


Using Extron Device Modules with ControlScript

Extron Device Module files for Serial and Ethernet controlled devices are easy to use and implement in Global Scriptor projects. This document provides the step-by-step procedure to integrate a module into your Global Scriptor project.

Obtaining the Extron Device Module and Communication Sheet

All available Extron Device Modules are in the "Global Scriptor Modules" download section of the Extron website.

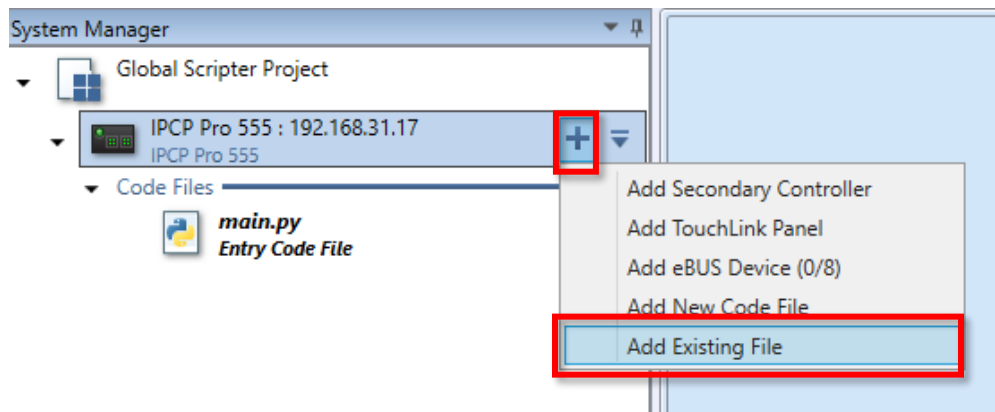
Model Number	Product Category	Version	Date Posted	Module	Communication Sheet
IN1604 HD	Scaler	1.4.1.2	Sep. 5, 2017	40 KB	 594 KB

Each module comes with a Python (.py) script file and a Communication Sheet PDF that details vital information required for using the module correctly, including:

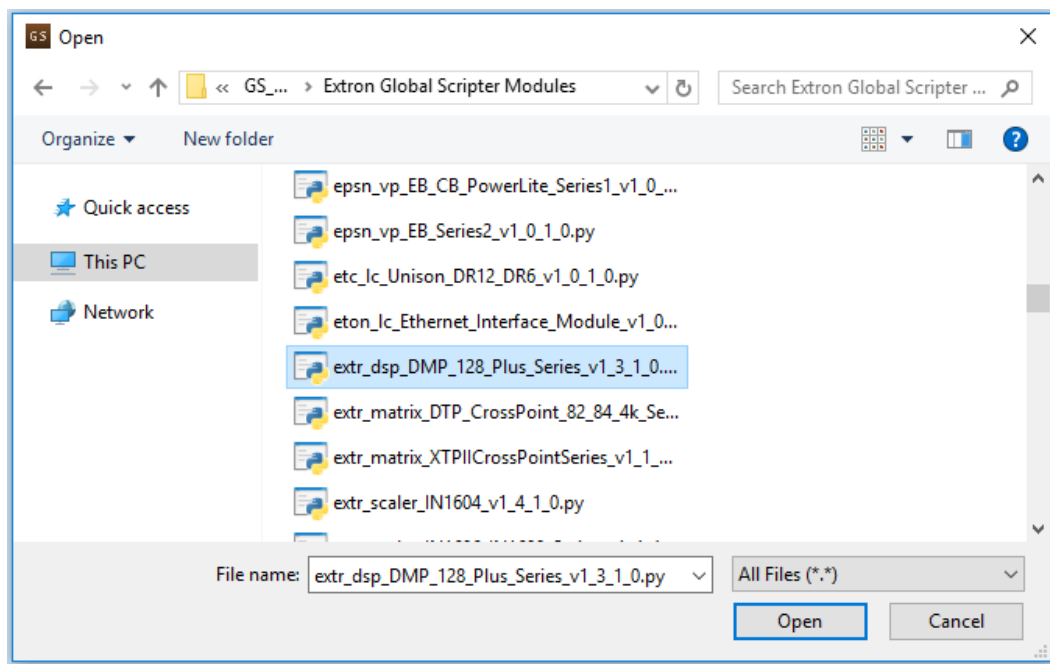
- Supported models
- Any special variables that must be set
- Supported classes
- Supported Set commands and their possible values and qualifiers
- Supported Update commands and their possible values and qualifiers
- Serial cable connection information
- Ethernet connection information
- The command strings sent out by the module
- Additional special information about the use of the module and device

Adding an Extron Device Module to a Global Scripter Project

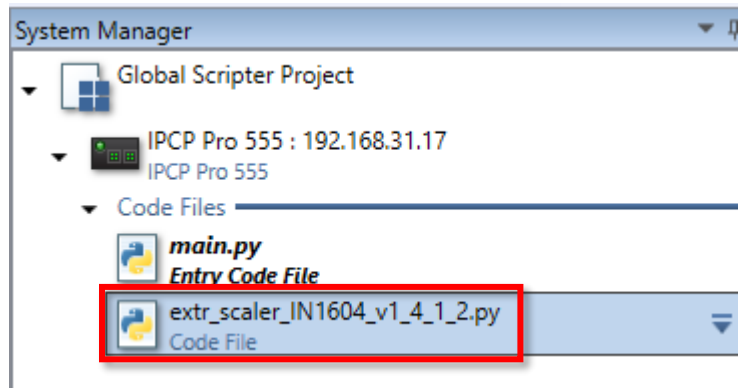
1. From the System Manager, click the Primary Controller's Add menu and click Add Existing File.



2. Select the appropriate Python (.py) file from the Open dialog box after navigating to the appropriate folder where the modules have been downloaded.



3. Click Open. This adds the Extron Module to the Global Scripter project.



Importing an Extron Device Module

To utilize an Extron Device Module in your project, access to the code file must be defined using Python's `import` keyword.

Within the comments `##Begin User Import` and `##End User Import`, import the Extron Device Module using the name of the .py file. Do not include the .py extension when importing.

Example:

```
## Begin ControlScript Import -----
from extronlib import event, Version
from extronlib.device import eBUSDevice, ProcessorDevice, UIDevice
from extronlib.interface import (ContactInterface, DigitalIOInterface,
    EthernetClientInterface, EthernetServerInterfaceEx, FlexIOInterface,
    IRInterface, RelayInterface, SerialInterface, SWPowerInterface,
    VolumeInterface)
from extronlib.ui import Button, Knob, Label, Level
from extronlib.system import Clock, MESet, Wait

print(Version())

## End ControlScript Import -----
##
## Begin User Import -----
import extr_scaler_IN1604_v1_4_1_2 as IN1604Module
## End User Import -----
##
## Begin Device/Processor Definition -----
```

The `as` keyword creates a new identifier that is bound to the imported module. In this example, the identifier `IN1604Module` can be used to reference the Extron Device Module.

Instantiating a Communication Port Using an Extron Module Class

Instantiating a Serial or Ethernet object of the Extron Module is similar to instantiating an object of the ControlScript library. The type of control communication will determine what type of object will be instantiated. These instantiations typically take place within the comments `##Begin Communication Interface Definition` and `##End Communication Interface Definition`.

Each Extron Module will have one or more classes available, depending on what communication protocol the device supports. These classes are `SerialClass`, `SerialOverEthernetClass`, `EthernetClass`, and `HTTPClass`.

See Communication Sheet for:

- Supported communication protocols
- Supported classes

Serial Communication Port

A Serial Communication port is defined using the Extron Module's `SerialClass` and setting the correct parameters. The parameters are the same as ControlScript's `SerialInterface` but with an extra parameter, `Model`, that must be set to the name of the model to be controlled. Do not create a separate `SerialInterface` for this serial port.

See Communication Sheet for:

- Valid model names for the `Model` parameter
- Valid serial settings

Example communication sheet with the valid models highlighted:

Device Specifications

Device Type:	Audio Processor
Manufacturer:	Extron
Firmware Version:	1.00
Model(s):	DMP 128 Plus, DMP 128 Plus C, DMP 128 Plus C AT, DMP 128 Plus AT

Example code:

```
## Begin Communication Interface Definition -----  
DMP128 = DMPModule.SerialClass(IPCP, 'COM1', Baud=9600, Model='DMP 128 Plus')  
## End Communication Interface Definition -----
```

Ethernet Communication Port (TCP and UDP)

An Ethernet Communication port for TCP and UDP devices is defined using the Extron Module's `EthernetClass` and setting the correct parameters. The parameters are the same as ControlScript's `EthernetClientInterface` but with an extra parameter, `Model`, that must be set to the name of the model being controlled. Do not create a separate `EthernetClientInterface` for this Ethernet port.

For TCP devices, the created object supports the same `Connect` and `Disconnect` methods found in ControlScript's `EthernetClientInterface` with the same behavior. Call `Connect` to initiate communication with the device. The `Disconnect` method is used to terminate communication with the device when necessary.

See Communication Sheet for:

- Valid model names for the `Model` parameter
- Valid ports for the `IPPort` parameter
- Valid ports for the `ServicePort` parameter for UDP devices

Example communication sheet with the valid models highlighted:

Device Specifications

Device Type: Audio Processor

Manufacturer: Extron

Firmware Version: 1.00

Model(s): DMP 128 Plus, DMP 128 Plus C, DMP 128 Plus C AT, DMP 128 Plus AT

Example code (TCP with a `Connect` call):

```
## Begin Communication Interface Definition -----
DMPEthernetPort1 = DMPModule.EthernetClass('192.168.254.249', 23, Model='DMP 128 Plus C AT')
DMPEthernetPort1.Connect(timeout=5)
## End Communication Interface Definition -----
```

Example code (UDP with a custom `ServicePort`):

```
## Begin Communication Interface Definition -----
UDPCamera = CameraModule.EthernetClass('192.168.254.241', 52381, ServicePort=52380, Model='BRBK-IP10')
## End Communication Interface Definition -----
```

Serial Over Ethernet Communication Port

An Extron Module may support the `SerialOverEthernetClass` which is used to control a serial device via Serial Over Ethernet capability. The parameters are the same as ControlScript's `EthernetClientInterface` but with an extra parameter, `Model`, that must be set to the name of the model being controlled. Do not create a separate `EthernetClientInterface` for this Ethernet port.

See Communication Sheet for:

- Valid model names for the `Model` parameter
- Valid serial settings

An example of a device that can facilitate Serial Over Ethernet communication is the Extron DTP CrossPoint 108 4K matrix switcher. The RS-232 Insertion settings in PCS must be set up correctly prior to use. The `Hostname` you specify for `SerialOverEthernetClass` would be the `Hostname` or IP Address of the Extron DTP CrossPoint 108 4K matrix switcher and the `IPPort` is the Insertion Port specified in PCS.

Example code:

```
## Begin Communication Interface Definition -----
SOEProjector = Projector.SerialOverEthernetClass('192.168.254.201', 2001, Model='SX800')
## End Communication Interface Definition -----
```

HTTP Communication

HTTP devices are defined using the Extron Module's `HTTPClass` and setting the correct parameters. The parameters are `ipAddress`, `port`, `deviceUsername`, `devicePassword`, and `Model`. All parameters are required, except for `deviceUsername` and `devicePassword` which can be left out if authentication is not required. If only a username is needed, use a blank string `' '` as the password.

See Communication Sheet for:

- Valid model names for the `Model` parameter
- Valid ports for the `port` parameter
- Default values for the `deviceUsername` and `devicePassword` parameters

Example communication sheet with the valid models highlighted:

Device Specifications

Device Type:	Video Conference
Manufacturer:	LifeSize
Firmware Version:	N/A
Model(s):	Icon 600, Icon 800

Example code with IP address of 192.168.254.250, HTTP port of 80, username of 'admin', and password of 'extron':

```
## Begin Communication Interface Definition -----  
VTC = DeviceModule.HTTPClass('192.168.254.240', 80, 'admin', 'extron', Model='Icon 600')  
## End Communication Interface Definition -----
```

Example code with username but no password:

```
## Begin Communication Interface Definition -----  
VTC = DeviceModule.HTTPClass('192.168.254.240', 80, 'admin', '', Model='Icon 600')  
## End Communication Interface Definition -----
```

Example code with no username nor password:

```
## Begin Communication Interface Definition -----  
VTC = DeviceModule.HTTPClass('192.168.254.240', 80, Model='Icon 600')  
## End Communication Interface Definition -----
```


Using the Extron Module

Overview

The module objects that are created in the [Instantiating a Communication Port Using an Extron Module Class](#) section support certain methods that enable communication with the device. Using these methods, you can control and obtain status from your device using an Extron Module. Use Python's dot notation to access these methods from the `SerialClass`, `EthernetClass`, `SerialOverEthernetClass`, or `HTTPClass` objects that were created.

Important notes:

- Polling of statuses is done by manually calling the `Update` method. Determine when and how often `Update` is called in your program.
- For TCP devices, determine when the TCP connection should be connected or disconnected and ensure that you handle any re-connection logic.
- Refer to the module's Communication Sheet to see the list of available commands for each method, along with examples.
- For more details on "command", "value", and "qualifier", see the [Method Arguments](#) section.

Setting Module Variables

Some Extron Modules require variables to be set before the module will function properly. Common examples are `deviceUsername`, `devicePassword` and `DeviceID`. Typically, you must set these variables right after instantiating your module object. The following example sets these variables for an Extron Module.

See Communication Sheet for:

- Variables that must be set
- Valid values and types of these variables

```
# Example of setting two instances of a module with different variables
cameraRoomA = VaddioModule.EthernetClass('192.168.254.249', 23, Model='ClearVIEW HD-USB PTZ')

cameraRoomA.deviceUsername = 'admin'
cameraRoomA.devicePassword = 'password_roomA'
cameraRoomA.DeviceID = '7'

cameraRoomB = VaddioModule.EthernetClass('192.168.254.248', 23, Model='ClearVIEW HD-USB PTZ')

cameraRoomB.deviceUsername = 'admin'
cameraRoomB.devicePassword = 'password_roomB'
cameraRoomB.DeviceID = '6'

# Once the variables are set, then proceed to Connect
cameraRoomA.Connect(timeout=5)
cameraRoomB.Connect(timeout=5)
```

Methods

The available methods are as follows:

- For device control: `Set`
- For device status: `Update`, `SubscribeStatus`, `ReadStatus`

Set(*command, value, qualifier=None*)

Sends a control command to the device.

Refer to the module's communication sheet to determine if a `qualifier` is necessary and for the correct `value` type.

Parameters:

command (*string*) – Name of the command

value (*string, int, float, None*) – Value to set the device to

qualifier (*dict, None*) – Parameters for the command in the form of a dictionary (if needed)

Examples

```
extronScaler1.Set('Input', '4')
extronScaler1.Set('AspectRatio', 'Fill Mode', {'Input' : '2'})
```

Update(*command, qualifier=None*)

Sends a query command to the device. This method will not return a value. Use the `ReadStatus` or `SubscribeStatus` methods to utilize the status.

Refer to the module's communication sheet to determine if a `qualifier` is necessary.

Parameters:

command (*string*) – Name of the command

qualifier (*dict, None*) – Parameters for the command in the form of a dictionary (if needed)

Examples

```
extronScaler1.Update('Input')
extronScaler1.Update('AspectRatio', {'Input' : '2'})
```

SubscribeStatus(*command, qualifier, callback*)

Specify a callback function to call when a status in the module is updated, which can possibly be triggered by responses to `Update` calls or unsolicited responses from the device.

If a `qualifier` is specified, the `callback` will be called only when the status for this particular qualifier is updated. If the `qualifier` is set to `None`, all updates to any qualifier will call the `callback`. The qualifier dictionary will be passed into the `callback` function in the same way for both cases.

Parameters:

command (*string*) – Name of the command to subscribe to

qualifier (*dict, None*) – Parameters for the command in the form of a dictionary or `None`. Unlike the other methods, `qualifier` is required here.

callback (*function*) – Function to call when the status is updated. The function must accept three parameters.

`SubscribeStatus` is used to tie a newly received status in the Extron Module to a callback function that the programmer creates. The callback function is called only when a change in the subscribed status is detected, including the very first status received.

`SubscribeStatus` does not automatically poll the device for the new status, so you must call `Update` to receive the latest status.

Some devices may provide "unsolicited" responses, in which case `Update` does not need to be continuously called for the `SubscribeStatus` to trigger the callback function. It is recommended to call `Update` once upon initialization to obtain the initial status. Refer to the module's Communication Sheet to determine if this applies for your device and command.

In general, set up all the desired `SubscribeStatus` functionality before performing any `Update` calls.

The callback function that you create must accept three arguments (refer to the module's Communication Sheet for the expected values and qualifiers and their types for the given command). The Extron Module will call your callback function and pass in certain arguments:

1. The first argument is the name of the command. The module will always provide a string.
2. The second argument is the newly received value for the status. The module will provide a string, integer, or float type.
3. The third argument is the qualifier for the command that has the new status. The module will provide either a dictionary or `None`, depending on if the command has a qualifier.

```

# Examples
mesInputButtons = MESet([btnInput1, btnInput2, btnInput3, btnInput4])

def ReceivedNewInputStatus(command, value, qualifier):

    # Set the currently selected button on an MESet of Buttons
    # based on the new value

    if value == '1':
        mesInputButtons.SetCurrent(btnInput1)
    elif value == '2':
        mesInputButtons.SetCurrent(btnInput2)
    elif value == '3':
        mesInputButtons.SetCurrent(btnInput3)
    else:
        mesInputButtons.SetCurrent(btnInput4)

extronScaler1.SubscribeStatus('Input', None, ReceivedNewInputStatus)

def ReceivedNewAspectRatioStatus(command, value, qualifier):

    # For commands that have a qualifier, the qualifier dictionary that has
    # the new status is passed into the "qualifier" argument and can be used
    # to determine the exact status that was changed

    affectedInput = qualifier['Input']

    if affectedInput == '1':
        print('New Aspect Ratio for Input 1 is', value)
    elif affectedInput == '2':
        print('New Aspect Ratio for Input 2 is', value)
    elif affectedInput == '3':
        print('New Aspect Ratio for Input 3 is', value)
    else:
        print('New Aspect Ratio for Input 4 is', value)

extronScaler1.SubscribeStatus('AspectRatio', None, ReceivedNewAspectRatioStatus)

```

ReadStatus(command, qualifier=None)

Return the latest status stored in the module for a particular command and qualifier combination.

This will not send a query to the device. Use the `Update` method to send the query.

Refer to the module's communication sheet to determine if a `qualifier` is necessary and the expected return type.

`ReadStatus` will return `None` if no status has been stored yet. For example, this would occur if `Update` has not been called yet for this particular command.

`ReadStatus` may return a status that is different from the actual status of the device. For example, this would occur if the status of the device changed after the most recent `Update`.

	command (<i>string</i>) – Name of the command
Parameters:	qualifier (<i>dict, None</i>) – Parameters for the command in the form of a dictionary (if needed)
Returns:	Value of the latest status
Return type:	string, int, float, None

```
# Examples
extronScaler1.Update('Input')
extronScaler1.Update('AspectRatio', {'Input' : '2'})

# A small time delay is introduced to allow the communication between the module
# and device to occur.
@Wait(1.5)
def PrintStatus():
    currentInputStatus = extronScaler1.ReadStatus('Input')
    print('The switcher is currently on Input', currentInputStatus)

    currentAspectRatioStatus2 = extronScaler1.ReadStatus('AspectRatio', {'Input' : '2'})
    print('The aspect ratio of Input 2 is currently set to', currentAspectRatioStatus2)
```

Method Arguments

This section details the command, value, and qualifier arguments that are used in `Set`, `Update`, `ReadStatus`, and `SubscribeStatus`.

Command

The command is the setting of the device to control. Examples include Power, Input, Video Mute, Volume, etc.

The commands that are supported in an Extron Module are listed in the **Set Commands** and **Status Available** tables in the communication sheet. Commands in the **Set Commands** table are commands that can be used to control the device and are available to use in the `Set` method. Commands in the **Status Available** table are commands that support status from the device and are available to use in the `Update`, `ReadStatus`, and `SubscribeStatus` methods (some commands may not support all three methods; see the Communication Sheet for more details).

All command names will always be strings in your code.

In this example, **Input**, **MenuNavigation**, **OnScreenDisplay**, and **Power** are some of the supported Set commands:

Command	Value	Value	Value
Input	'DTV (Antenna)'	'DTV (Cable)'	'Analog (Antenna)'
	'Analog (Cable)'	'AV 1'	'AV 2'
	'Component 1'	'Component 2'	'RGB-PC'
	'HDMI 1'	'HDMI 2'	'HDMI 3'
	'HDMI 4'		
# Input example InterfaceName.Set('Input', 'DTV (Antenna)')			
Command	Value	Value	Value
MenuNavigation	'Menu'	'Up'	'Down'
	'Left'	'Right'	'Enter'
	'Exit'	'Back'	
# MenuNavigation example InterfaceName.Set('MenuNavigation', 'Menu')			
Command	Value	Value	
OnScreenDisplay	'On'	'Off'	
# OnScreenDisplay example InterfaceName.Set('OnScreenDisplay', 'On')			
Command	Value	Value	
Power	'On'	'Off'	
# Power example InterfaceName.Set('Power', 'On')			

Value

The value is the state of the command that you would like to change to. For example, the Power command can have the values 'On' and 'Off', the Volume command can have the values 1, 2, 3, 4..., 100, etc.

Available values for a command are listed in the Extron Module's communication sheet.

Example:

Command Input	Value 'DTV (Antenna)' 'Analog (Cable)' 'Component 1' 'HDMI 1' 'HDMI 4'	Value 'DTV (Cable)' 'AV 1' 'Component 2' 'HDMI 2'	Value 'Analog (Antenna)' 'AV 2' 'RGB-PC' 'HDMI 3'
# Input example InterfaceName.Set('Input', 'DTV (Antenna)')			
Command MenuNavigation	Value 'Menu' 'Left' 'Exit'	Value 'Up' 'Right' 'Back'	Value 'Down' 'Enter'
# MenuNavigation example InterfaceName.Set('MenuNavigation', 'Menu')			
Command OnScreenDisplay	Value 'On'	Value 'Off'	
# OnScreenDisplay example InterfaceName.Set('OnScreenDisplay', 'On')			
Command Power	Value 'On'	Value 'Off'	
# Power example InterfaceName.Set('Power', 'On')			

In your code, the value is one of four Python types:

String

Strings are denoted with single quotes around them in the communication sheet.

Example:

Command	Value	Value
Power	'On'	'Off'
<pre># Power example InterfaceName.Set('Power', 'On')</pre>		

Integer

If the value is listed as a range with an integer step size, then the value must be an integer type.

Example:

Command	Value
Volume	0 to 100 in steps of 1
<pre># Volume example InterfaceName.Set('Volume', 100)</pre>	

Float

If the value is listed as a range with a decimal step size, then the value must be a float type.

Example:

Command	Value
Volume	0 to 100 in steps of 0.1
<pre># Volume example InterfaceName.Set('Volume', 0.1)</pre>	

None

If the value is listed as None, then the value must be None.

Command	Value
AutoImage	None
<pre># AutoImage example InterfaceName.Set('AutoImage', None)</pre>	

Qualifier

Qualifiers further specify the exact point of control for a given command. For example, for an Input Mute command with 'On' and 'Off' values, there may be a qualifier that specifies which exact input number to control.

All qualifiers are in the form of a [Python dictionary](#). In the dictionary, the keys are the names of the qualifiers (called "Qualifier Key"), and the values are the desired states of that qualifier (called "Qualifier Value"). The qualifier dictionary may contain several qualifier key/value pairs.

The possible Qualifier Keys and their corresponding Qualifier Values for a particular command are listed in the module's communication sheet.

Qualifier Values are one of three types: string, integer, or float (with behavior and documentation being the same as those of [Value](#)).

Example communication sheet with listed qualifiers and an example of setting the Virtual Return Mute of Channel A to On:

Command	Value	Value	
VirtualReturnMute	'On'	'Off'	
Qualifier Key	Qualifier Value	Qualifier Value	Qualifier Value
'Channel'	'A'	'B'	'C'
	'D'	'E'	'F'
	'G'	'H'	'I'
	'J'	'K'	'L'
	'M'	'N'	'O'
	'P'		
# VirtualReturnMute example InterfaceName.Set('VirtualReturnMute', 'On', {'Channel': 'A'})			

Example of using a variable to store the qualifier dictionary where the Qualifier Key is 'Channel' and the desired Qualifier Value of 'A':

```
qualifierDict = {'Channel' : 'A'}  
DMP128.Set('VirtualReturnMute', 'On', qualifierDict)
```

Example of a Set of an Extron matrix switcher's Matrix Tie Command, which uses a qualifier dictionary with multiple Qualifier Key and Qualifier Value pairs:

```
# Perform an Audio/Video tie from Input 1 to Output 3
qualifierDict = {
    'Input'    : '1',
    'Output'   : '3',
    'Tie Type': 'Audio/Video',
}

matrixDTP108.Set('MatrixTieCommand', None, qualifierDict)
```

Example of an Update using a qualifier:

```
qualifierDict = {
    'Output' : '8',
    'L/R'    : 'Left',
}

matrixDTP108.Update('DTPOutputLevel', qualifierDict)
```

Using the Communication Sheet

Available Set commands are listed in the **Set Commands** table.

Command 3DMode	Value '3D On' '2D to 3D'	Value '3D Off'	Value '3D to 2D'
Qualifier Key 'Option'	Qualifier Value 'Top and Bottom' 'Frame Sequential'	Qualifier Value 'Side by Side'	Qualifier Value 'Check Board'
Qualifier Key 'Direction'	Qualifier Value 'Right to Left'	Qualifier Value 'Left to Right'	
Qualifier Key '3D Depth'	Qualifier Value 0 to 20 in steps of 1		
# 3DMode example InterfaceName.Set('3DMode', '3D On', {'Option': 'Top and Bottom', 'Direction': 'Right to Left', '3D Depth': 20})			
Command AspectRatio	Value '4:3' 'Just Scan' 'Cinema Zoom 3' 'Cinema Zoom 6' 'Cinema Zoom 9' 'Cinema Zoom 12' 'Cinema Zoom 15'	Value '16:9' 'Cinema Zoom 1' 'Cinema Zoom 4' 'Cinema Zoom 7' 'Cinema Zoom 10' 'Cinema Zoom 13' 'Cinema Zoom 16'	Value 'Set by Program' 'Cinema Zoom 2' 'Cinema Zoom 5' 'Cinema Zoom 8' 'Cinema Zoom 11' 'Cinema Zoom 14'
# AspectRatio example InterfaceName.Set('AspectRatio', '4:3')			

Available Update commands are listed in the **Status Available** table.

Command	Value	Value	Value
AspectRatio	'4:3'	'16:9'	'Zoom'
	'Set by Program'	'Just Scan'	'Cinema Zoom 1'
	'Cinema Zoom 2'	'Cinema Zoom 3'	'Cinema Zoom 4'
	'Cinema Zoom 5'	'Cinema Zoom 6'	'Cinema Zoom 7'
	'Cinema Zoom 8'	'Cinema Zoom 9'	'Cinema Zoom 10'
	'Cinema Zoom 11'	'Cinema Zoom 12'	'Cinema Zoom 13'
	'Cinema Zoom 14'	'Cinema Zoom 15'	'Cinema Zoom 16'
<pre># AspectRatio examples InterfaceName.Update('AspectRatio') Value = InterfaceName.ReadStatus('AspectRatio') InterfaceName.SubscribeStatus('AspectRatio', None, FeedbackHandler)</pre>			
Command	Value	Value	
AudioMute	'On'	'Off'	
<pre># AudioMute examples InterfaceName.Update('AudioMute') Value = InterfaceName.ReadStatus('AudioMute') InterfaceName.SubscribeStatus('AudioMute', None, FeedbackHandler)</pre>			
Command	Value	Value	
ConnectionStatus	'Connected'	'Disconnected'	
<pre># ConnectionStatus examples Value = InterfaceName.ReadStatus('ConnectionStatus') InterfaceName.SubscribeStatus('ConnectionStatus', None, FeedbackHandler)</pre>			

The states listed in the "Value" sections are the possible values that will be passed into the `value` parameter of the callback function used in `SubscribeStatus`. They are also the possible values returned by `ReadStatus`. They are not used for the `Update` command.

In rare cases, a command in the **Status Available** table may support `Update` only, and not `ReadStatus` and `SubscribeStatus`. Similarly, a command may support `ReadStatus` and `SubscribeStatus`, but not `Update`. Please refer to the module's Communication Sheet for more information.

ConnectionStatus

- ConnectionStatus is a special status that is available in all Extron Modules with at least one other status available.
- The possible states of ConnectionStatus are 'Connected' and 'Disconnected'.
- ConnectionStatus does not support `Update`, but does support `ReadStatus` and `SubscribeStatus`.
- ConnectionStatus will become 'Connected' after the first attempted `Update`.
- ConnectionStatus will become 'Disconnected' after 15* consecutive `Update` calls that receive no response. *This number can be changed via the `connectionCounter` variable; see the module's Communication Sheet for more information.
- ConnectionStatus will become 'Connected' again after any successful `Update` that receives a valid response from the device.

Additional Examples

Power On and Off Buttons for a Display

The following is an example of creating Power On and Power Off buttons, using them in an `MESet`, and setting the buttons' `Pressed` events to call the `Set` method of the Extron Module. The `MESet` is used only for visual feedback on the UI interface and does not affect the behavior of the Extron Module.

```
## Begin User Import -----
import lg_display_xxLW_xxLZ_xxPZ_xxLV_xxLK_v1_0_0_0 as DisplayModule
## End User Import -----
##
## Begin Device/Processor Definition -----
IPCP = ProcessorDevice('IPCP550')
## End Device/Processor Definition -----
##
## Begin Device/User Interface Definition -----
TLP = UIDevice('TLP1022')
## End Device/User Interface Definition -----
##
## Begin Communication Interface Definition -----
display1 = DisplayModule.SerialClass(IPCP, 'COM1', Baud=9600, Model='47LW7700')
## End Communication Interface Definition -----

btnPowerOn = Button(TLP, 1)
btnPowerOff = Button(TLP, 2)
mesPowerButtons = MESet([btnPowerOn, btnPowerOff])
## Event Definitions -----
@event([btnPowerOn, btnPowerOff], 'Pressed')
def PowerButtonPressed(button, state):
    |
    |
    | if button is btnPowerOn:
    |     display1.Set('Power', 'On')
    |
    | elif button is btnPowerOff:
    |     display1.Set('Power', 'Off')
    |
    | mesPowerButtons.SetCurrent(button)
## End Events Definitions-----
```

Level Gauge for Volume Status

The following is an example of creating a level gauge to track the status of an Extron Module's Volume command. `SubscribeStatus` is used to set the level of the gauge each time the module receives a different status for Volume. A `Timer` is used to periodically call `Update` for Volume. See the [Polling Loop](#) section for more examples on creating polling loops.

```
lvlVolumeStatus = Level(TLP, 3)
lvlVolumeStatus.SetRange(0, 100)

def ReceivedNewVolumeStatus(command, value, qualifier):
-   lvlVolumeStatus.SetLevel(value)

display1.SubscribeStatus('Volume', None, ReceivedNewVolumeStatus)

def PollVolume(timer, count):
-   display1.Update('Volume')

volumePollTimer = Timer(3, PollVolume)
```

Multiple Instances of a Module

Since classes are used in Extron Modules, it is possible to create multiple instances of the same module and use them in your Global Scripter project. The following is an example of using the same Extron Module to handle three physical matrix switchers, which utilize two serial ports and an Ethernet port.

```
import extr_matrix_XTPIICrossPointSeries_v1_1_1_1 as moduleXTP

matrixRoomA = moduleXTP.SerialClass(IPCP, 'COM1', Baud=9600, Model='XTP II CrossPoint 1600')
matrixRoomB = moduleXTP.SerialClass(IPCP, 'COM2', Baud=9600, Model='XTP II CrossPoint 1600')

matrixMainHall = moduleXTP.EthernetClass('192.168.254.230', 23, Model='XTP II CrossPoint 6400')
matrixMainHall.devicePassword = 'extron'
matrixMainHall.Connect(timeout=5)

# Recall Preset 1 on all matrix switchers
matrixRoomA.Set('PresetRecall', '1')
matrixRoomB.Set('PresetRecall', '1')
matrixMainHall.Set('PresetRecall', '1')
```

Matrix Tie Command

All modules for Extron matrix switchers utilize a command called MatrixTieCommand to handle creating ties from inputs to outputs for both audio and video.

Command MatrixTieCommand	Value None		
Qualifier Key 'Input'	Qualifier Value '0' – '10'		
Qualifier Key 'Output'	Qualifier Value '1' – '8'	Qualifier Value 'All'	
Qualifier Key 'Tie Type'	Qualifier Value 'Audio'	Qualifier Value 'Audio/Video'	Qualifier Value 'Video'
# MatrixTieCommand example InterfaceName.Set('MatrixTieCommand', None, {'Input': '0', 'Output': '1', 'Tie Type': 'Audio'})			

The format of the command is that value is `None` while the Input, Output, and Tie Type are specified via its qualifier. An input value of '0' constitutes a break of a tie.

The following is an example of dynamically creating the qualifier dictionary during a button press and adjusting the contents of the dictionary depending on which button is pressed.

```
@event([btnLaptopToLeft, btnLaptopToRight, btnLaptopToBoth], 'Pressed')
def MakeLaptopMatrixTies(button, state):

    qualifierDict = {
        # The laptop source is on input 3 of the matrix
        'Input'      : '3',
        'Tie Type'   : 'Audio/Video',
    }

    if button is btnLaptopToLeft:

        # Adding the Output Qualifier Key to qualifierDict
        # and set its Qualifier Value to '1'
        qualifierDict['Output'] = '1'

        # This will route Input 3 to Output 1 for Audio/Video
        matrixDTP108.Set('MatrixTieCommand', None, qualifierDict)

    elif button is btnLaptopToRight:

        # Adding the Output Qualifier Key to qualifierDict
        # and set its Qualifier Value to '2'
        qualifierDict['Output'] = '2'

        # This will route Input 3 to Output 2 for Audio/Video
        matrixDTP108.Set('MatrixTieCommand', None, qualifierDict)

    elif button is btnLaptopToBoth:

        # Need to do the same exercise twice since the desired functionality
        # is the laptop source is routed to both the left and right outputs

        # This will route Input 3 to Output 1 for Audio/Video
        qualifierDict['Output'] = '1'
        matrixDTP108.Set('MatrixTieCommand', None, qualifierDict)

        # This will route Input 3 to Output 2 for Audio/Video
        qualifierDict['Output'] = '2'
        matrixDTP108.Set('MatrixTieCommand', None, qualifierDict)
```


Polling Loop

There are several different ways to incorporate a polling loop using ControlScript's `Wait` class to continuously call an Extron Module's `Update` method. The following are a few examples.

Using a single `Wait` object, a query list, and an index variable to control which query is being sent:

```
# Example of continuously calling Update to query three different statuses
# with 0.3 seconds of delay in between each query.
MATRIX_QUERY_DELAY = 0.3

MATRIX_QUERY_LIST = [
    ('ExecutiveMode', None),
    ('VideoMute', {'Output' : '1'}),
    ('VideoMute', {'Output' : '2'}),
]

queryIndex = 0

def QueryDTP108():
    global queryIndex
    command, qualifier = MATRIX_QUERY_LIST[queryIndex]
    matrixDTP108.Update(command, qualifier)

    queryIndex += 1
    if queryIndex >= len(MATRIX_QUERY_LIST):
        queryIndex = 0

matrixPollingWait.Restart()

matrixPollingWait = Wait(MATRIX_QUERY_DELAY, QueryDTP108)
```

To add more queries, append tuples of the form `(command, qualifier)` to the `MATRIX_QUERY_LIST` list. If no qualifier is necessary for a command, use `None`.

```
MATRIX_QUERY_LIST = [
    ('ExecutiveMode', None),
    ('VideoMute', {'Output' : '1'}),
    ('VideoMute', {'Output' : '2'}),
    ('InputSignalStatus', {'Input' : '1'}),
    ('Freeze', {'Output' : '5'}),
]
```

Similar to the previous example but using Python's built-in `itertools.cycle`:

```
# Example of continuously calling Update to query three different statuses
# with 0.3 seconds of delay in between each query.
import itertools

MATRIX_QUERY_DELAY = 0.3

MATRIX_QUERY_LIST = [
    ('ExecutiveMode', None),
    ('VideoMute', {'Output' : '1'}),
    ('VideoMute', {'Output' : '2'}),
]

matrixCycle = itertools.cycle(MATRIX_QUERY_LIST)

def QueryDTP108():
    command, qualifier = next(matrixCycle)
    matrixDTP108.Update(command, qualifier)
    matrixPollingWait.Restart()

matrixPollingWait = Wait(MATRIX_QUERY_DELAY, QueryDTP108)
```

Similar to the previous example but using Python's built-in `collections.deque`:

```
# Example of continuously calling Update to query three different statuses
# with 0.3 seconds of delay in between each query.
import collections

MATRIX_QUERY_DELAY = 0.3

MATRIX_QUERY_LIST = [
    ('ExecutiveMode', None),
    ('VideoMute', {'Output' : '1'}),
    ('VideoMute', {'Output' : '2'}),
]

matrixQueue = collections.deque(MATRIX_QUERY_LIST)

def QueryDTP108():
    matrixDTP108.Update(*matrixQueue[0])
    matrixQueue.rotate(-1)
    matrixPollingWait.Restart()

matrixPollingWait = Wait(MATRIX_QUERY_DELAY, QueryDTP108)
```

TCP Connection Handling

The physical TCP connection of an Extron Device Module must be handled in the code of your program. This includes the initial call of the `Connect` method to attempt to create the TCP connection as well as attempting to re-connect if the TCP connection becomes disconnected. The following example uses the module's `ConnectionStatus` as well as the `Connected` event to determine if the TCP connection is connected or disconnected.

```
matrixDTP108 = ExtronModule.EthernetClass('192.168.254.202', 23, Model='DTP Crosspoint 108 4K')

# Handling TCP connection of the Extron Matrix
def AttemptConnectMatrix():
    """Attempt to create a TCP connection to the Matrix.
    If it fails, retry in 15 seconds.
    """
    print('Attempting to connect to the Matrix')
    result = matrixDTP108.Connect(timeout=5)
    if result != 'Connected':
        reconnectWait.Restart()

reconnectWait = Wait(15, AttemptConnectMatrix)

def ReceivedMatrixConnectionStatus(command, value, qualifier):
    """If the module's ConnectionStatus becomes Disconnected, then many
    consecutive Updates have failed to receive a response from the device.
    Attempt to re-establish the TCP connection to the Matrix by calling
    Disconnect on the module instance and restarting reconnectWait.
    """
    print('Matrix module ConnectionStatus is', value)
    if value == 'Disconnected':
        matrixDTP108.Disconnect()
        reconnectWait.Restart()

matrixDTP108.SubscribeStatus('ConnectionStatus', None, ReceivedMatrixConnectionStatus)

@event(matrixDTP108, 'Connected')
def MatrixPhysicalConnectionEvent(interface, state):
    """If the TCP connection has been established physically, stop attempting
    reconnects. This can be triggered by the initial TCP connect attempt in
    the Initialize function or from the connection attempts from
    AttemptConnectMatrix.
    """
    reconnectWait.Cancel()

def Initialize():
    matrixDTP108.Connect(timeout=5)

Initialize()
```

When the 'Disconnected' status is received, you may also call `Cancel` on the `Wait` object that is handling the polling loop to cease polling. After a successful `Connect` attempt, call the `Wait` object's `Restart` to resume polling.

`Restart` is called after a successful `Connect`:

```
# Handling TCP connection of the Extron Matrix
def AttemptConnectMatrix():
    """Attempt to create a TCP connection to the Matrix.
    If it fails, retry in 15 seconds.
    """
    print('Attempting to connect to the Matrix')
    result = matrixDTP108.Connect(timeout=5)
    if result != 'Connected':
        reconnectWait.Restart()
    else:
        matrixPollingWait.Restart()
```

`Cancel` is called when the module becomes disconnected:

```
def ReceivedMatrixConnectionStatus(command, value, qualifier):
    """If the module's ConnectionStatus becomes Disconnected, then many
    consecutive Updates have failed to receive a response from the device.
    Attempt to re-establish the TCP connection to the Matrix by calling
    Disconnect on the module instance and restarting reconnectWait.
    """
    print('Matrix module ConnectionStatus is', value)
    if value == 'Disconnected':
        matrixDTP108.Disconnect()
        reconnectWait.Restart()
        matrixPollingWait.Cancel()
```

Handling BrokenPipeError Exceptions Caused by Ungraceful TCP Disconnects from Downstream Devices

In rare scenarios, calling `Set` or `Update` may produce a Python `BrokenPipeError` exception, due to the downstream device not gracefully disconnecting the TCP socket. To handle this error, a try/except block can be used.

```
try:
    dvProjector.Update('Power')
except BrokenPipeError:
    dvProjector.Disconnect()
```