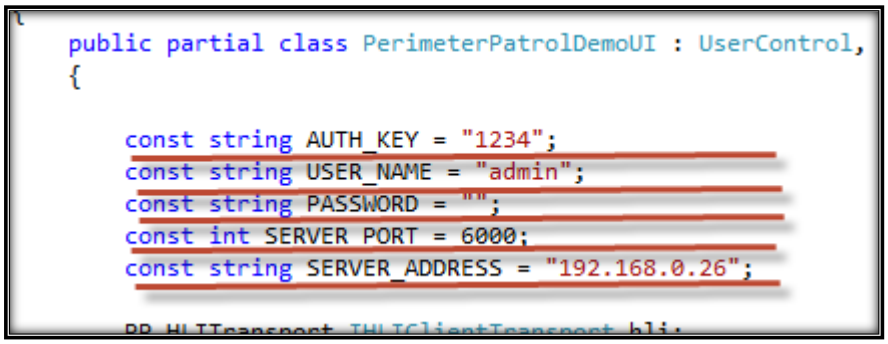
Now you need to setup the connection parameters so that it can connect to Perimeter Patrol Server.

Right-click **PerimeterPatrolUI.cs** and choose **View Code**



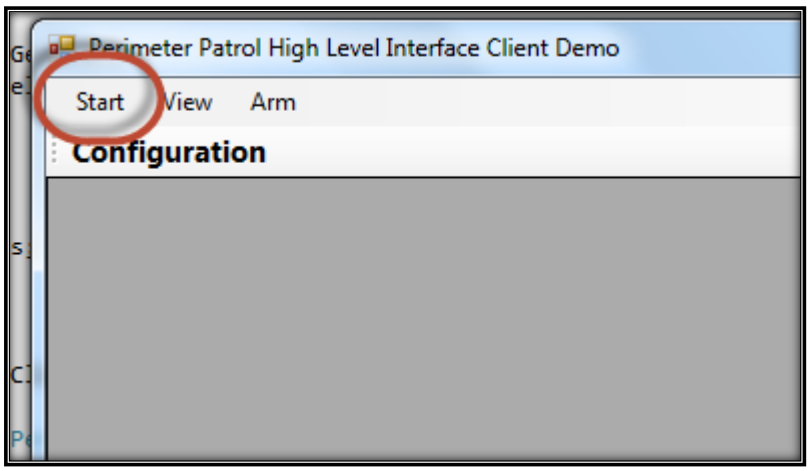
In the code window that opens, modify the parameters shown below with the values needed to connect to Perimeter Patrol Server.

- Set the USER_NAME and PASSWORD parameters to be the same as one of the user accounts setup in the Perimeter Patrol's configuration settings.
- Set the SERVER_ADDRESS parameter to the static IP address of the computer running Perimeter Patrol
- Set the SERVER_PORT parameter to the same value as the "TCP Listen Port" that you specified in Perimeter Patrol's configuration.
- Set the AUTH_KEY parameter to the same value as the "Authentication Key" that you specified in Perimeter Patrol's configuration.



Now press the **F5** key to start the demo program.

When the program runs, click **Start** to make it connect.



If the programs connects and begins to display data, then you know that you have the connection settings correct. If not, you will have to figure out what you need to change.

3.1 The Interpreter object

The Perimeter Patrol Demo HLI client project contains an Interpreter class which is used to gather and make sense of the information received over the HLI connection. This class has been written specifically for the purpose of helping you to understand how to interpret and use incoming HLI information in your own custom projects.

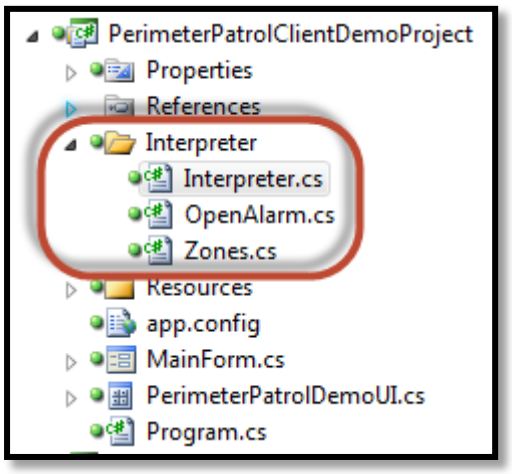


Figure 3 - The Interpreter object

4 Using the HLI in your custom software

4.1 Step 1 - Adding the HLI to your development environment

1. In your visual studio project, add a reference to the PP.HLITransport.dll.
2. This dll targets the .Net Framework Version 4. Ensure your visual studio project targets the same .Net Framework version.

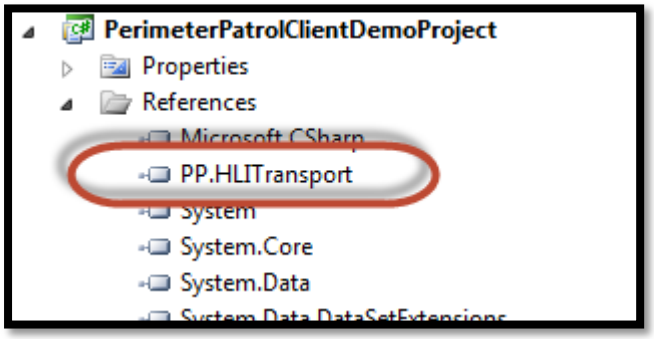


Figure 4 - A reference to PP.HLITransport.dll

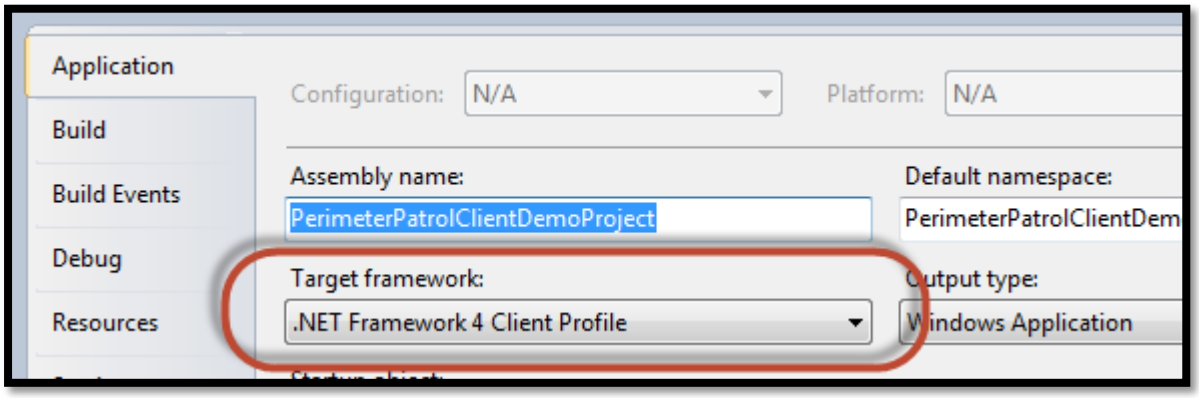


Figure 5 - Referencing .Net Framework version 4. It's not necessary to use the client profile.

4.2 Step 2 - Creating a HLITransportUser object

Your software uses the HLITransport to communicate with the JVA Perimeter Patrol software. Therefore it is considered to be a HLITransportUser. Please create a class in your project that inherits the `PP.HLITransport.IHLIClientTransportUser` interface.

When you inherit this interface and implement the functions it requires, you give the HLI transport object a hook to pass information into your system or request information from your system.

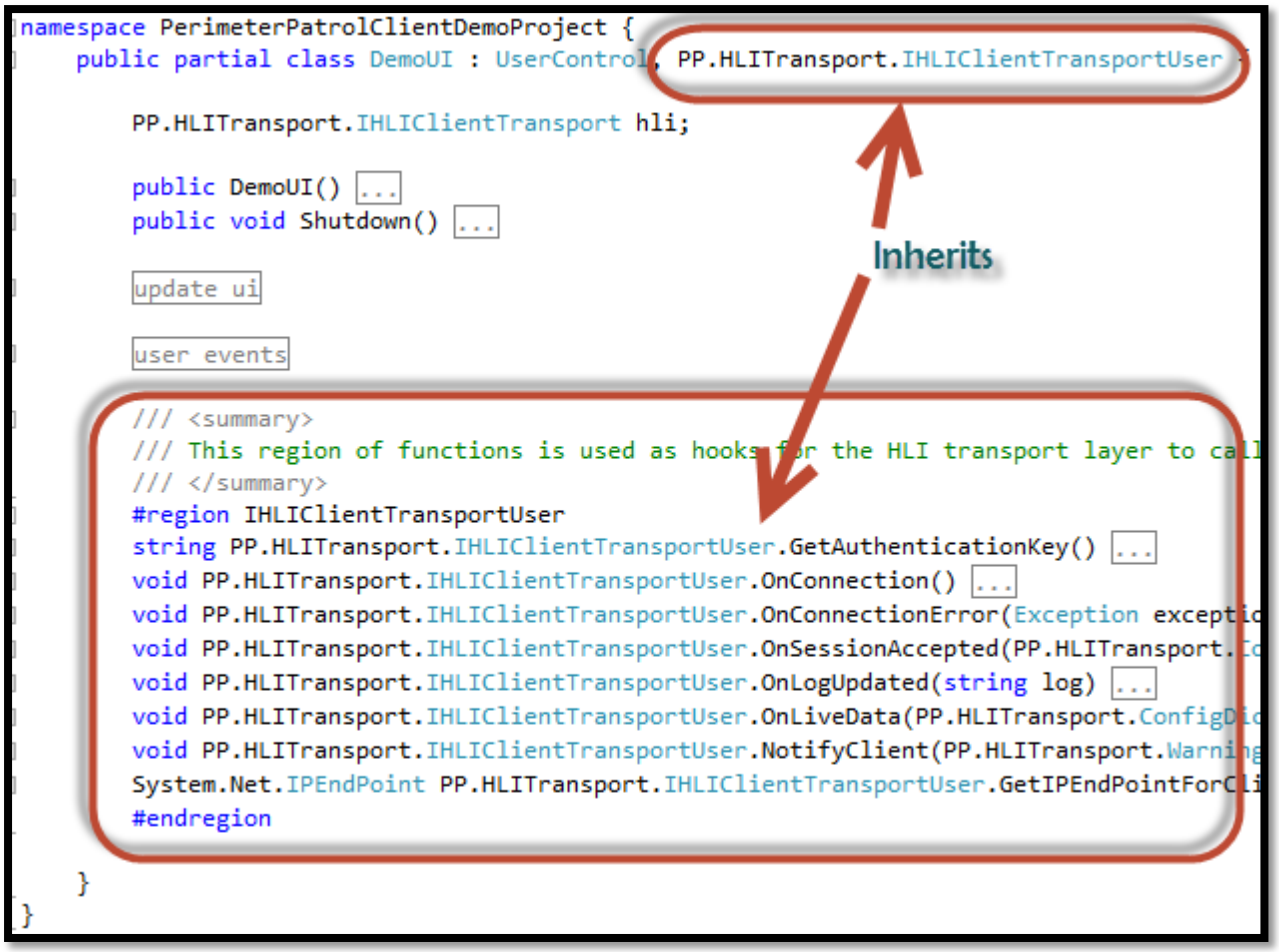


Figure 6 - Create a class that inherits PP.HLITransport.IHLIClientTransportUser

Now you are ready to create and dispose the HLITransport object. First notice that to create the HLI Transport object you call a static function inside **`PP.TransportLayer`**, passing in an object that inherits the **`PP.HLITransport.IHLIClientTransportUser`** interface. This is the object that will be used by the HLI Transport object as the hook it needs for posting and requesting data.

When you **Start** the object, it will immediately begin attempting to connect to the JVA Perimeter Patrol software. If it fails to connect, it will simply keep on trying to connect over and over again. If a connection is established and then fails, it will continually attempt to reconnect.

There is no Stop function, and an exception will be thrown if you attempt to call the **Start** function more than once.

Notice that we take special care to **Dispose** of the HLI Transport object when we are finished with it. It's very important to **Dispose** it when you are finished with it, because it needs to shut down all the threads that it had running.

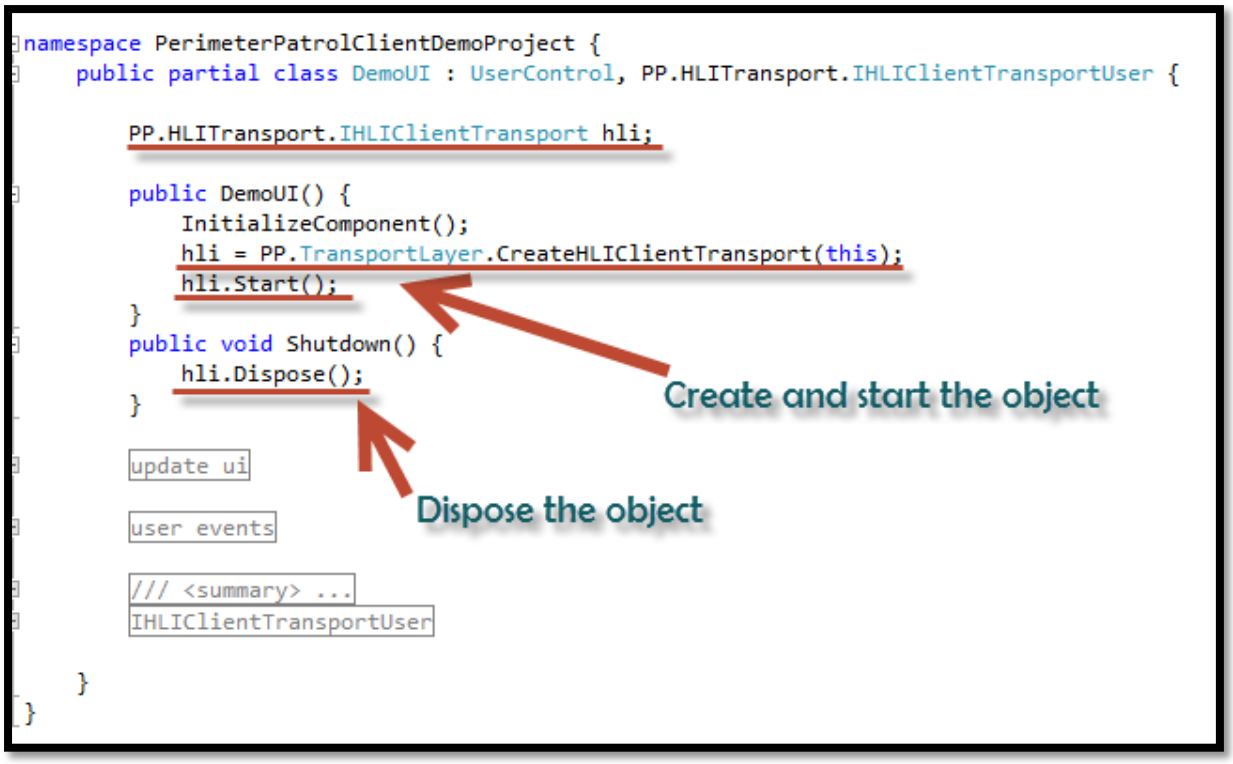


Figure 7 – Creating, starting and disposing the HLI Transport object

There are two more steps left for you to understand. The next step, Step 3, is about implementing all the functions required by PP.HLITransport.`IHLIClientTransportUser` so that the HLI Transport object can "hook into" your application to deliver information to your system or request information from your system.

Step 4 is about using the HLI Transport object to send commands to the JVA Perimeter Patrol.

4.3 Step 3 - Implementing PP.HLITransport.`IHLIClientTransportUser`

This section is about functions you have to implement in the object you create that inherits the PP.HLITransport.`IHLIClientTransportUser` interface.

Before we go ahead, note the following important facts that all of these functions have in common:

1. If your function throws an exception, it will stop the worker threads in the HLI Transport object and the HLI Transport object will stop working. Catch all your exceptions or make sure they do not throw exceptions.
2. If your function takes a long time to execute, it may prevent the HLI Transport object's worker threads from reading the underlying tcp/ip socket connection and result in a connection loss. Keep your functions very fast and simple.
3. Your functions are called by worker threads that are not the same as the main thread your software is using. Therefore, you will have to make sure that your functions access your data structures in a thread-safe manner and that you use "Invoke" to call methods in your UI.

GetAuthenticationKey()

When you were setting up JVA Perimeter Patrol in the “Getting Started” section of this document, you had to provide JVA Perimeter Patrol with an authentication key that your software would have to know before it would be allowed to connect to JVA Perimeter Patrol.

Normally you will store the authentication key in the configuration settings of your software project. In the example code below, I cheated a little by simply hard-coding an authentication key that matches the JVA Perimeter Patrol I used for development and testing.

The HLI Transport object will call this function every time it tries to connect, which could be as often as once every second if it is failing to connect. Make sure it is not a time-consuming function.

```
#region IHLIClientTransportUser
/// <summary>
/// The key that will be used to encrypt communications between the
/// Perimeter Patrol Server and client.
/// The client must provide the correct key or the session will not be accepted.
/// </summary>
/// <returns>The authentication key to use to encrypt messages between the
/// Perimeter Patrol Server and Client</returns>
string PP.HLITransport.IHLIClientTransportUser.GetAuthenticationKey() {
    return "abcd";
}
```

Figure 8 – Example implementation for GetAuthenticationKey()

GetIPEndPointForClientConnection()

The HLI Transport object will call this function to get the information it needs about how to connect to the JVA Perimeter Patrol software.

In the example code below, I have stored the server address as a string and use the **`System.Net.`**`Dns` object to convert it to a proper IP address object. If you are in development and need to connect to localhost, the string “localhost” will not work. Use “127.0.0.1” instead. Alternatively, you can skip the Dns conversion and use **`System.Net.`**`IPAddress`**`.Any`**. This function will be called every time the HLI Transport object attempts to connect to the JVA Perimeter Patrol server, which could be as often as once every second. Make sure it is not a time-consuming function.

```
/// <summary>
/// Called whenever making a connection to the Perimeter Patrol Server.
/// To connect to the server, the client must know the address
/// (IP Address and Port) of the server.
/// </summary>
/// <returns>An IPEndPoint object containing the address (IP address and port)
/// of the Perimeter Patrol Server</returns>
System.Net.IPEndPoint PP.HLITransport.IHLIClientTransportUser.GetIPEndPointForClientConnection() {
    try {
        string serverAddress = Properties.Settings.Default.ServerAddress;
        System.Net.IPAddress ip = System.Net.Dns.GetHostAddresses(serverAddress)[0];
        int serverPort = Properties.Settings.Default.ServerPort;
        return new System.Net.IPEndPoint(ip, serverPort);
    } catch {
        this.NotifyClient(PP.HLITransport.WarningLevel.Error,
            "Could not create an IPEndPoint for connecting to the Perimeter Patrol Server. Please chec
        return null;
    }
}
```

Figure 9 – Example implementation for GetIPEndPointForClientConnection()

OnConnection() and OnConnectionError()

The HLI Transport object will call these functions to notify your software that it has connected to or disconnected from the JVA Perimeter Patrol software.

When **OnConnection** is called, it means that you have connected to the JVA Perimeter Patrol, but you have not been authenticated yet. The JVA Perimeter Patrol will check your authentication key and will soon disconnect you if it is not correct.

When **OnConnectionError** is called, it could mean that the HLI Transport layer was unable to connect to the JVA Perimeter Patrol but it could also mean that the JVA Perimeter Patrol rejected your connection because your authentication key was incorrect.

If the HLI Transport object is disconnected from the JVA Perimeter Patrol, it will attempt to reconnect once every second. It will call the **OnConnectionError** function after every single failed connection attempt.

The example implementation of these functions simply calls functions that make the User Interface tell the user it is not connected to JVA Perimeter Patrol. Note that to do so in a thread-safe manner, the UI methods **ShowTransportLayerConnected** and **ShowTransportLayerDisconnected** must use “Invoke” to prevent cross-thread errors.

```
/// <summary>
/// This function is called whenever a connection is established with the Perimeter Patrol Server.
/// </summary>
void PP.HLITransport.IHLIClientTransportUser.OnConnection() {
    this.ShowTransportLayerConnected();
}

/// <summary>
/// Called whenever the connection to the Perimeter Patrol server is terminated.
/// The connection can be terminated by the following circumstances:
///   1. A normal ethernet connection problem.
///   2. The connection was refused by the Perimeter Patrol server due to incorrect authentication.
///   3. The Perimeter Patrol server was shut down.
///   4. Configuration settings on the Perimeter Patrol server have been changed.
///      The client will be allowed to automatically reconnect and the
///      modified configuration settings will be passed in the next Session Acceptance Message.
/// </summary>
/// <param name="exception"></param>
void PP.HLITransport.IHLIClientTransportUser.OnConnectionError(Exception exception) {
    this.ShowTransportLayerDisconnected(exception == null ? string.Empty : exception.Message);
}
```

Figure 10 - Example implementation for OnConnection() and OnConnectionError()

OnSessionAccepted()

The **OnSessionAccepted** function is called after a connection is established with the JVA Perimeter Patrol and your authentication key has been verified.

This function brings along some important information for your software system to use:

- The JVA Perimeter Patrol's configuration settings
  - As a ConfigDictionary object
- The JVA Perimeter Patrol's log file
  - As an XML-serialized string containing a database table
- The JVP Perimeter Patrol's open alarms
  - As an XML-serialized string containing a database table of the alarms.
- The JVA Perimeter Patrol's map image
  - As a byte array

Remember to handle the function quickly so that the HLI Transport object can continue reading data from the TCP/IP connection. If you have any heavy processing of the data, do it in a different thread.

Imagine the scenario where an administrator changes the configuration settings on JVA Perimeter Patrol while your software is connected via the HLI Transport. Of course, JVA Perimeter Patrol will have to notify your software system about the configuration changes. To do so, JVA Perimeter Patrol will disconnect your HLI Transport object and allow it to automatically reconnect one second later. After the reconnection is successful, JVA Perimeter Patrol will send another **OnSessionConnected** message with the new configuration settings.

You can read the **Understanding the Configuration Data Structure** section **Error! Reference source not found.** to learn more about the JVA Perimeter Patrol configuration settings.

```
/// <summary>
/// Called after a connection has been established with the Perimeter Patrol server
/// and the session has been authenticated successfully.
/// Note that whenever the Perimeter Patrol Server's configuration settings have been changed,
/// the client will be disconnected from the server.
/// This done so that when the client automatically reconnects to the server,
/// it will receive updated configuration settings.
/// </summary>
/// <param name="config">Perimeter Patrol Server configuration settings</param>
/// <param name="log">Contents of the log file as an xml string</param>
/// <param name="openLog">Open alarms as a datatable serialized into xml</param>
/// <param name="mapImage">Map Image as a byte array</param>
void PP.HLITransport.IHLIClientTransportUser.OnSessionAccepted(
    PP.HLITransport.ConfigDictionary config, string log, string openLog, byte[] mapImage)
{
    interpreter.OnSessionAcceptance(config, log, openLog, mapImage);
    this.SetMapImage(mapImage);
    this.SetConfig(config);
    this.SetLog(log);
    this.SetOpenAlarms(openLog);
}
```

Figure 11 - Example implementation for OnSessionAccepted()

```
delegate void SetMapImageDelegate(byte[] mapImage);
internal void SetMapImage(byte[] mapImage) {
    if (this.InvokeRequired) {
        this.Invoke(new SetMapImageDelegate(this.SetMapImage), mapImage);
    } else {
        pImage.BackgroundImage = new Bitmap(new MemoryStream(mapImage));
    }
}
```

Figure 12 - Displaying the Map Image

```
delegate void SetConfigDelegate(PP.HLITransport.ConfigDictionary config);
internal void SetConfig(PP.HLITransport.ConfigDictionary config)
{
    if (this.InvokeRequired)
    {
        this.Invoke(new SetConfigDelegate(this.SetConfig), config);
    }
    else
    {

        mnuControlOnSchedule.Checked = config.GetBoolean("ScheduledControlEnable");
        mnuArmAllHighPower.Enabled = !mnuControlOnSchedule.Checked;
        mnuArmAllLowPower.Enabled = !mnuControlOnSchedule.Checked;
        mnuDisarmAll.Enabled = !mnuControlOnSchedule.Checked;

        DataTable newData = new DataTable();
        newData.Columns.Add("Name");
        newData.Columns.Add("Value");
        newData.PrimaryKey = new DataColumn[] { newData.Columns[0] };
        foreach (KeyValuePair<string, PP.HLITransport.ConfigValue> kvp in config.ToDictionary())
        {
            newData.Rows.Add(kvp.Key, kvp.Value.Value.ToString());
        }

        if (dgConfig.DataSource == null)
        {
            dgConfig.DataSource = newData;
        }
        else
        {
            int save = dgConfig.FirstDisplayedScrollingRowIndex;
            ((DataTable)dgConfig.DataSource).Merge(newData);
            dgConfig.FirstDisplayedScrollingRowIndex = save;
        }
    }
}
```

Figure 13 - Displaying JVA Perimeter Patrol's configuration settings in a DataGridView. Note that the code shown also demonstrates how to merge new incoming data to a DataGridView without causing it to flicker.

OnOpenLogUpdated()

This function is called whenever there is a change to the open alarms stored in Perimeter Patrol. You will need to thoroughly understand the data that you receive in this function, because it will be important that you are able to display the alarms properly. The best way for you to learn how to interpret the alarm data is to read and understand the coding in the **Interpreter** object that is supplied with the demo project. You can see in the code sample below that we send the new open alarm information to the interpreter object.

```
/// <summary>
/// Called whenever there is a change to the open alarms.
/// </summary>
/// <param name="openLog">A data table containing open alarm information serialized as an xml string</param>
void PP.HLITransport.IHLIClientTransportUser.OnOpenLogUpdated(string openLog)
{
    interpreter.OnOpenAlarmsUpdated(openLog);
    this.SetOpenAlarms(openLog);
    this.UpdateInterpreter();
}
```

Figure 14 – Implementing the OnOpenLogUpdated() function

OnLogUpdated()

This function is called by the HLI Transport object whenever JVA Perimeter Patrol adds a new log entry or modifies an existing log entry. We keep things simple for you by sending the entire log contents every time. The log is passed in as a string containing an XML-serialized DataTable. You most likely do not need to process the log data – displaying it to the user will be sufficient for most use cases.

```
/// <summary>
/// Called whenever there is a new log entry or a log entry has been modified.
/// </summary>
/// <param name="log">The log data table serialized as an xml string</param>
void PP.HLITransport.IHLIClientTransportUser.OnLogUpdated(string log) {
    this.SetLog(log);
}
```

Figure 15 - Implementing the OnLogUpdated() function

```
delegate void SetLogDelegate(string log);
internal void SetLog(string log)
{
    if (this.InvokeRequired)
    {
        this.Invoke(new SetLogDelegate(this.SetLog), log);
    }
    else
    {
        DataSet newData = new DataSet();
        StringReader sr = new StringReader(log);
        newData.ReadXml(sr, XmlReadMode.ReadSchema);

        if (dgLog.DataSource == null)
        {
            dgLog.DataSource = newData.Tables[0];
        }
        else
        {
            int save = dgLog.FirstDisplayedScrollingRowIndex;
            ((DataTable)dgLog.DataSource).Merge(newData.Tables[0]);
            dgLog.FirstDisplayedScrollingRowIndex = save;
        }
    }
}
```

Figure 16 - Deserializing the log file and displaying it in a DataGridView. Note that the code shown also demonstrates how to merge new incoming data to a DataGridView without causing it to flicker.

OnLiveData()

The HLI Transport object will call your **OnLiveData()** function every time it receives a Live Data update from the JVA Perimeter Patrol, which is about once every second depending on your network connection speed. To keep things simple for you, the HLI Transport object passes the entire live data structure to you on every call.

Please study the **Interpreter** object in the HLI Client Demo Project to understand how you should read the live data.

```
/// <summary>
/// Called approximately once per second.
/// Contains all the live data (status values and alarms) in the system.
/// </summary>
/// <param name="liveData">A config dictionary containing all the live data - status values and alarms.</param>
void PP.HLITransport.IHLIClientTransportUser.OnLiveData(PP.HLITransport.ConfigDictionary liveData)
{
    interpreter.OnLiveData(liveData);
    this.SetLiveData(liveData);
    this.UpdateInterpreter();
}
```

Figure 17 - Implementing the OnLiveData() function. Note how the demo project sends the live data to the interpreter to have the live data placed in interpreted objects.

```
delegate void SetLiveDataDelegate(PP.HLITransport.ConfigDictionary liveData);
internal void SetLiveData(PP.HLITransport.ConfigDictionary liveData)
{
    if (this.InvokeRequired)
    {
        this.Invoke(new SetLiveDataDelegate(this.SetLiveData), liveData);
    }
    else
    {
        DataTable newData = new DataTable();
        newData.Columns.Add("Name");
        newData.Columns.Add("Value");
        newData.PrimaryKey = new DataColumn[] { newData.Columns[0] };
        foreach (KeyValuePair<string, PP.HLITransport.ConfigValue> kvp in liveData.ToDictionary())
        {
            newData.Rows.Add(kvp.Key, kvp.Value.Value.ToString());
        }

        if (dgLiveData.DataSource == null)
        {
            dgLiveData.DataSource = newData;
        }
        else
        {
            int save = dgLiveData.FirstDisplayedScrollingRowIndex;
            ((DataTable)dgLiveData.DataSource).Merge(newData);
            dgLiveData.FirstDisplayedScrollingRowIndex = save;
        }
    }
}
```

Figure 18 - Displaying LiveData in a DataGridView

NotifyClient()

This function is called whenever the JVA Perimeter Patrol sends a message that you should display to your user. Messages are usually related to authentication problems, or when the JVA Perimeter Patrol is unable to complete the action that the user requested. In the example code below, the message is displayed by a popup message box window. You may like to implement a more user-friendly way to display messages to the users of your software system.

```
/// <summary>
/// Called when there is a message that should be displayed to the user.
/// </summary>
/// <param name="warningLevel">How important is the message</param>
/// <param name="message">The message to display to the user</param>
void PP.HLITransport.IHLIClientTransportUser.NotifyClient(
                    PP.HLITransport.WarningLevel warningLevel,
                    string message) {
    this.NotifyClient(warningLevel, message);
}
```

Figure 19 - Implementing the NotifyClient() function

```
delegate void NotifyClientDelegate(PP.HLITransport.WarningLevel warningLevel, string message);
internal void NotifyClient(PP.HLITransport.WarningLevel warningLevel, string message) {
    if (this.InvokeRequired) {
        this.Invoke(new NotifyClientDelegate(this.NotifyClient), warningLevel, message);
    } else {
        MessageBox.Show(message);
    }
}
```

Figure 20 - Displaying the "NotifyClient" message to the user

4.4 Step 4 – Sending commands to Perimeter Patrol

This section contains all the commands that you can send to the JVA Perimeter Patrol.

All commands require the username and password of the currently logged-in user. This username and password must match the usernames and passwords set up in the Configuration Settings inside JVA Perimeter Patrol. They are required for our second layer of authentication security and for logging purposes as well. In this way, we have given you the ability to run your software in display-only mode with commanding disabled. If you want to, you can set up a system whereby when your software user needs to send a command, they will be prompted to enter the higher-level supervisor's username and password.

Most of the commands are asynchronous. This means that the command is sent, and the function continues without waiting for the command to be executed or any feedback about the command results.

Asynchronous functions return a boolean value. They return true if the command was successfully sent to the JVA Perimeter Patrol, or false if there was a problem sending the command. For example, an interrupted connection will result in the function returning false. If the function returns false, you can check the exception object to find out the error message. The functions will still return true if the message reached JVA Perimeter Patrol, even if JVA Perimeter Patrol was unable to carry out the action requested.

There are a couple of synchronous functions, made obvious by the **_block** suffix on the function name. These functions block execution until their results have been returned. These functions throw an exception if the operation times out or there is some circumstance preventing the result of the command from being received.

If JVA Perimeter Patrol receives the command but is unable to carry out the requested action, it will send a message to your software using the “NotifyClient” function that you have to implement for receiving these messages.

4.4.1 Special commands
ControlOnSchedule

- When True, tells JVA Perimeter Patrol to automatically arm and disarm the energiser zones according to the weekly schedule set up in the JVA Perimeter Patrol Server.
- When False, tells JVA Perimeter Patrol to wait for user commands for arming and disarming the energiser zones.

The status of this parameter is contained in the JVA Perimeter Patrol’s configuration settings. When you change it, JVA Perimeter Patrol will disconnect your software and all the other HLI Transport clients that are attached. When your HLI Transport object automatically reconnects, you will receive a new configuration settings containing the modified ControlOnSchedule parameter. Therefore your User Interface display should reflect the state of the variable contained in the configuration settings.

4.4.2 Commands that apply to all electric fence zones
JVA Perimeter Patrol cannot obey these commands if **ControlOnSchedule** is enabled.

Therefore setup your user interface to disable these commands unless the user first disabled **ControlOnSchedule**.

- ArmAllHighPower
- ArmAllLowPower
- DisarmAll

4.4.3 Commands that apply to individual zones.
These commands require a zoneId parameter. All the zones and their zoneId parameters are contained in the configuration settings that you receive in the OnSessionAccepted message after you first connect.

JVA Perimeter Patrol cannot obey these commands if “ControlOnSchedule” is switched on.

Therefore set up your user interface to disable these commands unless the user first switched “ControlOnSchedule” off.

- ArmHighPower
- ArmLowPower
- Disarm

```
bool ArmAllHighPower(string username, string password, out Exception exception);
bool ArmAllLowPower(string username, string password, out Exception exception);
bool DisarmAll(string username, string password, out Exception exception);
bool ArmHighPower(string username, string password, int zoneId, out Exception exception);
bool ArmLowPower(string username, string password, int zoneId, out Exception exception);
bool Disarm(string username, string password, int zoneId, out Exception exception);
bool ControlOnSchedule(string username, string password, bool on, out Exception exception);
```

Figure 21 - List of commands that can be sent to the JVA Perimeter Patrol

4.4.4 Other commands

ToggleOutputState

Many of the zones, including those belonging to energisers, zone monitors and IO Boards, have output relays. You can toggle the output state of the relays with this command. The **zoneIndex** parameter identifies the zone, and because each zone may have several output relays, the **outputIndex** parameter identifies the output relay.

Close Alarms

This command allows you to close resolved alarms. See the **Understanding the Alarms system** section for more information about alarms.

ClearAlarmMemory

This command allows you to close latching alarms that were generated by devices attached to JVA Perimeter Patrol Server. The “Understanding Alarms” section will give you more information about the latching alarms.

RequestArchiveLog_block

Requests an archived log for a given year and month in the past. The archived log is returned inside an `EventLogArchiveResult` object, which, amongst other helpful information, contains a string with the log DataTable serialized as an xml string.

Notice that this function is blocking.

MuteAlarms

Commands JVA Perimeter Patrol Server to stop sounding sirens for the open alarms. When JVA Perimeter Patrol Server receives this command and mutes the alarms, you will notice a couple of changes in the data that it sends you: a) The openLog containing the open alarms will all have their muted fields set to a not-null value, and b) the “HasUnmutedAlarms” value in live data will be false.

Your custom software’s siren function should activate and deactivate according to the “HasUnmutedAlarms” value in Live Data, so that it stays synchronized with the JVA Perimeter Patrol Server. In this way, any client can mute alarms and have the results duplicated not only in the server, but also on all the other clients that are connected.

4.5 Understanding the Alarms system

JVA Perimeter Patrol monitors the fence status and raises alarms whenever there is a condition on the fence or the security system that users should be aware of.

Some alarms are generated by devices such as electric energisers or monitors and sent to the JVA Perimeter Patrol. Other, more detailed alarms are generated by JVA Perimeter Patrol itself. You will need to understand the differences between these alarms when you are writing software that interfaces with the JVA Perimeter Patrol, because they have a slightly different life cycle and you will need to handle them differently.

Alarms generated by devices such as electric fence energisers and monitors

Alarms generated by devices such as electric fence energisers and monitors include those listed below. You can find out more about them in the user manuals for each device. This list is not guaranteed to be up to date as it is intended to be an example, not an authoritative reference.

- Fence-related alarms
    - Fence alarm – there is a problem with the fence
    - Ground alarm – there is a problem with the energiser's earth connection
    - Gate alarm – one or more gates have been left open (or jammed open) for too long
- Device-related alarms
    - AC Fail alarm – there is no mains power and the device is relying on its battery.
    - Low Battery alarm – the device's battery is getting low.
    - Bad Battery alarm – the device's battery is flat, too old, or has been damaged
    - Tamper alarm – the device's case has been opened by an unauthorized person
    - PCB Fault alarm – the device needs repair

JVA Perimeter Patrol reports the status of these alarms exactly as they are reported by the device generating them. Alarms generated by devices can "latch on", which means they are not closed when the condition causing the alarm has been resolved. Instead, the devices wait for a user to acknowledge the alarm before they close the alarms.

Imagine for example that a gate has been left open for too long and one of the devices has generated a Gate alarm. After the gate has been closed, the condition causing the alarm has been resolved. However, the device will hold the alarm open until it receives a **ClearAlarmMemory** command from a keypad or from JVA Perimeter Patrol. Because JVA Perimeter Patrol reports alarms exactly as they are reported by the devices, JVA Perimeter Patrol will also indicate these alarms until after they have been cleared in the devices.

Clearing the device's alarm memory is the purpose of the **ClearAlarmMemory** command that you will see in the High Level Interface documentation. Without this command, your software users will not be able to stop the devices from reporting resolved alarms.

The best way for you to know whether an open alarm was generated by a device is to use the code found in the Interpreter object for this purpose in the demo HLI client project.

Alarms generated by JVA Perimeter Patrol

In addition to reporting alarms generated by devices, the JVA Perimeter Patrol generates its own alarms too. Many of them look similar to the alarms generated by devices, but they are often more informative. For example, while a device can only generate a general fence alarm, the JVA Perimeter Patrol can create Under Voltage and Over Voltage alarms for each of three locations at which voltage is measured on each zone. This extra information helps users to diagnose the alarm causes more easily.

JVA Perimeter Patrol also generates specialized alarms such as the following:

- Coms failure – trouble communicating with the devices.
- Control Override – devices have been disarmed either by someone not using your software or JVA Perimeter Patrol. For example, someone may have used a keypad to manually disarm a device.

Every alarm must be acknowledged by users, regardless of whether the alarm condition still exists. For example, the voltage on a fence may temporarily drop below the acceptable level and an **Under Voltage** alarm will be opened. If the voltage on the fence rises back up to an acceptable level, the condition causing the alarm will no longer exist, but Perimeter Patrol will hold the alarm open until a user has acknowledged it.

Summarizing, JVA Perimeter Patrol alarms have a three-stage lifecycle:

- **Stage 1: Open**
  - Alarm condition has been detected.
  - Eg, Fence voltage has fallen below the lower alarm level.
- **Stage 2: OpenResolved**
  - Alarm condition has been resolved but no user has acknowledged it yet.
  - Eg, Fence voltage has risen back up to acceptable levels.
  - Even though the condition causing the alarm no longer exists, JVA Perimeter Patrol is holding the alarm open until a user has acknowledged that the alarm had occurred.
- **Stage 3: Closed**
  - User has acknowledged that the alarm existed and written a note in the log explaining the situation.

Sometimes there will be a situation where the alarm cannot be resolved. For example, somebody might crash a car into an electric fence, causing the fence voltage to drop below the lower alarm level. It will take a day or more for an electric fence installation contractor to come and repair the fault. In this case, the user will not be able to close the alarm, because the alarm is not resolved. Users can only close alarms that are resolved.

When this is the case, the user will want to switch off the alarm sirens, strobe lights or any other systems are being used to alert people about the alarm. JVA Perimeter Patrol allows users to **Mute** open alarms. Muting alarms will switch off the sirens, strobe lights, etc, but it will not close the alarm. You can send a Mute command to Perimeter Patrol through the HLI.

Your custom HLI Client software can also use the **CloseAlarms** command to close the alarms when a user has acknowledged them and entered a note to explain the circumstances causing the alarm.

4.6 Understanding the Configuration Data structure

JVA Perimeter Patrol sends its configuration settings to your software system when the HLI Transport object connects and is authenticated successfully. The configuration settings are a ConfigDictionary of string keys and values.

The best way for you to understand the Configuration Data fields is to look at the code of the example **Interpreter** object and see how it reads the configuration settings to create objects that can be used in your code.

An appendix lists the data contained in the configuration data.

4.7 Understanding the LiveData structure

The Live Data structure is a ConfigDictionary of string keys and string values.

Look at the code in the **Interpreter** object to see how it maps incoming live data to the objects that it created from the configuration data.

Alarms

There are a lot of named alarm values contained in this data in addition to arrays of open unresolved and open resolved alarms. Do not use these values in your custom software. They are present because they are used when two instances of Perimeter Patrol connect to each other. You don’t need them for your custom software. Examine the code used in the **Interpreter** object in the HLI Client demo project and use the same methodology it uses for reporting alarms in your own custom software.

Voltages and Arm States

Read the code in the **Interpreter** object in the HLI Client demo project to see how to get voltages and arm states of zones. Use the same methodology in your own custom software.

5 Appendix 1 – Fields in the Configuration Data

Key	Example Value	Comment
FullScreenPassword		Password to exit full screen mode. Administrators can set this parameter to stop people from exiting JVA Perimeter Patrol.
FullScreenRequiresPassword	False	Does JVA Perimeter Patrol require a password to exit full screen mode?
ScheduledControlEnable	False	Can JVA Perimeter Patrol automatically arm and disarm the zones according to the weekly control schedule? Use the "ControlOnSchedule" command to set this value from your own software.
Zones	7	Number of zones that have been configured in JVA Perimeter Patrol. Since there are seven zones, the rest of the configuration dictionary will contain values for Zones.0, Zones.1, … Zones.6
Zones.0.ArmStates	336	Number of arm states. Is always 336 because that is the number of 30-minute periods in one week. The ArmStates represent the weekly schedule that JVA Perimeter Patrol will use to automatically arm and disarm zones when the "ScheduledControlEnable" setting is True.
Zones.0.ArmStates.0	Disarm	Represents the scheduled arm state for the first 30 minutes of the week starting at 12:00am on Sunday
Zones.0.ArmStates.1	Disarm	The second 30-minute block of the week
…	…	Continues until Zones.0.ArmStates.335
Zones.0.Ch	0	Some energiser types can power more than one zone.

		If this zone belongs to one of these energisers, this parameter contains the index of the zone in the energiser.
Zones.0.ForceUseIpAddress	False	Applies only when ComMode = ETHERNET. If true, Perimeter Patrol has been configured to connect to the zone through a PAE212 ethernet adapter device using a static IP Address. If false, Perimeter Patrol is connecting to the PAE212 ethernet adapter using its Network Name. This information is helpful for you to know how to display information about the zone in your custom UI.
Zones.0.HostEnergiserType	Z14	The type of energiser powering this zone? The energiser type helps us know how many voltages will be measured, how many relay outputs the zone has etc. If this zone is an IOBoard or a zone monitor instead of an energiser zone, this value will help you determine that.
Zones.0.Inputs	0	How many inputs does this zone have? Applies to IO Boards only. If this value is non-zero, there will be extra fields representing the individual values of each input. Eg: Zones.0.Inputs.0 = 1
Zones.0.Ip	4	Number of items for the zone's IP address. Is always equal to 4. Applies only when ComMode = ETHERNET
Zones.0.Ip.0	192	
Zones.0.Ip.1	168	
Zones.0.Ip.2	0	
Zones.0.Ip.3	13	
Zones.0.KeypadId	2	The keypad id of the device

		for this zone.
Zones.0.UpperAlarmVoltage	9	Raise an alarm if voltage rises above this value while the zone is armed at high power.
Zones.0.LowerAlarmVoltage	3	Raise an alarm if voltage drops below this value while the zone is armed at high power
Zones.0.LowPowerLowerAlarmVoltage	0.5	Raise an alarm if the voltage drops below this value while the zone is armed at low power
Zones.0.LowPowerUpperAlarmVoltage	1.8	Raise an alarm if the voltage rises above this value while the zone is armed at low power
Zones.0.Name	My Zone	A user-friendly name for the zone. This value may not have been set by the administrator who configured the JVA Perimeter Patrol
Zones.0.NetworkName	ETH-161371	The network name PAE212 device that connects this zone to the Perimeter Patrol Server. Applies only when ComMode = ETHERNET.
Zones.0.NumLinePoints	0	Number of line points for this zone on the map.
Zones.0.LinePointsX	2	Number of X coords for the line points on the map
Zones.0.LinePointsX.0	100	X coord for the line point
Zones.0.LinePoints.X.1	200	X coord for the line point
Zones.0.LinePointsY	0	Number of Y coords for the line points on the map
Zones.0.LinePointsY.0	100	Y coord for the line point
Zones.0.LinePointsY.1	100	Y coord for the line point
Zones.0.Outputs	0	Number of output relays this zone has.
Zones.0.Outputs.0	True	False means the output is off (no current can flow through it)
Zones.0.RelayControlEnable	False	Indicates whether it is possible to toggle output relays for this zone.
Zones.0.Relays	5	Number of output relays this zone has. Applies only to zones, and zone monitors, not to IO Boards

Zones.0.Relays.0	False	False means the relay is off (no current can flow through it)
Zones.0.Relays.1	False	
Zones.0.Relays.2	False	
Zones.0.Relays.3	False	
Zones.0.Relays.4	False	
Zones.0.Sectors	0	
Zones.0.XPos	0	Position of the zone label on the image map. Measured in pixels from the left.
Zones.0.YPos	0	Position of the zone label on the image map. Measured in pixels from the top.