ControlScript Framework Documentation

Introduction

Programming Framework

The **Programming Framework** is a collection of resources and guides designed to handle the common tasks and features of programmed systems to make development easier, more consistent, and more supportable. It answers questions on where to place common code, how to solve common problems, and lets the programmer focus on the unique parts of the customer's project. The **Programming Framework** is made up of several parts: **Code Folder Structure**, **Module Support**, **Device Modules**, and **Helper Modules**.

Code Folder Structure

The goal of the **Code Folder Structure** is to define the project code structure for maximum supportability of standard control systems.

Module Support

ModuleSupport.py is a library that provides tools used by device, helper and project modules. These are purpose-built, individually tested, minimal tools that can be used throughout the project.

Device Modules

Device Modules are collections of classes and functions that provide interfaces to devices controlled in an Extron programmed control system.

Helper Modules

Helper Modules are purpose-built modules solve common problems in standard way. They are designed with the **Programming Framework** in mind and seek to work together to help the development of programmed control systems. **Helper Modules** are individually documented.

Version History

Version	Date	Description
1.0.0	3/7/2023	Initial release

Table of Contents

Code Folder Structure details

As stated above, the goal of the **Code Folder Structure** is to define the project code structure for maximum supportability of standard control systems. It does this by laying out the **Project Source Structure** in logical partitions to maintain a cohesive Python project that guarantees the individual parts are established in a functional order and contextually relevant.

Code Folder Path is defined in the project file (e.g. 'code_folder_path': 'src').

```
<code_folder_path>/
|_ modules
| |_ device
| | |_ device_module_1.py
| | |...
| | |_ device_module_n.py
| |_ helper
| | |_ ModuleSupport.py
| | |_ helper_module_1.py
| | |_ ...
| | |_ helper_module_n.py
| |_ project
|   |_ project_module_1.py
|   |_ ...
|   |_ project_module_n.py
|_ ui
| |_ uidevice_1.py
| |...
| |_ uidevice_n.py
|_ control
| |_ concern_1.py
| |...
| |_ concern_n.py
|_ main.py
|_ variables.py
|_ devices.py
|_ system.py
|_ user_defined_1.py
|_ ...
|_ user_defined_n.py
```

Modules Folder

The *modules* folder is a place to put all of the modules used in this project. The modules folder is further broken down into subfolders for **Device Modules** (`device`), **Helper Modules** (`helper`) and **Project Modules** (`project`). The filenames in the structure above are just for indicating any number of files can be there. There is no need to rename them from how they are distributed.

device

This is the place to put all of the **Device Modules**.

> **Note:**
>
> This folder may be managed by Extron tools in the future.

helper

This is the place to put all of the **Helper Modules** (including *ModuleSupport.py*).

> #### Note:
>
> This folder may be managed by Extron tools in the future.

project

This is the place to put the project specific modules that are not Device or Helper modules. The modules here should contain classes and generic data/tools that will be used throughout the project. This is different than code organization. For breaking up code that otherwise would be in the same file, for organizational purposes, files and folders in the `<code_folder_path>` root folder should be used.

> #### Note:
>
> This folder is managed by the programmer.

UI Folder

This is the place to put the modules for each UI in the system. One module for each unique UI – mirrored panels should be in the same file.

- UI object definition
- UI navigation
- UI Emulated Feedback
- UI Live Feedback

Control Folder

This is the place to put control code for various types of systems (e.g. AV, Building Management). The core purpose is for separation of concerns. Each concern should be as isolated as possible, taking advantage of the **Code Folder Structure** and **Helper Modules**. See the various example projects for additional clarity.

Examples:

- AV devices
- Building management systems
    - Lighting
    - HVAC
- Cloud Services

Core Code Files

These files will be generated in the root source folder (`"code_folder_path"`).

main.py

The main program entrance file. The contents of this should be:

- Identification of the platform and version.
- Imports of the project components.
- A call to initialize the system.

Example main.py

```
# Extron Library Imports
from extronlib import Platform, Version

print('ControlScript', Platform(), Version())

# Project imports
import variables
import devices
import ui.tlp
import control.av
import system

system.Initialize()
```

variables.py

The variables file is for data that will be used throughout the project. This could be static or dynamic data. After being initially loaded by `main.py`, it can be imported and used in any module throughout the project.

Given the following variables.py

```
SOME_CONSTANT = 'somevalue'
someothervariable = True
someobject = {'x': 1, 'y': 2}
```

Here are example usages

```
import variables
from variables import someobject

print('Constant value', variables.SOME_CONSTANT)

if variables.someothervariable:
    # Do something
    ...
    # Toggle boolean variable
    variables.someothervariable = False
else:
    # Do something else
    ...
    # Toggle boolean variable
    variables.someothervariable = True

someobject['x'], someobject['y'] = 5, 15
print(someobject)
```

devices.py

This is the place to define each of the devices in the system.

- Extron control devices (e.g. all extronlib.device objects)
- Non-control devices and services (e.g. device modules)
- User defined devices (e.g. all extronlib.interface objects or custom python coded devices)

> ### Note:
>
> This is for definition only. Connection and logic are defined in system.py (see below).

system.py

The `system.py` is the place to define system logic, automation, services, etc. as a whole. It should provide an *Initialize* method that will be called in `main.py` to start the system after variables, devices, and UIs have been defined.

Examples of items in the system file:

- Clocks and scheduled things
- Connection of devices that need connecting
- Set up of services (e.g. ethernet servers, CLIs, etc.)
- Device control

> ### Note:
>
> In larger systems, `system.py` can be broken down further.

User Defined Python files

For larger projects it may be organizationally convenient to break down code that would otherwise belong in `system.py`. Code can be placed into files in the `src` root folder or subfolders within `src`. Please follow these guidelines:

- These files and objects therein should be imported into `system.py`.
- Avoid cross imports (i.e. import from one subfile back to another.) – data should pass through `system.py`.

Module Support

Module Support, a Python module, is a collection of tools for use throughout the **Programming Framework**.

> ## Note:
>
> See the Module Support API within the Appendix for full API documentation.

eventEx

Event handlers can be assigned using the **`eventEx`** or `event` method, typically in decorator form. The syntax for `eventEx` is the same as `event` and can be used as a drop-in replacement for event. `eventEx` allows for multiple handlers per event.

More examples of how `eventEx` can be used.

Support for all the ways *event* can be used...

```
# eventEx can be used as a drop-in replacement for event.
@eventEx(powerButton, 'Pressed')
def handlePowerButtonPressed(button, state):
    print(button.Name, state)

# eventEx supports stacking
@eventEx(button1, 'Pressed')
@eventEx(button1, 'Released')
def handleButton1(button, state):
    print(button.Name, state)

# eventEx supports lists of objects and events
@eventEx([button1, button2], ['Pressed', 'Released'])
def handleButton1(button, state):
    print(button.Name, state)
```

eventEx supports multiple handlers...

```
@eventEx(powerButton, 'Pressed')
def handlePowerButtonPressed(button, state):
    print('Will call this.', button.Name, state)

@eventEx(powerButton, 'Pressed')
def handlePowerButtonPressed-ThisToo(button, state):
    print('Will call this too.', button.Name, state)
```

> ## Note:
>
> Handlers are called in the order of assignment.

Manual Events

Manual Events provide a way to create familiar events/handlers that are both handled and called within the project. *Manual Events* are only called by code within the programmer defined code (i.e. not by internal Extron code). There are two types: **`GenericEvent`**, used to handle programmer defined events, and **`WatchVariable`**, used to handle asynchronous notifications of data changes.

Note:

See the Manual Events within the Appendix for full API documentation.

Loggers

The logger classes provide several options for collecting diagnostic information from running systems. When logging is desired across several parts of the program, it is usually sufficient to instantiate just one logger and pass/import that instance to where it is needed.

The provided classes allow diagnostics to be collected via Trace, the Program Log, and TCP connections (i.e. with DataViewer), but new logger classes can be created by the programmer. As long as new loggers implement the `Log` method using the same signature as the provided classes they can be substituted for one another without code changes.

Example Log method implementation

```
class NewLogger:
    def Log(self, *recordobjs, sep=' ', severity='info'):
        # Your implementation here. recordobjs is an iterable.
```

To reduce the need to modify code in the case where logging is no longer needed, centralize use of the logger to a single class method or function in the system. The example below outlines how to structure a class with optional logging.

Class outline with optional logging

```
class SomeClass:
    def __init__(self, ..., logger=None):
        self._logger = logger

    def SomeMethod(self):
        self._Log('log message')

    def _Log(self, *recordobjs, sep=' ', severity='info'):
        if self._logger:
            self._logger.Log(*recordobjs, sep, severity)
```

Note:

See the Loggers within the Appendix for full API documentation.

Device Module Content Overview

Module

Each Device Module contains all the content unique to the supported device(s). This includes the details about how the device can be connected, the protocol for communicating with the device and the way the program will interact with the device.

The main module features are:

- **Device Class(es)** - The device class(es) define(s) the data that will go "on the wire". This translates the user intent into the data that the device understands and vice versa. There will be at least one base device class; however, some devices may vary across models or may even support different protocols for each of its supported transports. These classes also define limited device connection and protection logic necessary to communicate with the device.
- **Transport Specific Sub-Class(es)** - The transport specific sub-class(es) subclass the device class(es) and define(s) the differences between the transports. For example, there may be subtle changes to the protocol (e.g. a prefix) or additional information (e.g. device ID) needed.

Communication Sheet

Each Device Module will come with a communication sheet that describes the module in written form, including wiring and device specific setup (if any) required.

The core document sections are:

- **Device Specification** - Describes the device(s) supported the by the module. Including type, manufacturer, firmware version(s), and model(s).
- **Supported Versions** - The software and hardware/firmware versions that are supported by the module.
- **Version History** - The release notes of the model documenting changes to each version.
- **Module Notes** - Any caveats or additional settings required to support this module.
- **Examples** - For most modules this will include the various ways to create the object. In some cases, this could include additional code required to set up the environment or anything else out of the ordinary.
- **Command Overview** - Tables of Control and Status Commands supported by the module.
- **Port Details** - Communication details such as what TCP port is used or default serial settings.
- **Command Strings** - A complete listing of all the strings sent to and from the device.

Using Device Modules in a ControlScript Project

This section will describe how to use a **Device Module** in a ControlScript project.

Device Modules

The programmer workflow for **Device Modules** is:

1. Add all modules to project.
2. Create class instances.
3. Create all subscriptions.
4. Create variables to hold data.
5. Create connection handling (possibly use our helper module).
6. Create polling logic.
7. Create data handler code.
8. Call connect method to begin communication.
9. Use the class instances throughout the code.

Adding a Device Module to a ControlScript Project

Place the Device Module into the **Device Modules** folder within the project. The standard location within the project folder is: `<code_folder_path>/modules/device/`.

Note:

If the project is not using the standard folder structure, the imports will need to be adjusted accordingly.

Importing

There are several ways to import the module into a ControlScript project. The standard way is to “import as” using a unique, meaningful name.

Within devices.py

```
# Given a module file name of extr_sp_DVS_605_Series_v1_2_5_0.py
import modules.device.extr_sp_DVS_605_Series_v1_2_5_0 as modSP
```

Note:

If the project is not using the standard folder structure, the import will need to be adjusted accordingly.

Instantiating

To instantiate a module, an interface instance will be created to provide the transport mechanism for a device instance.

Serial

```
# Extron Library imports
from extronlib.device import ProcessorDevice

# Project imports
import modules.device.extr_sp_DVS_605_Series_v1_2_5_0 as modSP
from modules.helper.ConnectionHandler import GetConnectionHandler

# Processor Device
dvIPL = ProcessorDevice('ProcessorAlias')

# Module
dvSP = GetConnectionHandler(modSP.SerialClass(dvIPL, 'COM1'), 'ExecutiveMode',
                            pollFrequency=5)
dvSP.Connect()
```

Ethernet

```
# Project imports
import modules.device.extr_sp_DVS_605_Series_v1_2_5_0 as modSP
from modules.helper.ConnectionHandler import GetConnectionHandler

# Module
dvSP = GetConnectionHandler(modSP.SSHClass('192.168.254.254', 22023),
                            'ExecutiveMode', pollFrequency=5)
dvSP.Connect()
```

Serial over Ethernet/SSH

```
# Project imports
import modules.device.extr_sp_DVS_605_Series_v1_2_5_0 as modSP
from modules.helper.ConnectionHandler import GetConnectionHandler

# Module
# Note: For Serial over SSH, add Protocol='SSH' and Credentials=('admin', 'password')
dvSP = GetConnectionHandler(modSP.SerialOverEthernetClass('192.168.254.254', 22023),
                            'ExecutiveMode', pollFrequency=5)
dvSP.Connect()
```

HTTP

```
# Project imports
import modules.device.pano_mp_Remote_recorder_v1_1_1_1 as modMP
from modules.helper.ConnectionHandler import GetConnectionHandler

# Module
dvMP = GetConnectionHandler(modMP.HTTPClass('192.168.254.254', 443, 'admin',
                                            'password',
                                            Model='Remote Recorder',
                                            SSLVerifyMode='On'),
                            'RecordingDuration', {'Recorder': 15},
                            pollFrequency=5)
dvMP.Connect()
```

Commands

Commands are actions to take with devices. They can be used to cause the device to change state or take an action (`Set`); to report the current state or status (`Update`, `SubScribeStatus`, `ReadStatus`). These actions are taken through set, update, read, subscribe and handle methods.

Set

Use the `Set` method to take action with a device.

```
  # Set the Audio and Video to input 1.
  dvSP.Set('Input', 1, {'Type': 'Audio/Video'})
```

Update

Use the `Update` method to query the device for the state of a command.

```
  # Update the Audio and Video input.
  dvSP.Update('Input', {'Type': 'Audio/Video'})
```

Read

Use the `ReadStatus` method to get that latest value.

```
  # Read the current Audio and Video input.
  avinput = dvSP.ReadStatus('Input', {'Type': 'Audio/Video'})
```

Subscribe and Handle

Use the `SubscribeStatus` method to have the module call a handler when a command status changes. See **`GenericEvent`** for more detail.

```
  # Project Imports
  from modules.helper.ModuleSupport import eventEx, GenericEvent

  dvSPStatus = GenericEvent()

  # Subscribe to the current Audio and Video input.
  dvSP.SubscribeStatus('Input', {'Type': 'Audio/Video'}, dvSPStatus.Trigger)

  # Handle the dvSPStatus Triggered event along with feedback code.
  @eventEx(dvSPStatus, 'Triggered')
  def handlescalersub(source, command, value, qualifier):
      if source is dvSPStatus and command == 'Input':
          ...
```

Misc. Examples

These examples are intended to show specific problems and how they are solved within the **Programming Framework**.

Power On/Off

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 device module. The MESet is used only for visual feedback on the user interface and does not affect the behavior of the device module.

Assuming TLP and dvDisplay exist and imports are correct.

```
btnPowerOn = Button(TLP, 1)
btnPowerOff = Button(TLP, 2)
mesPowerButtons = MESet([btnPowerOn, btnPowerOff])

@eventEx(mesPowerButtons.Objects, 'Pressed')
def handlePowerButtonPressed(button, state):
    if button is btnPowerOn:
        dvDisplay.Set('Power', 'On')
    elif button is btnPowerOff:
        dvDisplay.Set('Power', 'Off')

    mesPowerButtons.SetCurrent(button)
```

Level Gauge for Volume Status

The following is an example of creating a level gauge to track the status of a device module's Volume command. **eventEx** is used to set up the handler for the volume changed event to update the level of the gauge each time the module receives a different status for Volume. The device module will handle getting and providing the information automatically.

Assuming TLP and dvDisplay exist and imports are correct.

```
lvlVolumeStatus = Level (TLP, 3)
lvlVolumeStatus.SetRange(0, 100)
display1Status = GenericEvent()

display1.SubscribeStatus('Volume', None, display1Status.Trigger)

@eventEx(display1Status, 'Triggered')
def ReceivedNewVolumeStatus(source, command, value, qualifier):
    lvlVolumeStatus.Setlevel(value)

@Timer(3)
def PollVolume(timer, count):
    display1.Update('Volume')
```

Multiple Instances of a Module

Since classes are used in **Device Modules**, it is possible to create multiple instances of the same module and use them in a ControlScript project. The following is an example of using the same device 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)
```

Appendix

Module Support API

The following are part of the `ModuleSupport.py` library that contains tools that support device and other **Helper Modules**. `ModuleSupport.py` should be placed along with the Helper Modules (`<code_folder_path>/modules/helper/`).

eventEx

eventEx(*Object*, *EventName*, **args*, ***kwargs*)

Decorate a function to be the handler for *Object* when *EventName* occurs. This decorator offers extensions over the capabilities of extronlib's built-in event decorator:

- The event can trigger multiple handlers.
- In addition to property names, *EventName* can refer to method names.

The decorated function must have the exact signature as specified by the definition of EventName, which must appear in the *Object* class or one of its parent classes. Lists of *Object* and/or *EventName* can be passed in to apply the same handler to multiple events.

This decorator may be used as a drop-in replacement for extronlib's built-in event decorator; however, it introduces a minimal amount of latency both in event setup and dispatch.

Parameters
- **Object** (*Python object instance or list of instances*) – The Object instance(s) that define EventName.
- **EventName** (*str or list of str*) – Name of an event or list of event names. If a list, all object instances in *Object* must implement all event names.
- ***args** – A variable number of positional arguments that will be passed to the function that sets up the event handling (i.e. the EventName). Not applicable to events set via properties.
- ****kwargs** – Keyword arguments that will be passed to the function that sets up the event handling (i.e. the EventName). Not applicable to events set via properties.

Raises
- **AttributeError** – If *Object* does not have an *EventName* attribute.
- **TypeError** – If *EventName* is not a string or list of strings or if the attribute name in EventName is something other than a property or method.

Manual Events

These classes expand the types of changes that can be handled by functions decorated with extronlib.event or eventEx.

class **GenericEvent**(*Name='unnamed event'*)

Trigger an `extronlib.event` or **eventEx** event handler with a function call.

Some notification implementations use a callback function supplied as an argument in a method call. This is incompatible with how the @event decorator in extronlib is used. This class connects those callbacks to an event handler compatible with @event.

Examples

```
MyEvent = GenericEvent('my event')

@event(MyEvent, 'Triggered')
def HandleTriggered(src, value):
    print('Event "{}" was triggered with value {}.'.format(src.Name, value))

MyEvent.Trigger(123)
```

Trigger(**args, **kwargs*)

Calls to this method will cause the **Triggered** event to trigger.

It accepts a variable number of positional and keyword arguments which, along with this instances, are passed to the `Triggered` handler.

property **Triggered**

`Event` - Triggers when this instance's **Trigger** method is called.

The parameters passed to the handler are variable. The first parameter will be the **GenericEvent** instance triggering the event followed by any positional and/or keyword parameters that were passed to **Trigger**.

class **WatchVariable**(*Name='unnamed variable'*)

Wrap a variable in so that changes to that variable can trigger an event handler compatible with the `extronlib.event` and **eventEx** decorators (eventEx preferred).

Parameters	**Name** (*str*) – A friendly name used to identify this instance. Usable in logging output, for example. Defaults to 'unnamed variable'.

Use this class to signal to other parts of your program that the state of a system variable has been changed.

Examples

In the variable definition (*variables.py*) portion of the ControlScript program:

```
CallStatus = 'On Hook'
CallStatusWatch = WatchVariable('Video call status')

@eventEx(CallStatusWatch, 'Changed')
def HandleCallStatusChanged(src, value):
    global CallStatus
    # Remember to update the state of the variable.
    CallStatus = value
    print('{} changed to {}'.format(src.Name, value))
```

In the portion of the ControlScript program that handles device state changes:

```
from variables import CallStatusWatch

CallStatusWatch.Change('Ringing')
```

Add another eventEx handler if you need to react to a state change elsewhere in your program.

```
from variables import CallStatusWatch

@eventEx(CallStatusWatch, 'Changed')
def HandleCallStatusChanged(src, value):
    # Handle the new state
```

If the variable state is needed outside of a **Changed** handler:

```
import variables

print('CallStatus is', variables.CallStatus)
```

Change(**args*, ***kwargs*)

Calls to this method will cause the **Changed** event to trigger.

It accepts a variable number of positional and keyword arguments which, along with this instance, are passed to the Changed handler.

Examples

```
MyWatcher.Change('newstate')    # handlers will be called at this point
```

property **Changed**

`Event` - Triggers when this instance's **Change** method is called.

The parameters passed to the handler are variable. The first parameter will be the **WatchVariable** instance triggering the event followed by any positional and/or keyword parameters that were passed to **Change**.

Examples

```
@eventEx(MyWatcher, 'Changed')
def HandleMyWatcherChanged(src, value):
    print('{} changed to {}'.format(src.Name, value))
```

Loggers

class **ProgramLogLogger**

Implements a logger that sends log records to the Program Log.

Log(**recordobjs*, *sep=' '*, *severity='info'*)

Print recordobjs to ProgramLog, separated by sep.

Parameters

- **recordobjs** (*objects*) – objects to print in the log. Each object is converted to a string prior to printing.
- **sep** (*str*) – the separator to add between each recordobj. Defaults to ' '.
- **severity** (*str*) – User defined indicator of attention suggested (e.g. 'error', 'info', 'warning').

Example

```
logger = ProgramLogLogger()
msg = 'log record.'
logger.Log('This is', 'a', msg)
```

class **TcpServerLogger**(*IPPort*, *Interface='Any'*, *end='\n'*)

Implements a logger that sends log records to all clients connected to a TCP server.

Parameters
- **IPPort** (*int*) – IP port number of the listening service.
- **Interface** (*str*) – Defines the network interface on which to listen (`'Any'`, `'LAN'`, or `'AVLAN'`)
- **end** (*str*) – The terminator for each log record sent to clients. Defaults to '\n'.

Example

```
logger = TcpServerLogger(5000)
msg = 'log record.'
logger.Log('This is', 'a', msg)
```

property **IPPort**

The server's listening port.

property **Interface**

The interface this logger is listening on for client connections.

Log(**recordobjs*, *sep=' '*, *severity='info'*)

Sends recordobjs to all connected clients, separated by sep.

Parameters
- **recordobjs** (*objects*) – objects to send to clients. Each object is converted to a string prior to sending.
- **sep** (*str*) – the separator to add between each recordobj. Defaults to ' '.
- **severity** (*str*) – User defined indicator of attention suggested (e.g. 'error', 'info', 'warning').

class **TraceLogger**

Implements a logger that sends log records to Trace Messages.

Log(**recordobjs*, *sep=' '*, *severity='info'*)

Print recordobjs to Trace, separated by sep.

Parameters
- **recordobjs** (*objects*) – objects to print in the trace. Each object is converted to a string prior to printing.
- **sep** (*str*) – the separator to add between each recordobj. Defaults to ' '.
- **severity** (*str*) – User defined indicator of attention suggested (e.g. 'error', 'info', 'warning').

Example

```
logger = TraceLogger()
msg = 'log record.'
logger.Log('This is', 'a', msg)
```