****

**Crestron Certified Drivers**

Driver Creation Guide

Crestron Electronics, Inc.

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Comments | Author |
| 1.00.00 | 10/2/2018 | First draft | Casey Blicharz |
| 1.00.01 | 1/25/2018 | Added more sections and correcting grammar mistakes | Casey Blicharz |
| 1.00.02 | 2/14/2018 | Split document and added more information | Casey Blicharz |
| 1.00.03 | 7/17/2019 | Added security system information to the document | Natalie Jackman |
| 1.00.04 | 11/5/2019 | Added Smart TV features | Casey Martineau |

Crestron and the Crestron logo are either trademarks or registered trademarks of Crestron Electronics, Inc. in the United States and/or other countries. Other trademarks, registered trademarks, and trade names may be used in this document to refer to either the entities claiming the marks and names or their products. Crestron disclaims any proprietary interest in the marks and names of others. Crestron is not responsible for errors in typography or photography.

Table of Contents

[1 Writing a driver 4](#_Toc23857961)

[1.1 Writing the main driver class 4](#_Toc23857962)

[1.2 Writing the protocol class 12](#_Toc23857963)

[1.2.1 New Feature Implementation 18](#_Toc23857964)

[1.2.2 Common Questions and Solutions 20](#_Toc23857965)

[1.3 Writing the ResponseValidation class 28](#_Toc23857966)

[1.4 Writing the transport class 30](#_Toc23857967)

# Writing a driver

The driver developer is required to implement the main class, the protocol class, and the response validation class. A custom transport must be implemented if one of the pre-built ones will not work.

## Writing the main driver class

The main driver class will need to be written by the driver developer and all drivers follow the same pattern. Developers should not override anything in this class except if authentication is required.

The first step is to create a Simpl# Library project (use a S# Pro library if the driver requires hardware access) and add the following references depending on the device type:

|  |  |
| --- | --- |
| Device Type | Required Reference |
| All | RADCommon.dll |
| Displays & projects | RADDisplay.dll |
| Cable boxes | RADCableBox.dll |
| Video servers | RADVideoServer.dll |
| AV receivers | RADAvReceiver.dll |
| Blu-ray players | RADBlurayPlayer.dll |
| Security Systems | RADSecuritySystem.dll |

\*If the driver is a COM driver then it also requires a reference to RADProTransports.dll

The next step is to create the driver class that applications will use. This class must inherit from the following based on the device type:

|  |  |
| --- | --- |
| Device Type | Required Inheritance |
| Displays & projects | ABasicVideoDisplay |
| Cable boxes | ABasicCableBox |
| Video servers | ABasicVideoServer |
| AV receivers | ABasicAVReceiver |
| Blu-ray players | ABasicBlurayPlayer |
| Security Systems | ABasicSecuritySystem |

The driver must also inherit the appropriate transport interfaces so that applications know how to initialize the driver. The following are available:

|  |  |
| --- | --- |
| Transport type | Required interface |
| TCP | ITcp |
| COM | ISerialComport & ISimpl |
| CEC | ICecDevice |

\*Crestron Connected and IR are not listed because pre-written drivers handle them. Refer to section 1.1 Libraries to see the available drivers.

Lastly, the only difference between each device type is what they initialize their protocol class with and the name of that protocol:

|  |  |  |
| --- | --- | --- |
| Device Type | Protocol Object | Initialize Parameter |
| Displays & projects | DisplayProtocol | DisplayData |
| Cable boxes | CableBoxProtocol | CableBoxData |
| Video servers | VideoServerProtocol | VideoServerData |
| AV receivers | ReceiverProtocol | AvrData |
| Blu-ray players | BlurayPlayerProtocol | BlurayPlayerData |
| Security Systems | SecuritySystemProtocol | SecurityData |

All device-types will have the same patterns used and the following pages will show examples of the various transport types.

An example of an HTTPS driver (custom transport defined in later sections)

|  |
| --- |
| public class ExampleDriverTcp : ABasicVideoDisplay, ITcp  {  public void Initialize(IPAddress ipAddress, int port)  {  // Instantiate the custom transport and pass in any current logging settings  // that may have been set before the driver was initialized  var customTransport = new ExampleDriverTransport  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  EnableRxDebug = InternalEnableRxDebug,  EnableTxDebug = InternalEnableTxDebug  };  // Must give the transport the IP address and port so it can set up its client  customTransport.SetCommunicationSettings(ipAddress);  // Set the connection transport to the custom transport  ConnectionTransport = customTransport;  // Instantiate the custom protocol and pass in any current logging settings  DisplayProtocol = new ExampleDriverProtocol(ConnectionTransport, Id)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  // Suscribe to the state change event  DisplayProtocol.StateChange += StateChange;  // Suscribe to the RxOut event  DisplayProtocol.RxOut += SendRxOut;  // Initialize the protocol with the driver's deserialized JSON data  DisplayProtocol.Initialize(DisplayData);  }  } |

An example of a TCP/IP driver that uses a pre-built transport:

|  |
| --- |
| public class ExampleDriverTcp : ABasicVideoDisplay, ITcp  {  public void Initialize(IPAddress ipAddress, int port)  {  var tcpTransport = new TcpTransport  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  EnableRxDebug = InternalEnableRxDebug,  EnableTxDebug = InternalEnableTxDebug  };  tcpTransport.Initialize(ipAddress, port);  ConnectionTransport = tcpTransport;  DisplayProtocol = new ExampleDriverProtocol(ConnectionTransport, Id)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  DisplayProtocol.StateChange += StateChange;  DisplayProtocol.RxOut += SendRxOut;  DisplayProtocol.Initialize(DisplayData);  }  } |

An example of a \*SSL TCP/IP driver that uses a pre-built transport:

|  |
| --- |
| public class ExampleDriverSslTcp : ABasicSecuritySystem, ITcp  {  public void Initialize(IPAddress ipAddress, int port)  {  Transport tcpSSLTransport = new Transport  {  EnableAutoReconnect = EnableAutoReconnect,  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  EnableRxDebug = InternalEnableRxDebug,  EnableTxDebug = InternalEnableTxDebug  };  tcpSSLTransport.Initialize(ipAddress, port);  ConnectionTransport = tcpSSLTransport;  tcpSSLTransport.DriverID = DriverID;  SecuritySystemProtocol = new ExampleDriverProtocol (ConnectionTransport, Id, \_password)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  SecuritySystemProtocol.StateChange += new StateChangeHandler(SecuritySystemProtocol\_StateChange);  SecuritySystemProtocol.RxOut += SendRxOut;  SecuritySystemProtocol.Initialize(SecurityData);  }  } |

\*The SSL TCP/IP is currently only supported by the Security System device type.

An example of a COM port driver:

|  |
| --- |
| public class ExampleDriverTcp : ABasicVideoDisplay, ISerialComport, ISimpl  {  // Initialize method for S# Pro applications  public void Initialize(IComPort comPort)  {  ConnectionTransport = new CommonSerialComport(comPort)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  EnableRxDebug = InternalEnableRxDebug,  EnableTxDebug = InternalEnableTxDebug  };  DisplayProtocol = new ExampleDriverProtocol(ConnectionTransport, Id)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  DisplayProtocol.StateChange += StateChange;  DisplayProtocol.RxOut += SendRxOut;  DisplayProtocol.Initialize(DisplayData);  }  // Initialize method for SIMPL Windows applications  public SimplTransport Initialize(Action<string, object[]> send)  {  var transport = new SimplTransport { Send = send };  ConnectionTransport = transport;  DisplayProtocol = new ExampleDriverProtocol(ConnectionTransport, Id);  DisplayProtocol.StateChange += StateChange;  DisplayProtocol.RxOut += SendRxOut;  DisplayProtocol.Initialize(DisplayData);  return transport;  }  } |

An example of a CEC driver:

|  |
| --- |
| public class ExampleDriverTcp : ABasicVideoDisplay, ICecDevice  {  // Initialize method for S# Pro applications  public void Initialize(ISerialTransport transport)  {  ConnectionTransport = transport;  DisplayProtocol = new ExampleDriverProtocol(transport)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  };  DisplayProtocol.StateChange += StateChange;  DisplayProtocol.RxOut += SendRxOut;  DisplayProtocol.Initialize(DisplayData);  }  // Initialize method for SIMPL Windows applications  public SimplTransport Initialize(int id, Action<string, object[]> send)  {  var simplTransport = new SimplTransport { Send = send };  ConnectionTransport = simplTransport;  DisplayProtocol = new ExampleDriverProtocol(simplTransport)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  DisplayProtocol.StateChange += StateChange;  DisplayProtocol.RxOut += SendRxOut;  DisplayProtocol.Initialize(DisplayData);  return simplTransport;  }  } |

An example of a TCP driver with a custom transport that supports authentication:

|  |
| --- |
| public class ExampleDriverTcp : ABasicVideoDisplay, ITcp  {  private ExampleDriverProtocol \_protocol;  private string \_usernameKey;  private string \_passwordKey;  private string \_overiddenUsername;  private string \_overiddenPassword;  public void Initialize(IPAddress ipAddress, int port)  {  var customTransport = new ExampleDriverTransport  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger,  EnableRxDebug = InternalEnableRxDebug,  EnableTxDebug = InternalEnableTxDebug  };  customTransport.SetCommunicationSettings(ipAddress);  ConnectionTransport = customTransport;  DisplayProtocol = new ExampleDriverProtocol(ConnectionTransport, Id)  {  EnableLogging = InternalEnableLogging,  CustomLogger = InternalCustomLogger  };  // Assign these values now if they were set before initialization  \_protocol.UsernameKey = \_usernameKey;  \_protocol.PasswordKey = \_passwordKey;  \_protocol.Username = \_overiddenUsername;  \_protocol.Password = \_overiddenPassword;  \_protocol.StateChange += StateChange;  \_protocol.RxOut += SendRxOut;  \_protocol.Initialize(DisplayData);  }  public override string UsernameKey  {  set  {  \_usernameKey = value;  if (\_protocol != null)  {  // Send the value to the protocol if it was changed after the driver was initialized  \_protocol.UsernameKey = value;  }  }  }  public override string PasswordKey  {  set  {  \_passwordKey = value;  if (\_protocol != null)  {  // Send the value to the protocol if it was changed after the driver was initialized  \_protocol.PasswordKey = value;  }  }  }  public override void OverrideUsername(string username)  {  \_overiddenUsername = username;  if (\_protocol != null)  {  // Send the value to the protocol if it was changed after the driver was initialized  \_protocol.Username = username;  }  }  public override void OverridePassword(string password)  {  \_overiddenPassword = password;  if (\_protocol != null)  {  // Send the value to the protocol if it was changed after the driver was initialized  \_protocol.Password = password;  }  }  } |

\*If the transport class needs the username and/or password then follow the same pattern but assign these values to the transport. The developer needs to create these fields in their custom transport or their custom protocol classes.

## Writing the protocol class

The driver’s protocol class is where most API-specific logic will go. The following is a sample not based on any particular API but does show some common use-cases where handling them may not be obvious:

|  |
| --- |
| public class ExampleDriverProtocol : ADisplayProtocol  {  // Authentication fields that will be set by the main class  internal string UsernameKey;  internal string PasswordKey;  internal string Username;  internal string Password;  // Used to keep track of authentication states by this driver  private bool \_authenticationComplete;  private Crestron.RAD.Common.CrestronDataStoreWrapper \_dataStore;  // A header and footer that is required for all messages sent to the device  private const string \_messageHeader = "\x00\x01\x02";  private const string \_messageFooter = "\x0D\x0A";  public ExampleDriverProtocol(ISerialTransport transport, byte id) :  base(transport, id)  {  // Assign the base ResponseValidation to the custom class  // with the parameter as "ValidatedData"  ResponseValidation = new ExampleDriverResponseValidation(ValidatedData);  // Sent PollingEnabled to false to prevent polling while authenticating with the device  PollingEnabled = false;  // Set the min/max volume levels from the device  // since the API specifies 0 = 0% and 50 = 100%  // The internal framework will handle converting the range 0-50 to 0%-100%  MinVolume = 0;  MaxVolume = 50;  // Set the polling sequence  // The device will give us power, mute, and input feedback with a PowerPoll command  // and volume is given using a different polling command  // The power off polling sequence is set to PowerPoll internally  ValidatedData.PowerOnPollingSequence =  new[] { StandardCommandsEnum.PowerPoll, StandardCommandsEnum.VolumePoll };  // Instantiate CrestronDataStore for authentication  \_dataStore = new Crestron.RAD.Common.CrestronDataStoreWrapper();  }  // This will be called when SendCommand is invoked.  // This is for adding any common formatting to commands such as headers and footers  // The command string that will appear here is what it is defined as in the JSON data  // You can set CommandPrepared to true here, but the base call will also set it to true  protected override bool PrepareStringThenSend(CommandSet commandSet)  {  if (\_authenticationComplete)  {  if (commandSet.CommandPrepared == false)  {  commandSet.Command = string.Format("{0}{1}{2}",  \_messageHeader, commandSet.Command, \_messageFooter);  }  // Always call the base  return base.PrepareStringThenSend(commandSet);  }  else  {  // Authentication is not complete so prevent any commands from being sent  // Return true to prevent the framework from calling Transport.Send on the command  return true;  }  }  // This is a command that cannot be implemented in JSON so it is defined here  public override void PlayPause()  {  // Create a CommandSet with custom data  var playPauseCommand = new CommandSet("Play-Pause", "\x08\x09",  Crestron.RAD.Common.Enums.CommonCommandGroupType.Other, null,  false, CommandPriority.Normal,  Crestron.RAD.Common.Enums.StandardCommandsEnum.PlayPause);  // Send command to the queue  SendCommand(playPauseCommand);  }  // SetVolume commands need a value embedded in the message to the device  // This can be done here  // NOTE: The JSON data's command for SetVolume is "\x65[VOL]\x03"  public override void SetVolume(uint volume)  {  // Retrieve the volume command from JSON using BuildCommand  var setVolumeCommand = BuildCommand(StandardCommandsEnum.Vol,  CommonCommandGroupType.Volume, CommandPriority.Normal);  // Replace [VOL] with the volume command  setVolumeCommand.Command = setVolumeCommand.Command.Replace(  "[VOL]", Convert.ToString(volume));  // Send the command to the queue  SendCommand(setVolumeCommand);  }  // Driver overrides this to pick out custom command group responses  protected override void ChooseDeconstructMethod(ValidatedRxData validatedData)  {  if (string.IsNullOrEmpty(validatedData.CustomCommandGroup))  {  // Must call the base if the custom command group is not set  base.ChooseDeconstructMethod(validatedData);  }  else  {  switch (validatedData.CustomCommandGroup)  {  case "RequestUsernamePassword":  // Response validation is letting the protocol know it has requested the pa  SendUsernameAndPassword();  break;  case "AuthenticationFailed":  UpdatedAuthenticatedState(false);  break;  case "AuthenticationWorked":  UpdatedAuthenticatedState(true);  break;  case "AuthenticationNotRequired":  // Authentication is not required so begin polling the device  \_authenticationComplete = true;  PollingEnabled = true;  break;  }  }  }  // This method will be called by ChooseDeconstructMethod  protected override void DeConstructVolume(string response)  {  int value = 0;  try  {  // The API will return the value as base16  value = Convert.ToInt32(response, 16);  }  catch (Exception e)  {  // Check if logging is enabled before creating a string to be logged  // The method Log will make this check as well but to avoid creating  // strings when they are not usedc heck here as well  if (EnableLogging)  {  Log(String.Format("DeConstructVolume: Failed to parse value. Reason={0}",  e.Message));  }  return;  }  base.DeConstructVolume(value.ToString(CultureInfo.InvariantCulture));  }  // This will be called when the transport indicates a change to the connection  protected override void ConnectionChanged(bool connection)  {  if (connection)  {  // The API requires a "hello" message with no header or footer when the driver  // initially connects.  // Authentication may be required depending on the response to this.  // Send the command and skip the queue  Transport.Send("Hello", null);  }  base.ConnectionChanged(connection);  }  // This will be called by the custom ResponseValidation object when it receives  // a response to PowerPoll since that response contains multiple responses  // The power response will be returned by ResponseValidation and will be  // handled internally while mute and input feedback will be handled here  // since ResponseValidation can only return one instance of ValidatedRxData  internal void HandleMuteAndInputFeedback(string muteState, string inputState)  {  DeConstructMute(muteState);  DeConstructInput(inputState);  }  private void SendUsernameAndPassword()  {  string username = string.Empty;  string password = string.Empty;  // Attempt to get the username from DataStore if the key exists  username = string.IsNullOrEmpty(UsernameKey) ?  Username : RetrieveFromDataStore(UsernameKey);  password = string.IsNullOrEmpty(PasswordKey) ?  Password : RetrieveFromDataStore(PasswordKey);  // This device API wants this in the form of "username:password\x0D\x0A"  var usernameAndPasswordCommand = string.Format("{0}:{1}{2}",  username, password, \_messageFooter);  // The driver needs to send this immediatly so skip queueing and send  // directly to the transport  Transport.Send(usernameAndPasswordCommand, null);  }  private string RetrieveFromDataStore(string key)  {  object retrievedData = default(object);  \_dataStore.GetLocalValue(key, typeof(string), out retrievedData);  return retrievedData == null ? string.Empty : retrievedData as string;  }  private void UpdatedAuthenticatedState(bool isAuthenticated)  {  \_authenticationComplete = isAuthenticated;  // Let the application know we are authenticated  FireEvent(DisplayStateObjects.Authentication, isAuthenticated);  if (\_authenticationComplete)  {  // Enable polling since the driver is now authenticated  PollingEnabled = true;  }  }  } |

### New Feature Implementation

#### Smart TV (IMediaServiceProvider, IVideoConfiguration, and IArtwork)

The Smart TV update exposes IMediaServiceProvider to applications and is supported by the driver mainly by the embedded JSON file. The overview document goes over how that file is structured.

The following methods were added that may be overridden, but if they are called from non-Smart TV methods, then the driver may fail to load properly on applications using an older driver framework.

|  |  |
| --- | --- |
| Method | Use |
| SelectMediaService(string) | This will select a media service from the embedded JSON, create a CommandSet, and will call SendCommand with it by default. |
| DeConstructActiveMediaServiceFeedback | This will attempt to match the response to a value within the embedded JSON’s definition of ActiveMediaServiceFeedback by default.  This can be overridden to handle a response from the device that indicates a media service is selected, but the name of it is unknown. The driver would list a media service with the ID “unknownMediaService” and set the response to that string before calling base.DeConstructActiveMediaServiceFeedback. |
| ArtworkOn | This will call SendCommand with a built CommandSet with the JSON specified command |
| ArtworkOff | This will call SendCommand with a built CommandSet with the JSON specified command |
| EnableArtworkOnPowerOff | This will set a flag internally that causes the PowerOff command to send ArtworkOn instead and the reverse by default. Drivers can override this if they need to handle this in a different way but must then keep track of the current state themselves. |
| DisableArtworkOnPowerOff | Disables feature above |
| SelectDisplayMode | This will call SendCommand with a built CommandSet with the JSON specified command. |

If a media service is active, then the video input should be reported as Unknown.

By default, if the JSON specifies that there will be no polling for the media service, then DeConstructInput will also call DeConstructActiveMediaServiceFeedback if the driver supports that type of feedback.

### Common Questions and Solutions

#### Ramping commands

The following ramping commands may be a little confusing because they are handled a little differently in the protocol class than in the base class. Ramping commands will specify a CommandAction parameter that is defined as:

|  |  |
| --- | --- |
| CommandAction | Use |
| None | The press and release should be sent to the device. For the majority of drivers this will just be the command itself with no release. Release is typically seen in IR drivers but the various pre-built IR drivers handle them. If the API specifies a different command for Press and Release then both must be sent. |
| Hold | The press command should be sent on a timer once every 250ms until Release or None is called. |
| Release | The release command should be sent or the hold timer should stop. |

The next page shows an example of how ArrowKey is handled internally but the same pattern is used on all other ramping commands. The methods are marked as virtual so that drivers may override them and use their own logic if needed.

**Base Class**

|  |
| --- |
| public virtual void ArrowKey(ArrowDirections direction, CommandAction action)  {  if (!SupportsArrowKeys)  {  LogCommandNotSupported("ArrowKey");  return;  }  if (DisplayProtocol.Exists())  {  switch (action)  {  case CommandAction.Hold:  DisplayProtocol.PressArrowKey(direction);  break;  case CommandAction.Release:  DisplayProtocol.ReleaseArrowKey();  break;  case CommandAction.None:  DisplayProtocol.ArrowKey(direction);  break;  }  }  else  {  LogProtocolNotInitialized("ArrowKey");  }  } |

**Protocol class**

|  |
| --- |
| public virtual void PressArrowKey(ArrowDirections direction)  {  ArrowKeyRampingDirection = direction;  ArrowKeyIsRamping = true;  if (ArrowKeyRampTimer == null)  {  ArrowKeyRampTimer = new CTimer(ArrowKeyTick, null, 0, \_rampingTickRate);  }  }  public virtual void ReleaseArrowKey()  {  if (ArrowKeyRampTimer == null)  {  return;  }  ArrowKeyRampTimer.TryDispose();  ArrowKeyRampTimer = null;  if (ArrowKeyIsRamping && \_arrowKeyTicks == 0)  {  ArrowKey(ArrowKeyRampingDirection);  }  \_arrowKeyTicks = 0;  ArrowKeyIsRamping = false;  }  protected void ArrowKeyTick(object obj)  {  if (!DriverLoaded)  {  return;  }  if (ArrowKeyIsRamping)  {  \_arrowKeyTicks++;  ArrowKey(ArrowKeyRampingDirection);  }  }  public virtual void ArrowKey(ArrowDirections direction)  {  CommandSet command;  switch (direction)  {  case ArrowDirections.Down:  command = BuildCommand(StandardCommandsEnum.DownArrow, CommonCommandGroupType.Arrow, CommandPriority.Normal);  break;  case ArrowDirections.Left:  command = BuildCommand(StandardCommandsEnum.LeftArrow, CommonCommandGroupType.Arrow, CommandPriority.Normal);  break;  case ArrowDirections.Up:  command = BuildCommand(StandardCommandsEnum.UpArrow, CommonCommandGroupType.Arrow, CommandPriority.Normal);  break;  case ArrowDirections.Right:  command = BuildCommand(StandardCommandsEnum.RightArrow, CommonCommandGroupType.Arrow, CommandPriority.Normal);  break;  default:  return;  }  if (command != null)  {  SendCommand(command);  }  } |

#### Volume Scaling

Certain devices will not report volume levels as 0 to 100 but instead will do different ranges.

SetVolume is responsible for converting a range of 0 to 100 to the device’s actual range and DeconstructVolume is responsible for converting from the device range to the 0 to 100. These statements are only true if the driver developer leaves MinVolume and MaxVolume at their default values of 0 and 100. These values define the device’s range but can be left at the default values if the conversions take place in SetVolume and DeconstructVolume. The example shows setting MinVolume and MaxVolume to let the conversion take place internally without a need to override.

If the device’s volume range includes negative numbers then it is required to override SetVolume and DeconstructVolume to convert the values.

#### Multiple feedback values in one response

Some APIs will return multiple states in one response, such as the example above which shows PowerPoll returning a power state, mute state, and the current input state.

ResponseValidation.ValidatedData should return an instance of ValidatedRxData that matches the command group of the polling command. Any extra states must be routed to their appropriate Deconstruct<Feature> methods by allowing the ResponseValidation class to have a reference to the protocol.

#### Handling authentication between the protocol and ResponseValidation

There are two ways to handle authentication-related responses that request the username and/or password, acknowledgements of authenticating with the device or acknowledging no need for authentication.

The first way is to use CustomCommandGroup on the ValidatedRxData object and set those to values that can be handled by the protocol’s ChooseDeconstructMethod method.

The second way is to add internal or public methods in the protocol and to give ResponseValidation your protocol that it can then call the custom methods on demand.

#### Devices that have no network connection when powered off

Some devices will shut off their LAN port when they are in an off state and only turn on via Wake on LAN. These devices require some custom code in the driver.

The first thing that must be handled is the power state. The driver’s protocol class should override ConnectionChanged, call the base, and if the parameter was false, it should call DeconstructPower with the OFF value.

The second thing that must be handled would be a custom PowerOn command that will trigger a Wake on LAN packet. It is easier to set the command string to something unique like “CUSTOMPOWERON” and then have the SendMethod within the transport handle sending the Wake on LAN packet.

Here is an example of a driver that handles Wake on LAN:

**Protocol**

|  |
| --- |
| public override void PowerOn()  {  if (string.IsNullOrEmpty(MacAddress))  {  LgLog("This driver requires a MAC address to power the device on via Wake On LAN. Use the user attribute ID \"MACAddress\" to set this");  }  else  {  CommandSet command = new CommandSet("PowerOn", "CUSTOMPOWERON", CommonCommandGroupType.Power, WarmUp, false, CommandPriority.Highest, StandardCommandsEnum.PowerOn);  SendCommand(command);  \_driver.Connect();  }  }  protected override void ConnectionChanged(bool connection)  {  base.ConnectionChanged(connection);  if (connection == false)  {  // "off" is the string used in the driver's JSON data for powered off feedback  DeConstructPower("off");  }  } |

**Transport**

|  |
| --- |
| public override void SendMethod(string message, object[] paramaters)  {  if (message.Equals("CUSTOMPOWERON"))  {  // Must use Wake On LAN for this part.  using (UDPServer server = new UDPServer("255.255.255.255", 4343, 102))  {  Log("Sending WakeOnLAN packet to LG 2018 TV");  server.EnableUDPServer();  var wakeOnLanPacket = CreateWakeOnLanPacket(MacAddress);  server.SendData(wakeOnLanPacket, wakeOnLanPacket.Length);  }  }  else  {  base.SendMethod(message, paramaters);  }  } |

\*In this case, the driver extended the pre-built TcpTransport and overrode the default SendMethod.

#### Handling User Attributes

The framework has no default implementation of SetUserAttribute within the protocol class so if any user-settable attributes are defined, then they must be implemented in the driver. The main driver class will handle sending these calls to the protocol internally so they do not need to be implemented in that class.

Here is an example of how LG 2018 handles user attributes:

|  |
| --- |
| public override void SetUserAttribute(string attributeId, string attributeValue)  {  if (string.IsNullOrEmpty(attributeValue))  {  LgLog("User attribute value was null or empty");  }  else  {  switch (attributeId)  {  case "MACAddress":  if (string.IsNullOrEmpty(attributeValue))  {  LgLog("Provided MACAddress is not valid");  }  else  {  MacAddress = attributeValue;  (Transport as LgTcpTransport).MacAddress = MacAddress;  LgLog("MAC Address has been set");  }  break;  case "OnScreenID":  GenerateKey(attributeValue);  LgLog("OnScreenID has been set");  break;  }  }  } |

## Writing the ResponseValidation class

The ResponseValidation class will handle responses from the device and is responsible for parsing the data found in the response. The ValidateResponse method may not assume that all data will come in at once – it is possible to receive messages from the device byte-by-byte depending on the processor used.

Here is an example that accompanies the protocol class in the previous section:

|  |
| --- |
| public class ExampleDriverResponseValidation : ResponseValidation  {  private ExampleDriverProtocol \_protocol;  private const string \_delimeter = "\x0D\x0A";  private const string \_responseHeader = "\xFF\xFF";  public ExampleDriverResponseValidation(DataValidation dataValidation,  ExampleDriverProtocol protocol) : base(dataValidation)  {  // Assign the passed DataValidation object  DataValidation = dataValidation;  \_protocol = protocol;  }  // Override this method to handle response validation  public override ValidatedRxData ValidateResponse(string response,  CommonCommandGroupType commandGroup)  {  var validatedData = new ValidatedRxData(false, string.Empty);  // Only parse the response if it is complete  if (response.EndsWith(\_delimeter))  {  if (response.Contains("Authentication Required"))  {  validatedData.CustomCommandGroup = "RequestUsernamePassword";  validatedData.Ready = true;  }  else if (response.Contains("Authentication Not Required"))  {  validatedData.CustomCommandGroup = "AuthenticationNotRequired";  validatedData.Ready = true;  }  else if (response.Contains("Authentication Complete"))  {  validatedData.CustomCommandGroup = "AuthenticationWorked";  validatedData.Ready = true;  }  else  {  // This is a normal response with the format [header][group][data][delimiter]  // The end goal is to get validatedData.Data to only be [data]  // and validatedData.CommandGroup to be [group]  // Remove header and delimiter  response = response.Replace(\_responseHeader, string.Empty);  response = response.Replace(\_delimeter, string.Empty);  // Find group  if (response.StartsWith(DataValidation.PowerFeedback.GroupHeader))  {  validatedData.CommandGroup = CommonCommandGroupType.Power;  // Remove group  response = response.Replace(  DataValidation.PowerFeedback.GroupHeader,  string.Empty);  // [data] will be [powerState]:[muteState]:[inputState]  var splitResponse = response.Split(':');  string muteState = string.Empty;  string inputState = string.Empty;  for (int i = 0; i < splitResponse.Length; i++)  {  switch (i)  {  case 0:  validatedData.Data = splitResponse[0];  validatedData.Ready = true;  break;  case 1:  muteState = splitResponse[1];  break;  case 2:  inputState = splitResponse[2];  break;  }  }  \_protocol.HandleMuteAndInputFeedback(muteState, inputState);  }  else if (response.StartsWith(DataValidation.VolumeFeedback.GroupHeader))  {  validatedData.CommandGroup = CommonCommandGroupType.Volume;  // Remove group  response = response.Replace(  DataValidation.VolumeFeedback.GroupHeader,  string.Empty);  validatedData.Data = response;  validatedData.Ready = true;  }  }  }  return validatedData;  }  } |

## Writing the transport class

Writing a custom transport class is only required if the driver uses a transport that is not listed in this document. Transport classes must be written to be asynchronous to avoid any blocking calls that may halt driver operation.

The following is an example of how a driver would implement an HTTPS transport that requires no certificate and no authentication headers:

|  |
| --- |
| using System;  using Crestron.SimplSharp;  using Crestron.RAD.Common.Transports;  using Crestron.SimplSharp.Net.Https;  namespace ExampleDriver  {  public class ExampleDriverTransport : ATransportDriver  {  private HttpsClient \_client;  private string \_ipAddress;  private bool \_communicationPossible;  public void SetCommunicationSettings(IPAddress ipAddress)  {  \_ipAddress = ipAddress.ToString();  }  // A custom transport can use "parameters" as any type since it can only be called by this driver  public override void SendMethod(string message, object[] paramaters)  {  HttpsClientRequest request = new HttpsClientRequest();  RequestType requestType = RequestType.Post;  if (paramaters != null)  {  for (int i = 0; i < paramaters.Length; i++)  {  if (paramaters[i] is RequestType)  {  // The driver can specify the request type if it is not POST  requestType = (RequestType)paramaters[i];  }  else if (paramaters[i] is string)  {  // The driver can specify the content string  request.ContentString = (string)paramaters[i];  }  }  }  request.Url.Parse(String.Format("https://{0}/{1}", \_ipAddress, message));  request.RequestType = requestType;  request.KeepAlive = false;  // Send the data to the device  var responseCode = \_client.DispatchAsync(request, HostResponseCallback);  // The client might fail to dispatch the request so the connection state must be handled here  if (responseCode != HttpsClient.DISPATCHASYNC\_ERROR.PENDING)  {  HandleConnectionStatus(false);  }  }  // Called when Connect is called on the driver  public override void Start()  {  if (\_client == null)  {  \_client = new HttpsClient();  }  }  // Called when Disconnect is called on the driver  public override void Stop()  {  if (\_client != null)  {  \_client.Abort();  \_client.Dispose();  }  }  // ASync response from HttpsClient  public void HostResponseCallback(HttpsClientResponse response, HTTPS\_CALLBACK\_ERROR Error)  {  switch (Error)  {  case HTTPS\_CALLBACK\_ERROR.COMPLETED:  HandleConnectionStatus(true);  break;  case HTTPS\_CALLBACK\_ERROR.INVALID\_PARAM:  case HTTPS\_CALLBACK\_ERROR.UNKNOWN\_ERROR:  HandleConnectionStatus(false);  break;  }  if (response != null &&  DataHandler != null)  {  // Send the received data to the protocol class  DataHandler(response.ContentString);  }  }  // Sends the connection event to the main driver class only if there was a change to the state  private void HandleConnectionStatus(bool isConnected)  {  if (isConnected)  {  if (!\_communicationPossible)  {  \_communicationPossible = true;  ConnectionChanged(\_communicationPossible);  }  }  else  {  if (\_communicationPossible)  {  \_communicationPossible = false;  ConnectionChanged(\_communicationPossible);  }  }  }  }  } |