Welcome to Connection Handler’s Documentation

This module allows the programmer to easily manage properties and implement automatic reconnection after the disconnect of an Ethernet or serial interface in a Global Scripter program.

The module consists of 6 classes: ConnectionHandler, RawSimplePipeHandler, ModuleSimplePipeHandler, RawTcpHandler, ModuleTcpHandler and ServerExHandler.

These classes do not need to be instantiated directly. Instead, the module implements the GetConnectionHandler function which inspects an interface instance to return the correct type of connection handler.

The returned instance implements the same API as the interface instance it was created from and adds features to it.

This module may be used with both ControlScript and ControlScript xi.

Minimum Software and Firmware Requirements

IP Link Pro xi Series

Product	Firmware	ControlScript Pro xi
IPCP Pro xi Firmware	1.11.0000-b006	1.8.21
IPCP Pro Q xi Firmware	1.11.0000-b006	1.8.21
TouchLink Pro 1025/725/525 with LinkLicense	3.10.0002-b002	1.5.4
TouchLink Pro 300M with LinkLicense	1.05.0000-b002	1.5.4

IP Link Pro Series

Product	Firmware	ControlScript Pro
IPL Pro Firmware	3.17.0002-b002	3.13.18

Software

Software Product	Version
Global Scripter	2.23.0
ControlScript Deployment Utility	1.9.0

Version History

Version	Date	Description
2.3.0	2/21/2022	Added DanteInterface support. Reorganized Interface table in GetConnectionHandler notes.
2.2.0	12/14/2021	Added SPInterface support.
2.1.0	2/9/2021	Added support for HTTP and SummitConnect modules.
2.0.1	2/14/2020	Fixed `RawTcpHandler` and `ModuleTcpHandler` ConnectedAlready scenario. Added new examples (Pause and Resume Polling and Setting Interface Properties). Misc. documentation typos.
2.0.0	2/7/2019	Updated.
1.0.0	1/1/2018	Initial version.

ModuleVersion Function

`ModuleVersion`()

The ConnectionHandler Module version.

Return type: string

GetConnectionHandler Function

`GetConnectionHandler`(*Interface*, *keepAliveQuery=None*, *keepAliveQueryQualifier=None*, *DisconnectLimit=15*, *pollFrequency=1*, *connectRetryTime=5*, *serverTimeout=300*)

Creates a new connection handler instance tailored to the object instance passed in the Interface argument.

```
Server = GetConnectionHandler(EthernetServerInterfaceEx(9000),
                              serverTimeout=30)

# The connection handler will use the Global Scripter Module's Video
# Mute query function to keep the connection alive.
Dvs605 = GetConnectionHandler(DvsModule.EthernetClass('192.168.1.1',
                              9000, Model='DVS 605'), 'VideoMute',
                              DisconnectLimit=5)

# MlcKeepAlive in this example is a function that sends a programmer-
# specified query string.
Mlc206 = GetConnectionHandler(SerialInterface(MainProcessor, 'COM2'),
                              MlcKeepAlive)
```

Parameters:

- **Interface** (*One of the interface types in the table below or a derivative of one of these types.*) – The extronlib.interface for which to create a connection handler.
- **keepAliveQuery** (*string*, *callable*) – A function name as a string or reference to a callable used to execute the keep alive query.
- **keepAliveQueryQualifier** (*dict*) – When Interface is a Global Scripter Module, any qualifiers needed by the update function.

- **DisconnectLimit** (*int*) – Maximum number of missed responses that indicates a disconnected device. Defaults to 15.
- **pollFrequency** (*float*) – How often to send the keep alive query in seconds. Defaults to 1.
- **connectRetryTime** (*float*) – Number of seconds to wait before attempting to reconnect after disconnect.
- **serverTimeout** (*float.*) – For server Interfaces, maximum time in seconds to allow before disconnecing idle clients. Defaults to 5 minutes.

Returns: An object instance with an API similar to an extronlib.interface object.

Raises:

- **TypeError** – if Interface is an *EthernetServerInferface* (non-Ex) or is a UDP *EthernetServerInferfaceEx*.
- **ValueError** – if Interface is an *EthernetClientInterface* and its Protocol type is not TCP, UDP, or SSH.

The returned object instance depends on the instance type passed in Interface.

Interface	Returned Instance
DanteInterface	`RawTcpHandler`
EthernetClientInterface - SSH	`RawTcpHandler`
EthernetClientInterface - TCP	`RawTcpHandler`
EthernetClientInterface - UDP	`RawSimplePipeHandler`
EthernetServerInterfaceEx	`ServerExHandler`
Global Scripter Module - Dante	`ModuleTcpHandler`
Global Scripter Module - HTTP	`ModuleSimplePipeHandler`
Global Scripter Module - Serial	`ModuleSimplePipeHandler`
Global Scripter Module - Serial-Over-Ethernet	`ModuleTcpHandler`
Global Scripter Module - SPInterface	`ModuleSimplePipeHandler`
Global Scripter Module - SSH	`ModuleTcpHandler`
Global Scripter Module - TCP	`ModuleTcpHandler`
Global Scripter Module - UDP	`ModuleSimplePipeHandler`
SerialInterface	`RawSimplePipeHandler`
SPInterface	`RawSimplePipeHandler`

Note:

- Only TCP EthernetServerInterfaceEx instances are supported.
- DanteInterface does not have a *SendAndWait* method.
- DanteInterface is only available in ControlScript Pro xi.

ConnectionHandler Class

class **`ConnectionHandler`**(*Interface*, *pollFrequency*)

Base class for all client-type connection handlers.

This class is not intended to be used directly as it is the base class for all connection handlers. Rather, use `GetConnectionHandler()` to instantiate the correct handler for your interface type.

`Connected`

`Event:` Triggers when the underlying interface instance connects.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string 'Connected'.

`ConnectionStatus`

Returns the current connection state: `Connected`, `Disconnected`, or `Unknown`.

`Disconnected`

`Event:` Triggers when the underlying interface instance disconnects.

The callback function must accept two parameters. The first is the ConnectionHandlerinstance triggering the event and the second is the string 'Disconnected'.

`DisconnectLimit`

Returns:	the value set as the maximum number of missed responses that indicates a disconnected device.
Return type:	int

`Interface`

Returns:	the interface instance for which this instance is handling connection.
Return type:	an extronlib interface or a Global Scripter Module

`PollTimer`

Returns:	the timer instance used to schedule keep alive polling.
Return type:	extronlib.system.Timer

RawSimplePipeHandler Class

class **`RawSimplePipeHandler`**(*Interface*, *DisconnectLimit*, *pollFrequency*, *keepAliveQuery*)

Wraps an instance of extronlib's SerialInterface or UDP EthernetClientInterface (or derivative of these types) to provide connect and disconnect events and periodic keep alive polling.

> **Note:** Use `GetConnectionHandler()` to instantiate the correct connection handler rather than instantiating `RawSimplePipeHandler` directly.

Parameters:

- **Interface** (*SerialInterface, EthernetClientInterface, or derivative*) – The interface instance to wrap.
- **DisconnectLimit** (*int*) – Maximum number of missed responses that indicates a disconnected device.
- **pollFrequency** (*float*) – How often in seconds to send the keep alive query.
- **keepAliveQuery** (*callable*) – The callback that will send the appropriate device query. The callback must accept the calling ConnectionHandler instance as an argument.

`ReceiveData`

`Event:` Triggers when data is received by the underlying interface instance.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the received data as a bytes object.

`Connect`()

Attempts to connect and starts the polling loop. The keepAliveQuery function will be called at the rate set by the PollTimer Interval.

`ResponseAccepted`()

Resets the send counter. Call this function when a valid response has been received from the controlled device.

> **Note:** This function must be called whenever your code determines that a good keep alive response has been received.

```
@event(SomeDevice, 'ReceiveData')
def HandleReceiveData(interface, data):
    DataBuffer += data.decode()

    # Parse DataBuffer content.

    if IsGoodResponse:
        interface.ResponseAccepted()
```

`Send`(*data*)

Send data to the controlled device without waiting for response.

Parameters:	**data** (*bytes, string*) – string to send out
Raise:	TypeError, IOError

```
MainProjector.Send('GET POWER\r')
```

`SendAndWait`(*data, timeout, **delimiter*)

Send data to the controlled device and wait (blocking) for response. It returns after *timeout* seconds expires, or returns immediately if the optional condition is satisfied.

> **Note:** In addition to *data* and *timeout*, the method accepts an optional delimiter, which is used to compare against the received response. It supports any one of the following conditions:
>
> - *deliLen* (int) - length of the response
> - *deliTag* (byte) - suffix of the response
> - *deliRex* (regular expression object) - regular expression

> **Note:** The function will return an empty byte array if *timeout* expires and nothing is received, or the condition (if provided) is not met.

Parameters:

- **data** (*bytes, string*) – data to send.
- **timeout** (*float*) – amount of time to wait for response.
- **delimiter** (*see above*) – optional conditions to look for in response.

Returns: Response received data (may be empty)

Return type: bytes

`Connected`

`Event:` Triggers when the underlying interface instance connects.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string ‘Connected’.

`ConnectionStatus`

Returns the current connection state: `Connected`, `Disconnected`, or `Unknown`.

`DisconnectLimit`

Returns: the value set as the maximum number of missed responses that indicates a disconnected device.

Return type: int

`Disconnected`

`Event:` Triggers when the underlying interface instance disconnects.

The callback function must accept two parameters. The first is the ConnectionHandlerinstance triggering the event and the second is the string ‘Disconnected’.

`Interface`

Returns: the interface instance for which this instance is handling connection.

Return type: an extronlib interface or a Global Scripter Module

`PollTimer`

Returns:	the timer instance used to schedule keep alive polling.
Return type:	extronlib.system.Timer

ModuleSimplePipeHandler Class

class **`ModuleSimplePipeHandler`**(*Interface*, *DisconnectLimit*, *pollFrequency*, *keepAliveQuery*, *keepAliveQualifiers=None*)

Wraps a Global Scripter Module instance derived from extronlib's SerialInterface or UDP EthernetClientInterface to provide connect/disconnect events and periodic keep alive polling.

> **Note:** Use `GetConnectionHandler()` to instantiate the correct connection handler rather than instantiating `ModuleSimplePipeHandler` directly.

Parameters:

- **Interface** (*SerialInterface, EthernetClientInterface, or derivative*) – The interface instance who's connection state will be managed.
- **DisconnectLimit** (*int*) – Maximum number of missed responses that indicates a disconnected device.
- **pollFrequency** (*float*) – How often to send the keep alive query in seconds.
- **keepAliveQuery** (*string*) – The name of an Update function that will be queried to keep the connection alive.
- **keepAliveQualifiers** (*dict*) – Dictionary of parameter and value pairs to be passed to the keep-alive function.

`Connect`()

Attempts to connect and starts the polling loop. The keepAliveQuery function will be called at the rate set by the PollTimer Interval.

`Connected`

`Event:` Triggers when the underlying interface instance connects.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string 'Connected'.

`ConnectionStatus`

Returns the current connection state: `Connected`, `Disconnected`, or `Unknown`.

`DisconnectLimit`

Returns:	the value set as the maximum number of missed responses that indicates a disconnected device.
Return type:	int

`Disconnected`

`Event:` Triggers when the underlying interface instance disconnects.

The callback function must accept two parameters. The first is the ConnectionHandlerinstance triggering the event and the second is the string 'Disconnected'.

`Interface`

Returns:	the interface instance for which this instance is handling connection.
Return type:	an extronlib interface or a Global Scripter Module

`PollTimer`

Returns:	the timer instance used to schedule keep alive polling.
Return type:	extronlib.system.Timer

RawTcpHandler Class

class **`RawTcpHandler`**(*Interface*, *DisconnectLimit*, *pollFrequency*, *keepAliveQuery*, *connectRetryTime*)

Wraps an extronlib EthernetClientInterface instance using TCP or SSH protocol to provide connect/disconnect events and periodic keep alive polling.

> **Note:** Use `GetConnectionHandler()` to instantiate the correct connection handler rather than instantiating `RawTcpHandler` directly.

Parameters:

- **Interface** (*extronlib.EthernetClientInterface*) – The interface to wrap.
- **DisconnectLimit** (*int*) – Maximum number of missed responses that indicates a disconnected device.
- **pollFrequency** (*float*) – How often to send the keep alive query in seconds.
- **keepAliveQuery** (*callable*) – The callback that will send the appropriate device query. The callback must accept the calling ConnectionHandler instance as an argument.
- **connectRetryTime** (*float*) – Time in seconds to wait before attempting to reconnect to the remote host.

`AutoReconnect`

Controls whether or not the connection handler attempts to reconnect after disconnect.

Returns:	The auto reconnect state.
Return type:	bool

`ConnectFailed`

`Event:` Triggers when a TCP connect attempt fails.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the failure reason as a string.

```
@event(SomeInterface, 'ConnectFailed')
def HandleConnectFailed(interface, reason):
    print('Connect failure:', reason)
```

ReceiveData

`Event:` Triggers when data is received by the underlying interface instance.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the received data as a bytes object.

Connect(*timeout=None*)

Attempt to connect to the server and starts the polling loop. The keepAliveQuery function will be called at the rate set by PollTimer Interval.

The connection will be attempted only once unless AutoReconnect is True.

Parameters:	**timeout** (*float*) – optional time in seconds to attempt connection before giving up.

ResponseAccepted()

Resets the send counter. Call this function when a valid response has been received from the controlled device.

> **Note:** This function must be called whenever your code determines that a good keep alive response has been received.

```
@event(SomeDevice, 'ReceiveData')
def HandleReceiveData(interface, data):
    global DataBuffer
    DataBuffer += data.decode()

    # Parse DataBuffer content.

    if IsGoodResponse:
        interface.ResponseAccepted()
```

Send(*data*)

Send data to the controlled device without waiting for response.

Parameters:	**data** (*bytes, string*) – string to send out
Raise:	TypeError, IOError

```
MainProjector.Send('GET POWER\r')
```

SendAndWait(*data, timeout, **delimiter*)

Send data to the controlled device and wait (blocking) for response. It returns after *timeout* seconds expires, or returns immediately if the optional condition is satisfied.

> **Note:** In addition to *data* and *timeout*, the method accepts an optional delimiter, which is used to compare against the received response. It supports any one of the following conditions:
>
> - *deliLen* (int) - length of the response
> - *deliTag* (byte) - suffix of the response
> - *deliRex* (regular expression object) - regular expression

> **Note:** The function will return an empty byte array if *timeout* expires and nothing is received, or the condition (if provided) is not met.

Parameters:	• **data** (*bytes, string*) – data to send. • **timeout** (*float*) – amount of time to wait for response. • **delimiter** (*see above*) – optional conditions to look for in response.
Returns:	Response received data (may be empty)
Return type:	bytes

`Connected`

`Event:` Triggers when the underlying interface instance connects.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string ‘Connected’.

`ConnectionStatus`

Returns the current connection state: `Connected`, `Disconnected`, or `Unknown`.

`DisconnectLimit`

Returns:	the value set as the maximum number of missed responses that indicates a disconnected device.
Return type:	int

`Disconnected`

`Event:` Triggers when the underlying interface instance disconnects.

The callback function must accept two parameters. The first is the ConnectionHandlerinstance triggering the event and the second is the string ‘Disconnected’.

`Interface`

Returns:	the interface instance for which this instance is handling connection.
Return type:	an extronlib interface or a Global Scripter Module

`PollTimer`

Returns:	the timer instance used to schedule keep alive polling.
Return type:	extronlib.system.Timer

ModuleTcpHandler Class

class **`ModuleTcpHandler`**(*Interface*, *DisconnectLimit*, *pollFrequency*, *keepAliveQuery*, *keepAliveQualifiers*, *reconnectTime*)

Wraps a Global Scripter Module instance derived from extronlib's EthernetClientInterface to provide connect/disconnect events and periodic keep alive polling.

> **Note:** Use `GetConnectionHandler()` to instantiate the correct connection handler rather than instantiating `ModuleTcpHandler` directly.

Parameters:

- **Interface** (*A Global Scripter Module instance derived from EthernetClientInterface*) – The interface instance who's connection state will be managed.
- **DisconnectLimit** (*int*) – Maximum number of missed responses that indicates a disconnected device.
- **PollTimer** (*extronlib.system.Timer*) – A Timer instance used to schedule keep alive queries.
- **keepAliveQuery** (*string*) – The name of an Update function that will be queried to keep the connection alive.
- **keepAliveQualifiers** (*dict*) – parameter and value pairs to be passed to the keep-alive function.

`AutoReconnect`

Enable or disable auto reconnect.

`ConnectFailed`

`Event:` Triggers when a TCP connect attempt fails.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the failure reason as a string.

```
@event(SomeInterface, 'ConnectFailed')
def HandleConnectFailed(interface, reason):
    print('Connect failure:', reason)
```

`Connect`(*timeout=None*)

Attempt to connect to the server and starts the polling loop. The keepAliveQuery function will be called at the rate set by PollTimer Interval.

The connection will be attempted only once unless AutoReconnect is True.

Parameters:	**timeout** (*float*) – optional time in seconds to attempt connection before giving up.

`Connected`

`Event:` Triggers when the underlying interface instance connects.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string ‘Connected’.

`ConnectionStatus`

Returns the current connection state: `Connected`, `Disconnected`, or `Unknown`.

`DisconnectLimit`

Returns:	the value set as the maximum number of missed responses that indicates a disconnected device.
Return type:	int

`Disconnected`

`Event:` Triggers when the underlying interface instance disconnects.

The callback function must accept two parameters. The first is the ConnectionHandlerinstance triggering the event and the second is the string ‘Disconnected’.

`Interface`

Returns:	the interface instance for which this instance is handling connection.
Return type:	an extronlib interface or a Global Scripter Module

`PollTimer`

Returns:	the timer instance used to schedule keep alive polling.
Return type:	extronlib.system.Timer

ServerExHandler Class

class `ServerExHandler`(*Interface*, *clientIdleTimeout*, *listenRetryTime*)

Wraps an EthernetServerInterfaceEx instance to provide automatic disconnection of idle clients.

> **Note:** Use `GetConnectionHandler()` to instantiate the correct connection handler rather than instantiating `ServerExHandler` directly.

Parameters:	• **Interface** – The interface to wrap. • **clientIdleTimeout** (*float*) – Time in seconds to allow clients to be idle

before disconnecting them.
- **listenRetryTime** (*float*) – If StartListen fails, length of time in seconds to wait before re-attempting.

`Connected`

`Event:` Triggers when a socket connection is established.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string ‘Connected’.

`Disconnected`

`Event:` Triggers when a socket connection is broken.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the string ‘Disconnected’.

`Interface`

Returns: the interface instance for which this instance is handling connection.
Return type: extronlib.interface.EthernetServerInterfaceEx

`ListenFailed`

`Event:` Triggers when StartListen fails.

The callback function must accept two parameters. The first is the ConnectionHandler instance triggering the event and the second is the failure reason as a string.

`ReceiveData`

`Event:` Triggers when data is received by the underlying interface instance.

The callback function must accept two parameters. The first is the ClientObject instance triggering the event and the second is the received data as a bytes object.

```
@event(SomeServer, 'ReceiveData')
def HandleClientData(client, data):
    ProcessClientData(data)
```

`StartListen`(*timeout=0*)

Start the listener.

Parameters: **timeout** (*float*) – length of time, in seconds, to listen for connections.

Examples

Creating a Connection Handler

Instantiate a handler for a Global Scripter Module using defaults and begin polling the keep alive query after program initialization.

```
from ConnectionHandler import GetConnectionHandler
import manf_DevName_vA_B_C_D as DevNameModule

Device = GetConnectionHandler(DevNameModule.EthernetClass('192.168.1.2',
                              9000, Model='ModelName'), 'CommandName')

def Initialize():
    # The Connect method will make the initial connection attempt and kick
    # off the keep alive polling loop.
    Device.Connect()

... The rest of the system code ...

Initialize()
```

Synchronous Devices Without a Global Scripter Module

```
 1  # First we need a function to send the keep alive query to the device and
 2  # process the response for correctness. It must accept the interface that
 3  # will be used to send the query string.
 4  def DeviceKeepAlive(interface):
 5      res = interface.SendAndWait('P', 3, deliTag=b'\r\n')
 6
 7      Command = msg[:3]
 8      State = msg[3:]
 9
10      if Command == 'Pwr' and State in ['0', '1', '2', '3']:
11          # When not using a Global Scripter Module you must manually inform
12          # the connection handler that a good response has been received.
13          interface.ResponseAccepted()
14
15  # Instantiate the connection handler referencing the function above for the
16  # keepAliveQuery parameter.
17  DeviceInterface = GetConnectionHandler(SerialInterface(MainProcessor, 'COM2'),
18                                         DeviceKeepAlive)
19
20  def Initialize():
21      DeviceInterface.Connect()
22
23  # While SerialInterface and UDP protocol EthernetClientInterface don't have
24  # Connected and Disconnected events, all ConnectionHandler instances do.
25  @event(DeviceInterface, ['Connected', 'Disconnected'])
26  def DeviceConnectHandler(interface, state):
27      print('Device is {}.'.format(state))
28
29  Initialize()
```

To change this example to a UDP Ethernet connection type, replace lines 19 and 20 above with:

```
DeviceInterface = GetConnectionHandler(EthernetClientInterface('192.168.1.1',
                                       5000, Protocol='UDP'), DeviceKeepAlive)
```

Asynchronous Devices Without a Global Scripter Module

```
 1  # First we need a function to send the keep alive query to the device. It
 2  # must accept the interface that will be used to send the query string.
 3  def DeviceKeepAlive(interface):
 4      interface.Send('P')
 5
 6  # Create a buffer to hold device response data.
 7  DeviceBuffer = ''
 8
 9  # Instantiate the connection handler referencing the function above for the
10  # keepAliveQuery parameter.
11  DeviceInterface = GetConnectionHandler(SerialInterface(MainProcessor, 'COM2'),
12                                         DeviceKeepAlive)
13
14  def Initialize():
15      DeviceInterface.Connect()
16
17  # Process device responses in a handler for SerialDevice's ReceiveData
18  # event.
19  @event(DeviceInterface, 'ReceiveData')
20  def DeviceReceiveData(interface, rcvString):
21      global DeviceBuffer
22      DeviceBuffer += rcvString.decode()
23
24      while True:
25          msg, delim, remainder = DeviceBuffer.partition('\r\n')
26          if not delim:
27              break   # '\r\n' wasn't in the buffer
28
29          DeviceBuffer = remainder  # save the leftovers
30
31          Command = msg[:3]
32          State = msg[3:]
33
34          if Command == 'Pwr' and State in ['0', '1', '2', '3']:
35              # When not using a Global Scripter Module you must manually
36              # inform the connection handler that a good response has been
37              # received.
38              interface.ResponseAccepted()
39
40  @event(DeviceInterface, ['Connected', 'Disconnected'])
41  def DeviceConnectHandler(interface, state):
42      print('Device is {}.'.format(state))
43
44  Initialize()
```

To change this example to a UDP Ethernet connection type, replace lines 11 and 12 above with:

```
DeviceInterface = GetConnectionHandler(EthernetClientInterface('192.168.1.1',
                                       5000, Protocol='UDP'), DeviceKeepAlive)
```

Global Scripter Modules

Global Scripter Modules implement the keep alive query function and required response validation logic. You need only supply the query name you wish to use.

Note: It is best practice to use a status value that you already intend to use in your program as the keep alive query.

See this section for imporant information about handling ports that can go offline.

```python
import extr_sp_DVS605_v1_1_0_1 as DvsModule

# Here we are using Video Mute as the keep alive query.
Dvs605 = GetConnectionHandler(DvsModule.EthernetClass('192.168.1.1', 23,
                              Model='DVS 605'), 'VideoMute')

def Initialize():
    Dvs605.Connect()

@event(Dvs605, ['Connected', 'Disconnected'])
def HandleDvsConnect(interface, state):
    print('DVS', state)

    if state == 'Disonnected':
        # Pause status polling on disconnect to prevent spurious Program Log
        # errors.
        FreezePollTimer.Pause()
    else:
        FreezePollTimer.Restart()

def DvsFreezeStatus(dev, state, qualifier):
    print('Freeze state:', state)

def DvsFreezeUpate(timer, count):
    # ConnectionHandler ensures that the Module's Update() function is
    # called.
    Dvs605.Update('Freeze')

def DvsVideoMuteStatus(dev, state, qualifier):
    print('Video Mute state:', state)

Dvs605.SubscribeStatus('Freeze', None, DvsFreezeStatus)
Dvs605.SubscribeStatus('VideoMute', None, DvsVideoMuteStatus)

FreezePollTimer = Timer(2, DvsFreezeUpate)
# The timer will be started in HandleDvsConnect once a connection has been
# established.
FreezePollTimer.Pause()

Initialize()
```

Server

The connection handler module extends *EthernetServerInterfaceEx* to automatically disconnect idle clients after a programmer-set time.

```python
# Instantiate a server ConnectionHandler that disconnects clients that are
# idle for 30 or more seconds.
Server = GetConnectionHandler(EthernetServerInterfaceEx(9000),
                              serverTimeout=30)

def Initialize():
    Server.StartListen()

@event(Server, 'ReceiveData')
def ServerData(client, data):
    print('Received data from', client.IPAddress, ':', data.decode())
```

```

@event(Server, ['Connected', 'Disconnected'])
def EthernetServerConnectionStatus(client, state):
    print('Server Client Activity:', client, state)
    if state == 'Connected':
        print('New connection from', client.IPAddress)
    else:
        print(client.IPAddress, 'disconnected')

Initialize()
```

Controlling Auto Reconnect

Automatic reconnection behavior can be programmatically enabled and disabled when using a TCP or SSH connection to a device.

This example shows a use case where the rack is powered down overnight.

```
def DisableReconnect(clock, dt):
    # Called by the NightlyShutdown Clock instance to stop automatic
    # reconnection attempts and polling.
    DeviceInterface.AutoReconnect = False

def EnableReconnect(clock, dt):
    # Called by the MorningPowerOn Clock instance to re-enable automatic
    # reconnection attempts and polling.
    DeviceInterface.AutoReconnect = True

    # Enabling automatic reconnection doesn't also begin making connection
    # attempts.
    DeviceInterface.Connect()

NightlyShutdown = Clock(['21:00:00'], None, DisableReconnect)
NightlyShutdown.Enable()

MorningPowerOn = Clock(['08:00:00'], None, EnableReconnect)
MorningPowerOn.Enable()
```

Pause and Resume Polling

Keep Alive Polling is scheduled using a ControlScript `Timer` instance accessible through the `PollTimer` property of all ConnectionHandler instances. Pausing this timer pauses Keep Alive Polling.

> **Note:** This **does not** affect any other status polling you created in your program.

```
def ResumeProjectorPolling():
    # Resume polling.
    DeviceInterface.PollTimer.Resume()


def HandleProjectorPower(dev, state, qualifier):
    if state == 'On':
        # handle On state
    elif state == 'Off':
        # handle Off state
    elif state == 'Cooling Down':
        # handle Cooling Down state

        # Pause polling.
        DeviceInterface.PollTimer.Pause()
        Wait(60, ResumeProjectorPolling)
    elif state == 'Warming Up':
        # handle Warming Up state

        # Pause polling.
        DeviceInterface.PollTimer.Pause()
        Wait(30, ResumeProjectorPolling)
```

Setting Interface Properties

To access properties of the an underlying interface or Global Scripter Module the programmer can use the Interface property of the GetConnectionHandler object. This can be useful for setting a Global Scripter Module device password, or other attributes. For example in the code example below the devicePassword attribute of the DVS605 Device module is set using the `Interface` property.

```
import extr_sp_DVS605_v1_1_0_1 as DvsModule

Dvs605 = GetConnectionHandler(DvsModule.EthernetClass('192.168.1.1', 23,
                              Model='DVS 605'), 'VideoMute')
Dvs605.Interface.devicePassword = 'abc123'

def Initialize():
    Dvs605.Connect()
```

TCP Extras

The connection handler module adds a new ‘ConnectFailed’ event to the `EthernetClientInterface` and Global Scripter Modules that implement the `EthernetClientInterface` when TCP or SSH protocols are used. This event is triggered and returns the reason for failure when a connection attempt fails.

```
def DeviceKeepAlive(interface):
    interface.Send('P')

DeviceBuffer = ''

EthInterface = EthernetClientInterface('192.168.1.1', 5000)

DeviceInterface = GetConnectionHandler(EthInterface, DeviceKeepAlive)

def Initialize():
    DeviceInterface.StartPolling()
```

```

@event(DeviceInterface, 'ConnectFailed')
def DeviceConnectFailed(interface, reason):
    print('Failed to connect to Device:', reason)

Initialize()
```

Online/Offline Port Host

`SerialInterface` and `SPInterface` hosts can go offline and return to an online state while the system is running for various reasons. However, a module communicating to a device through one of these hosts will continue to poll even if the host is offline, generating Program Log noise.

As a preventative measure, it is best practice to handle the host's Offline and Online events to pause polling and force the paired module into the Disconnected state as shown in the following example.

```
MainProcessor('PrimaryAlias')
ScondaryProcessor('SecondaryAlias')
SecureDevice = SPDevice('SDAlias')

MainProjector = GetConnectionHandler(
    ProjecorModule.SerialClass(SecondaryProcessor, 'COM1'),
    'Power')

ConfidenceMonitor = GetConnectionHandler(
    DisplayModule.SerialClass(SecureDevice, 'COM2'),
    'Power')

Switcher = GetConnectionHandler(
    SwitcherModule.SPIClass(SecureDevice),
    'Input')

@event(ScondaryProcessor, ['Offline', 'Online'])
HandleSecondaryOffOnline(device, state):
    if state == 'Offline':
        MainProjector.PollTimer.Pause()
        MainProjector.OnDisconnected()
    else:
        MainProjector.PollTimer.Restart()

@event(SecureDevice, ['Offline', 'Online'])
HandleSecondaryOffOnline(device, state):
    if state == 'Offline':
        ConfidenceMonitor.PollTimer.Pause()
        ConfidenceMonitor.OnDisconnected()

        Switcher.PollTimer.Pause()
        Switcher.OnDisconnected()
    else:
        ConfidenceMonitor.PollTimer.Restart()
        Switcher.PollTimer.Restart()
```