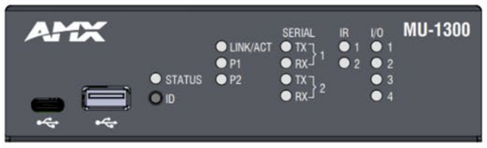
PROGRAMMING GUIDE

MUSE Automation Controllers



Contents

VS Code Extension

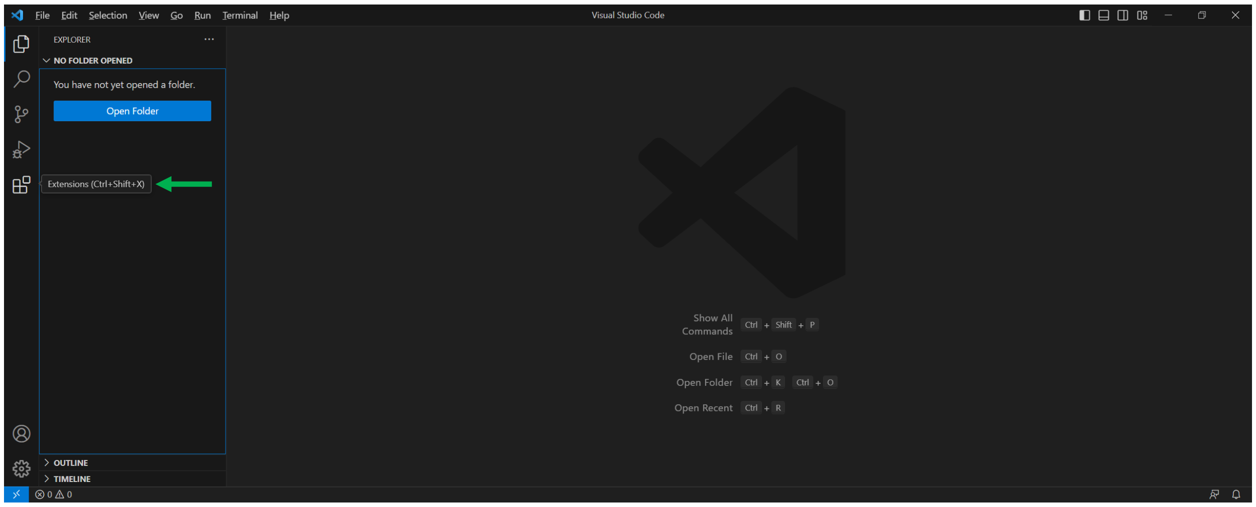
AMX has provided a lightweight VS Code extension to facilitate the configuration and programming of the MUSE series of controllers. Using the extension, you can:

- Discover MUSE on the network
- Connect to view Devices and Programs
- Write scripts for MUSE
- Upload scripts
- Attach to the standard output of the scripts for debugging

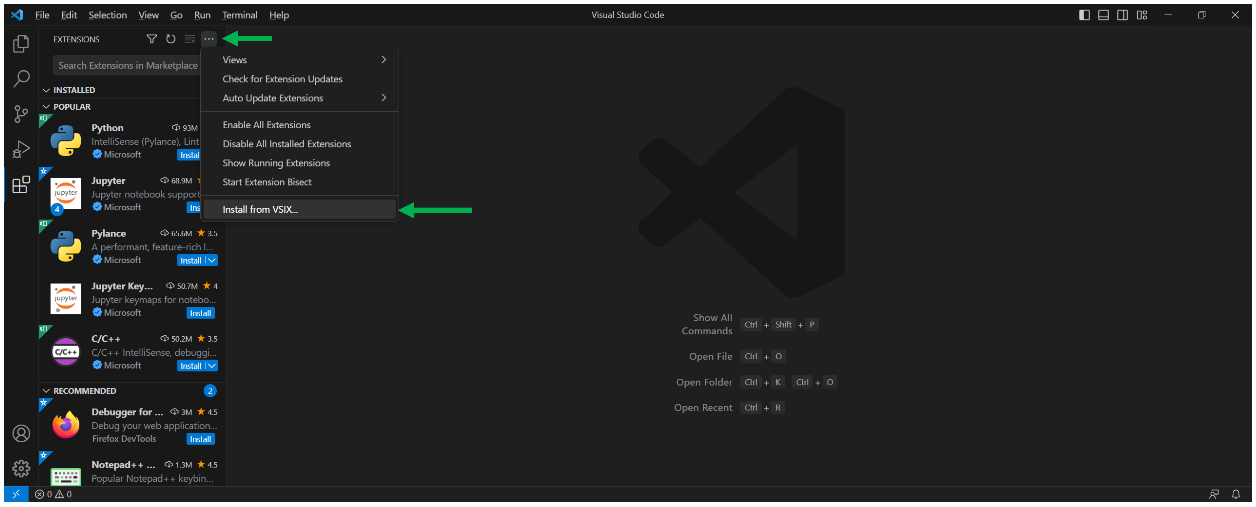
VS Code Extension installation

The Mojo VS Code Extension is available as a standalone .vsix installation file. To install the extension:

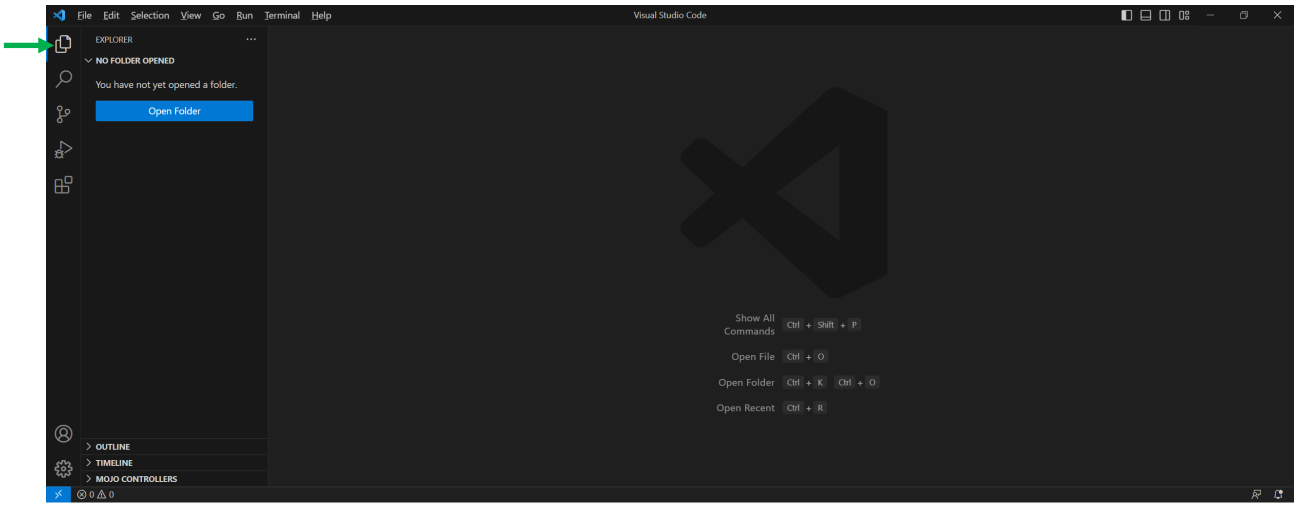
1. Launch VS Code (available from: https://code.visualstudio.com/download)
2. Choose the Extensions tab from the vertical ribbon


File Edit Selection View Go Run Terminal Help
Visual Studio Code
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Extensions (Ctrl+Shift+X)
Show All Commands Ctrl + Shift + P
Open File Ctrl + O
Open Folder Ctrl + K Ctrl + O
Open Recent Ctrl + R
OUTLINE
TIMELINE

3. Click on the Ellipses to show the Extensions context menu and select “Install from VSIX…”



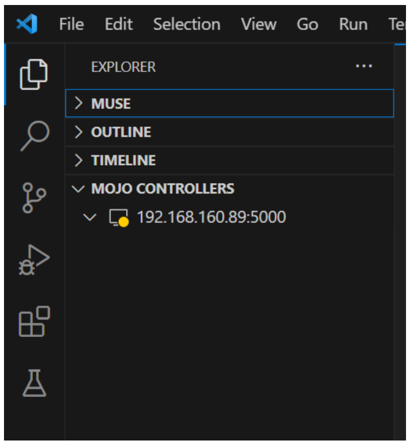
4. Choose the mojo-<version>.vsix file and click Install…
5. Click the Explorer tab on the vertical ribbon..


File Edit Selection View Go Run Terminal Help
Visual Studio Code
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Show All Commands Ctrl + Shift + P
Open File Ctrl + O
Open Folder Ctrl + K Ctrl + O
Open Recent Ctrl + R
OUTLINE
TIMELINE
MOJO CONTROLLERS

6. If this is the first time you have run the VS Code application, you will be prompted to open a folder. This folder will contain all the projects you want to group together. Each project will be in a separate folder and can contain any files your project or script needs. Choose a folder to create a place for your projects.

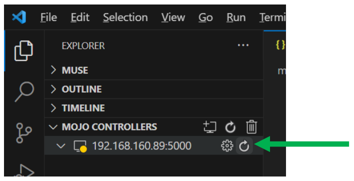
Discovering nearby controllers

In the Explorer there is a node labelled Mojo Controllers. This tree contains all the controllers you have interacted with recently as well as any controllers that have been discovered nearby. The Mojo VS Code extension sends an HControl discovery broadcast message. Any controllers that respond will appear in the list.

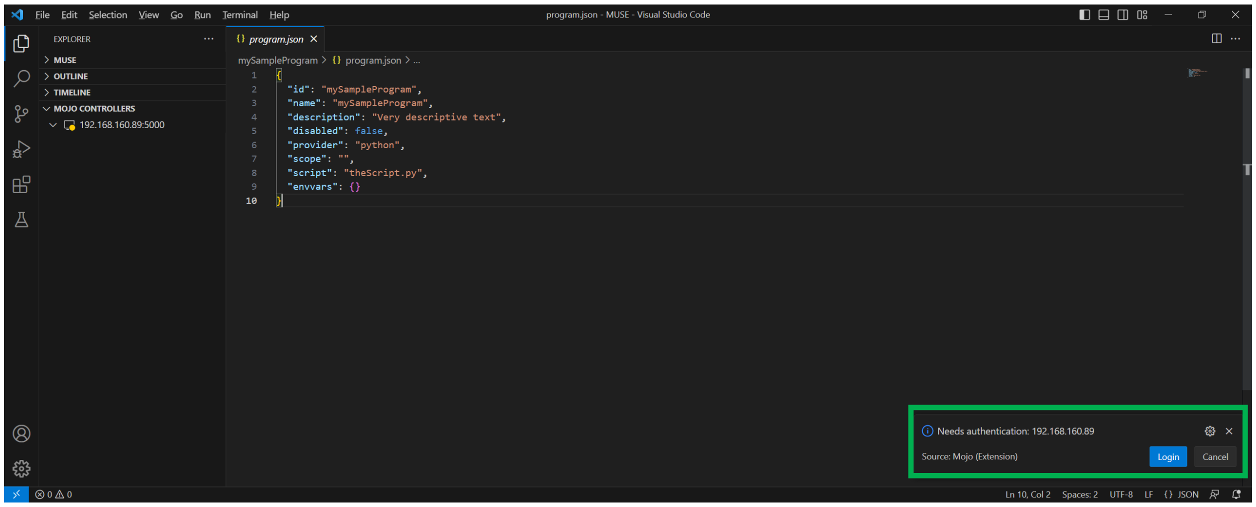


In the above example, a MUSE controller has been discovered at the IP address 192.168.160.89.
Note: the yellow pip on the controller icon indicates that it is present but not authorized.

To log in to the controller, click the refresh icon next to the IP address:



A login prompt will appear at the lower-right portion of the screen:

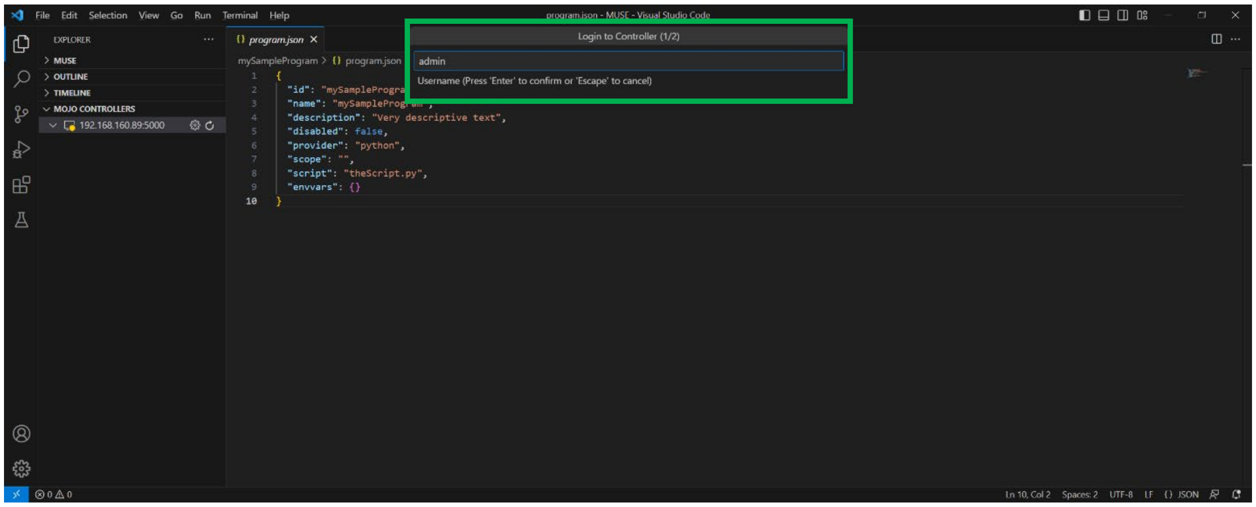

File Edit Selection View Go Run Terminal Help
program.json - MUSE - Visual Studio Code
EXPLORER
MUSE
OUTLINE
TIMELINE
MOJO CONTROLLERS
192.168.160.89:5000
program.json
mySampleProgram > {} program.json > ...
Ln 10, Col 2 Spaces: 2 UTF-8 LF {} JSON

```
{
  "id": "mySampleProgram",
  "name": "mySampleProgram",
  "description": "Very descriptive text",
  "disabled": false,
  "provider": "python",
  "scope": "",
  "script": "theScript.py",
  "envvars": {}
}
```

Needs authentication: 192.168.160.89

Source: Mojo (Extension) Login Cancel

Select 'Login' and follow the prompts at the top of the screen.


File Edit Selection View Go Run Terminal Help
program.json - MUSE - Visual Studio Code
EXPLORER
MUSE
OUTLINE
TIMELINE
MOJO CONTROLLERS
192.168.160.89:5000
program.json
Ln 10, Col 2 Spaces: 2 UTF-8 LF {} JSON

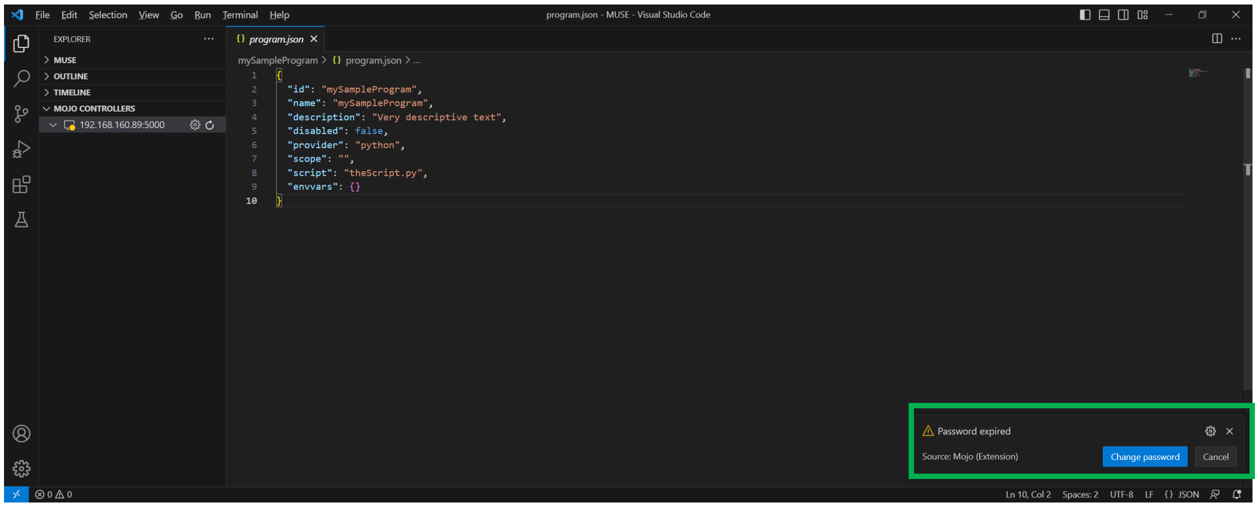
Login to Controller (1/2)

admin

Username (Press 'Enter' to confirm or 'Escape' to cancel)

The default credentials are admin : password

If this is the first time you have logged in to this controller, you must choose a new password. By default, password security is set to 'low', so any password besides 'password' is acceptable. Click 'Change Password' to continue:


File Edit Selection View Go Run Terminal Help
program.json - MUSE - Visual Studio Code
EXPLORER
MUSE
OUTLINE
TIMELINE
MOJO CONTROLLERS
192.168.160.89:5000
program.json
mySampleProgram > {} program.json > ...
Ln 10, Col 2 Spaces: 2 UTF-8 LF {} JSON

Password expired

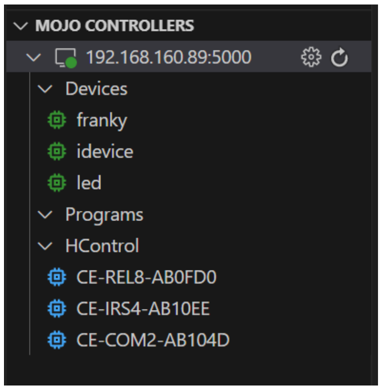
Source: Mojo (Extension) Change password Cancel

Follow the prompts to change the password and log in.

Mojo Controllers tree view

Once authenticated, the tree view for a Mojo controller contains three nodes:

- Devices – Known AMX devices or modules that the script can access
- Programs - Any scripts loaded to the controller, running or disabled
- HControl – Any Harman HControl devices that the MUSE controller has discovered

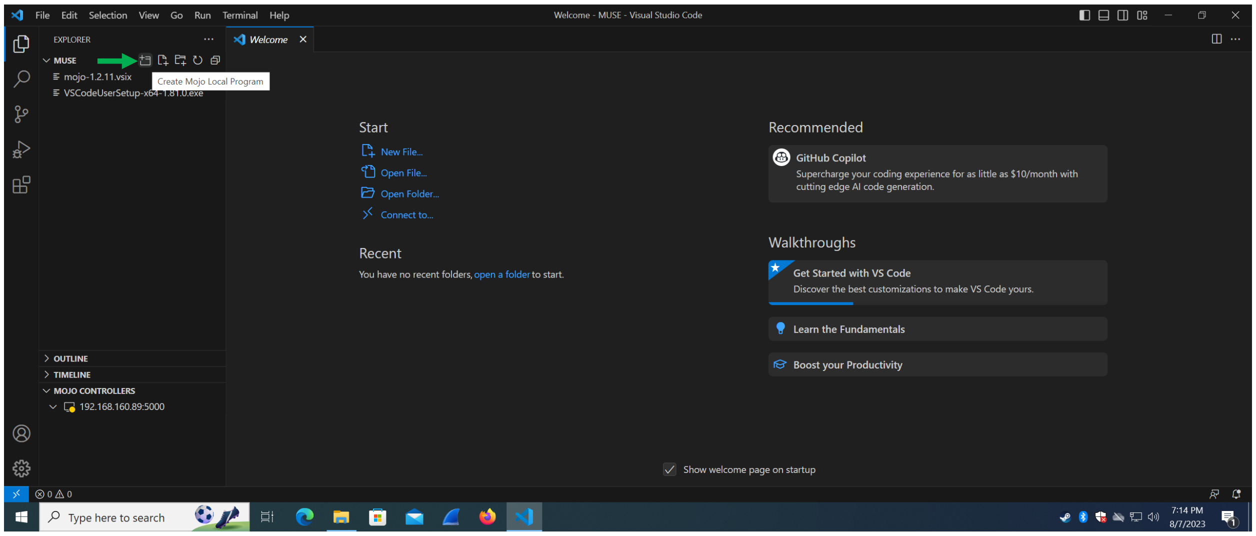


MOJO CONTROLLERS
- 192.168.160.89:5000
  - Devices
    - franky
    - idevice
    - led
  - Programs
  - HControl
    - CE-REL8-AB0FD0
    - CE-IRS4-AB10EE
    - CE-COM2-AB104D

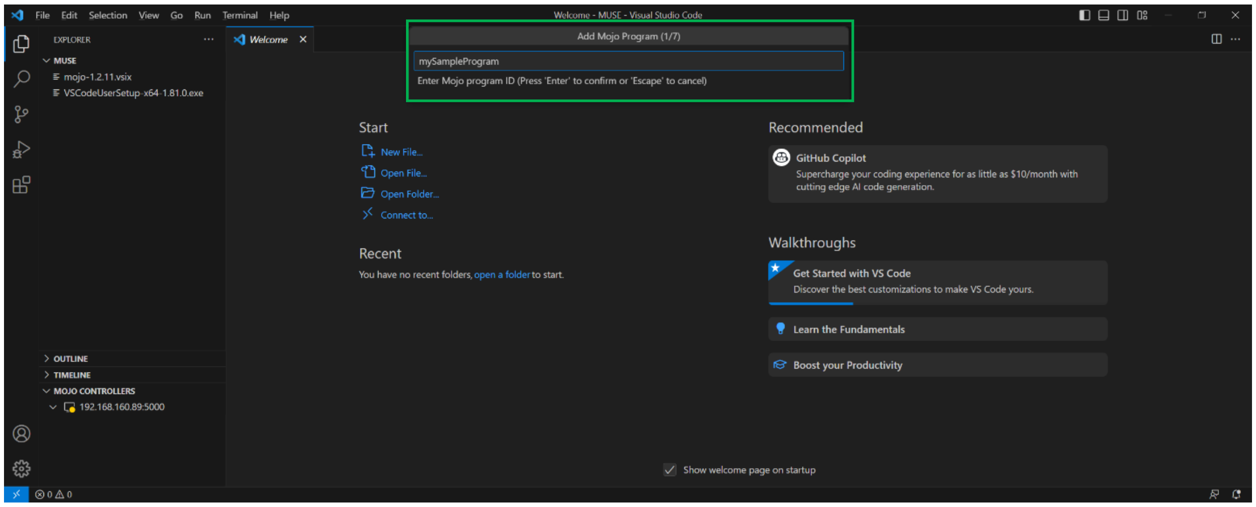
In this example, there are three Devices named franky, idevice, and led.
There are no programs currently loaded
There are three nearby HControl devices. Each of these are the CE- family of control extender.
Note: selecting nearby HControl devices will display their IP address along with other properties.

Creating new scripts

1. To create a new project to contain a script for MUSE to run, choose the "Create Mojo Local Program" icon in your folder:


Welcome - MUSE - Visual Studio Code
EXPLORER
MUSE
mojo-1.2.11.vsix
VSCodeUserSetup-x64-1.81.0.exe
Create Mojo Local Program
Welcome
Start
New File...
Open File...
Open Folder...
Connect to...
Recent
You have no recent folders, open a folder to start.
Recommended
GitHub Copilot
Supercharge your coding experience for as little as $10/month with cutting edge AI code generation.
Walkthroughs
Get Started with VS Code
Discover the best customizations to make VS Code yours.
Learn the Fundamentals
Boost your Productivity
Show welcome page on startup
OUTLINE
TIMELINE
MOJO CONTROLLERS
192.168.160.89:5000
Type here to search
7:14 PM
8/7/2023

2. Follow the prompts at the top of the screen to create your project:



Add Mojo Program (1/7)

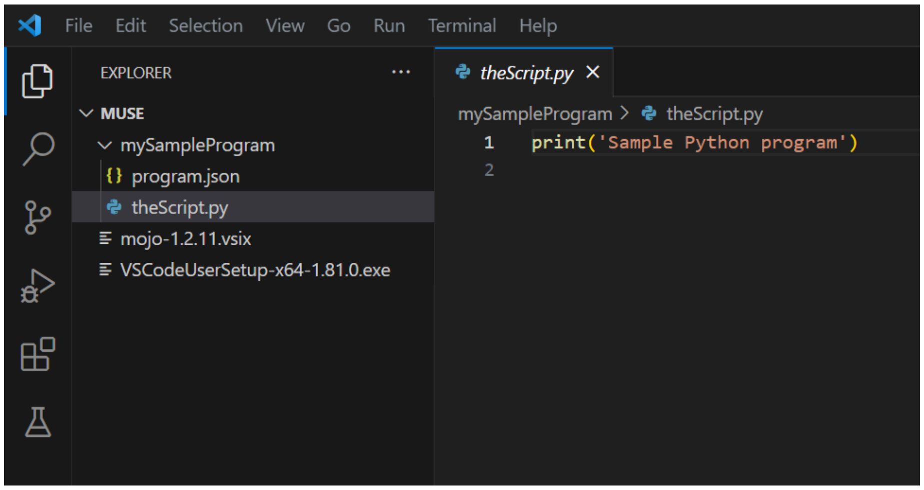
mySampleProgram

Enter Mojo program ID (Press 'Enter' to confirm or 'Escape' to cancel)

You will be prompted for:

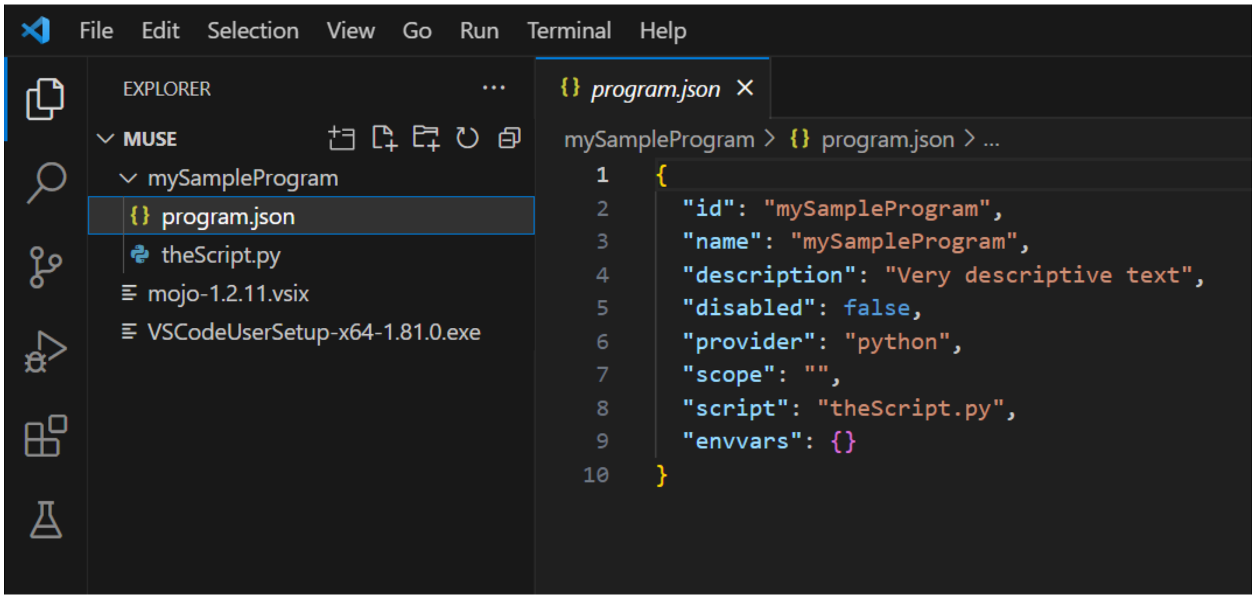
- Mojo Program ID – The friendly name displayed within VS Code for the script
- Program Description – Programmer-provided metadata for the project
- Program Enable/Disable – A flag to determine if the project should run if loaded
- Program Provider – Choose the scripting language for the project
- Program Scope – Future use: logically divide devices and scripts into groups
- Program Entry Script – The name of the script to use as a starting point

At this stage a folder and two files have been created for your MUSE script based on your input.



```python
print('Sample Python program')
```

In this example, 'theScript.py' is the Python script that has been created. The program.json file contains the MUSE script's target information. If at some later time you create a different Python file in this project and would like to choose it as the starting point, you could edit program.json to make it so.



```json
{
  "id": "mySampleProgram",
  "name": "mySampleProgram",
  "description": "Very descriptive text",
  "disabled": false,
  "provider": "python",
  "scope": "",
  "script": "theScript.py",
  "envvars": {}
}
```

For a basic script that runs at startup, no further intervention is needed.

NOTE: For Python scripts only, the following two lines need to be added to your script to properly use all the available MUSE resources:

At the top:

```python
from mojo import context
```

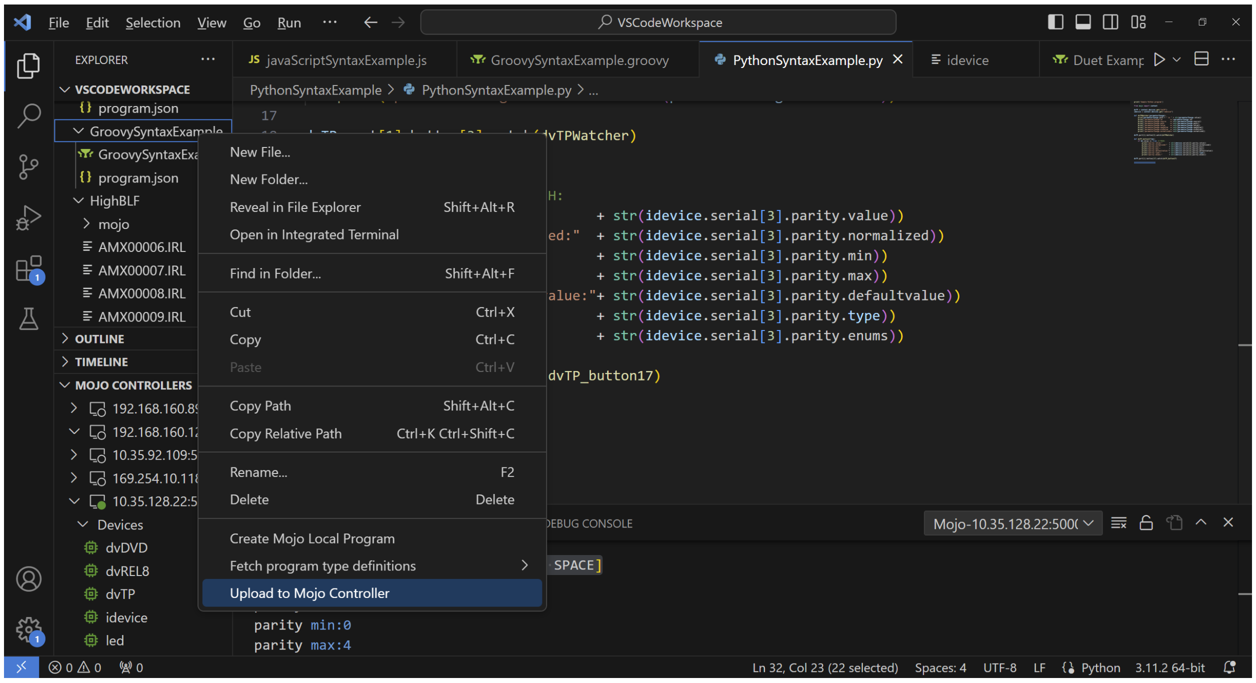
At the bottom:

```
context.run(globals())
```

Loading Scripts

To quickly load a script to a controller, right click on the script's main folder. In the example above, ***mySampleProgram*** is the main folder.

Choose Upload to Mojo controller. A dialog will appear at the top center of the VS Code application. First, confirm the script to be loaded. Next, choose any controller you are currently attached to as a target for uploading. Clicking OK will upload the script and start running.



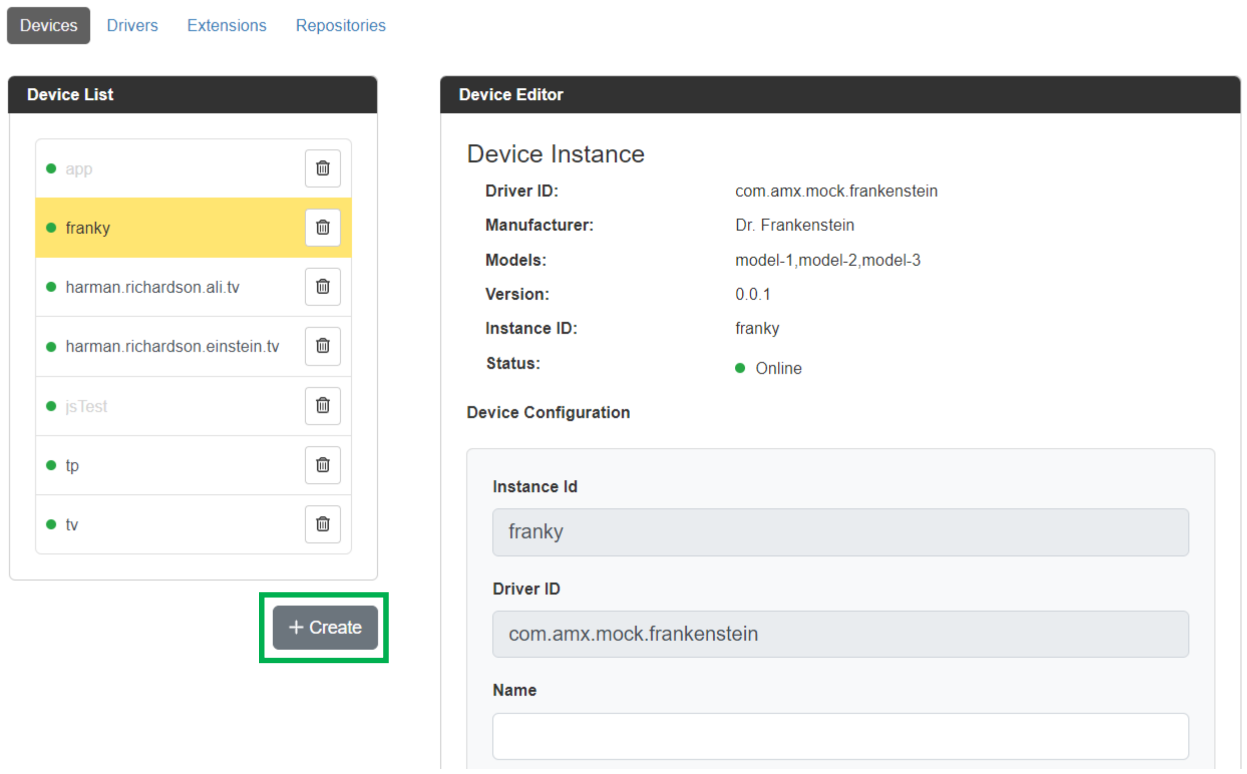
```
dvTPWatcher)
H:
       + str(idevice.serial[3].parity.value))
ed:"   + str(idevice.serial[3].parity.normalized))
       + str(idevice.serial[3].parity.min))
       + str(idevice.serial[3].parity.max))
alue:"+ str(idevice.serial[3].parity.defaultvalue))
       + str(idevice.serial[3].parity.type))
       + str(idevice.serial[3].parity.enums))

dvTP_button17)
```

```
SPACE]
parity min:0
parity max:4
```

Device Declarations

MUSE uses a dynamic device binding system that allows each running script to access any device that is attached to the MUSE Controller. Touch panels, keypads, modules, and other ICSP, HiQNet, and HControl devices are configured outside of the script.

Devices are acquired by the scripts by name. The naming of the attached devices depends on the connection type. Some are assignable at the time of configuration, some are automatic.



Device Instance

Driver ID:	com.amx.mock.frankenstein
Manufacturer:	Dr. Frankenstein
Models:	model-1,model-2,model-3
Version:	0.0.1
Instance ID:	franky
Status:	● Online

Device Configuration

Instance Id: franky

Driver ID: com.amx.mock.frankenstein

Name:

The primary method to declare a device is through the MUSE Controller's web server. Depending on the connection type, some devices may auto-register. The following is a description of the behavior of each device type:

ICSP

ICSP connections have multiple behaviors that may be supported. URL mode, Listen mode, NDP each behave slightly differently.

URL Mode

The most common connection method is URL mode. The peripheral devices (a touchpanel, keypad, etc…) is given the IP address or DNS resolvable hostname of the MUSE Controller as a target to connect to. The device reaches out on TCP port 1319 (or 1320 for ICSPS) and negotiates a connection with the controller.

Once connected, URL Mode devices will be automatically assigned a name based on the device number. If you have a touchpanel set to device number 10001, the device name will be AMX-10001.

NDP

MUSE supports the NDP method of connection. Devices in NDP mode can be discovered and bound to the controller. The controls are available in the CLI and the controller's web server.

Once created, it will behave as a URL mode device and receive a name based on the device number.

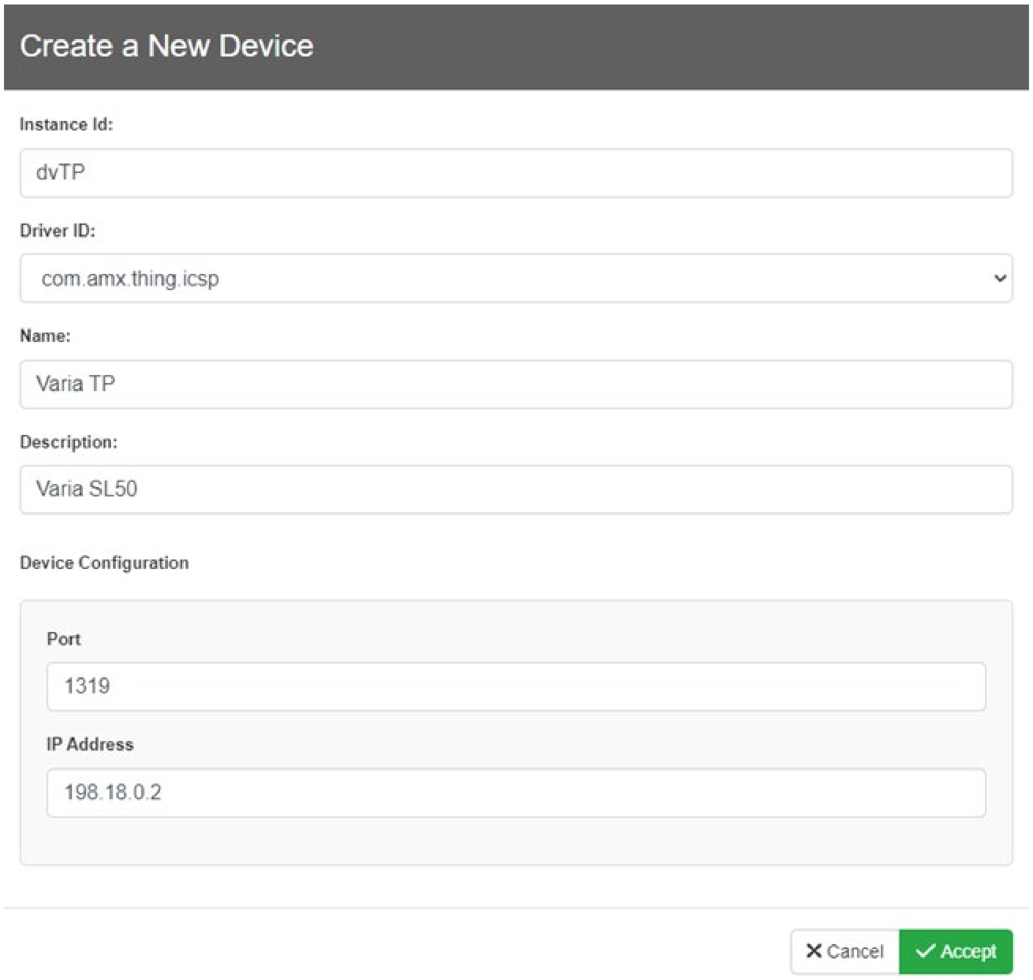
Listen Mode

ICSP Listen Mode reverses the connection initiation. A device in Listen mode opens a socket to listen to incoming port 1319 connections. It is the controller's responsibility to initiate a connection with the device. To inform the controller of the IP address or DNS resolvable hostname of the target device, a Device needs to be configured. The primary method of configuring the device is on the MUSE Controller's web server. Point a web browser at the URL of the MUSE Controller and perform the log in as prompted.

Next, navigate to the Devices page by choosing the System tab, then selecting Devices. Click the **Create+** button to start configuration of the device.

For each Device configuration, a name is required. The name is what is used to access the device from a script.

To create an ICSP Listen Mode device definition, use the Driver ID pulldown and choose the com.amx.thing.ICSP entry. This will reveal two more fields that are specific to the ICSP driver. The IP address field is used to convey the target IP address for the device in Listen mode. The target port is read-only information containing the default ICSP port.



Create a New Device

Instance Id: dvTP

Driver ID: com.amx.thing.icsp

Name: Varia TP

Description: Varia SL50

Device Configuration

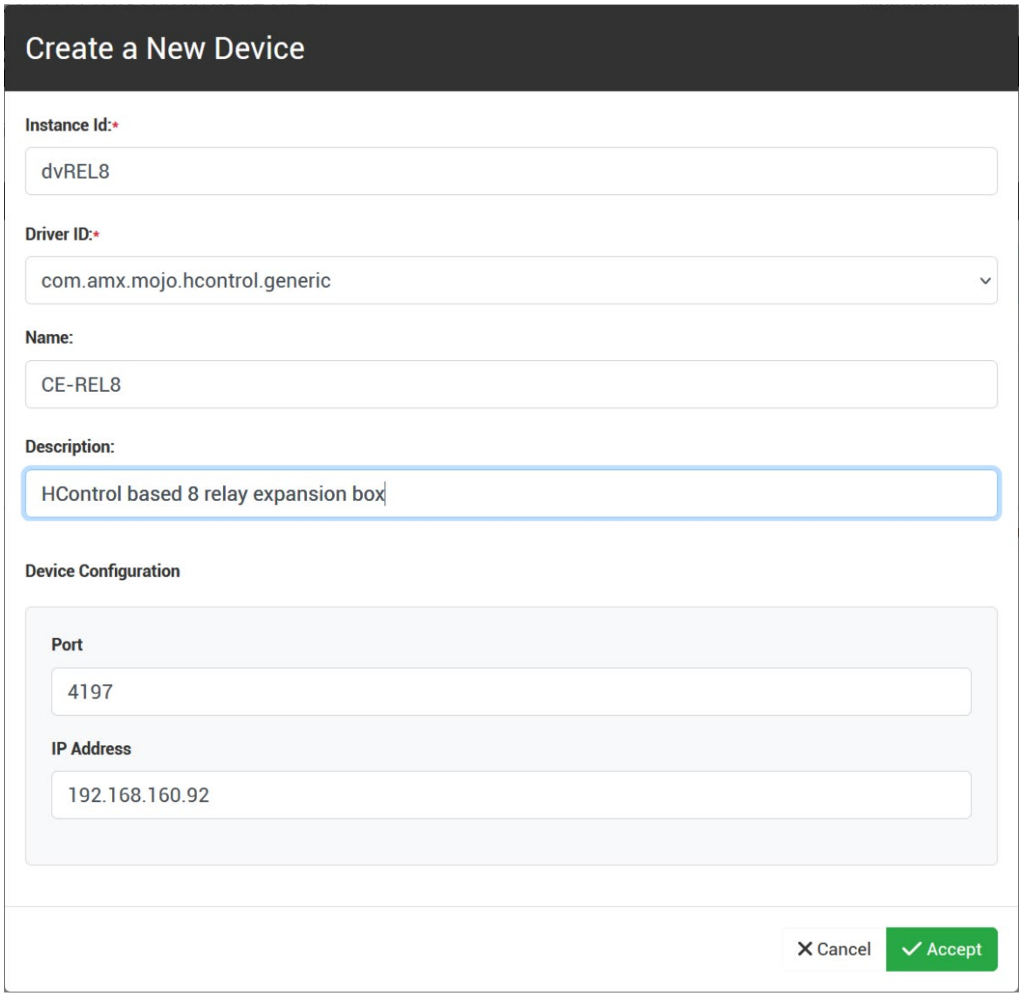
Port: 1319

IP Address: 198.18.0.2

The other fields are metadata that do not affect the functionality of the ICSP device.

HControl

All HControl devices listen for a connection. To connect to an HControl device, choose the com.amx.thing.hcontrol.generic item in the Driver ID pulldown.


Create a New Device

Instance Id:*

dvREL8

Driver ID:*

com.amx.mojo.hcontrol.generic

Name:

CE-REL8

Description:

HControl based 8 relay expansion box

Device Configuration

Port

4197

IP Address

192.168.160.92

Device Online/Offline Behavior

There are two ways to determine the online/offline status of a Device.

The get the current state, you can call the .isOnline() or .isOffline() method.

To be informed of changes to the online/offline status of the Device, use the .online() or .offline() method to register a callback.

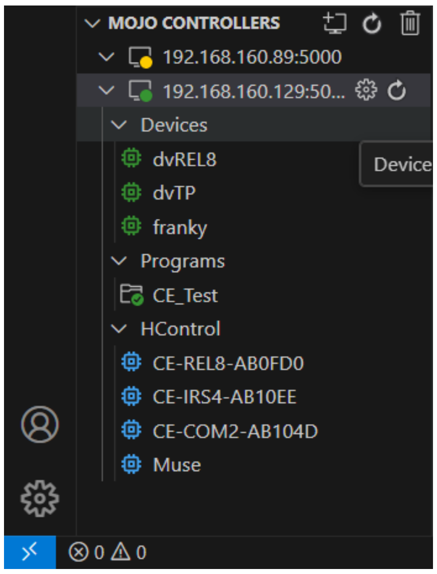
Interpreting Descriptor Files

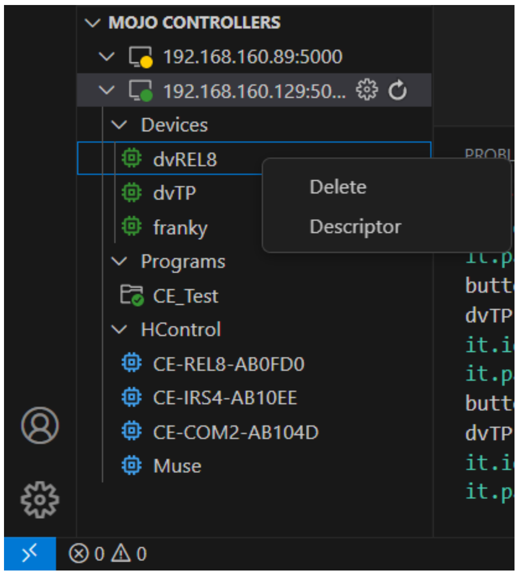
Any Device in MUSE has its capabilities described in a Descriptor File. This document contains information about the interface you can use to interact with the device. In the case of HControl, this document is provided by the device itself during the connection process. In the case of other communication methods like ICSP and HiQNet the document is dynamically generated based on information provided during the connection process.

The Descriptor File is a .json formatted document. This makes it reasonably human readable and available to parse programmatically.

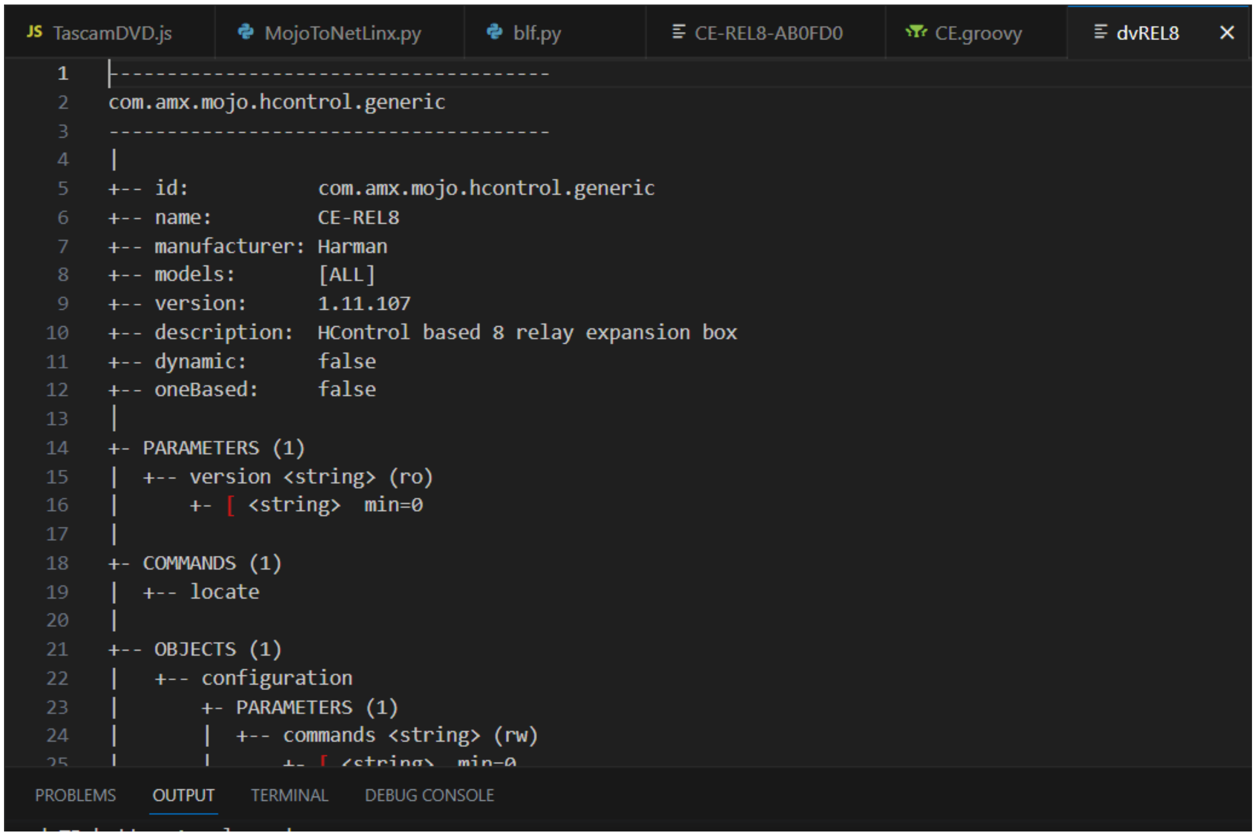
There are several ways to access this information. The VS Code extension provides a quick way to see an expanded view of the information.

Expand the tree view of the controller to reveal the Devices node:



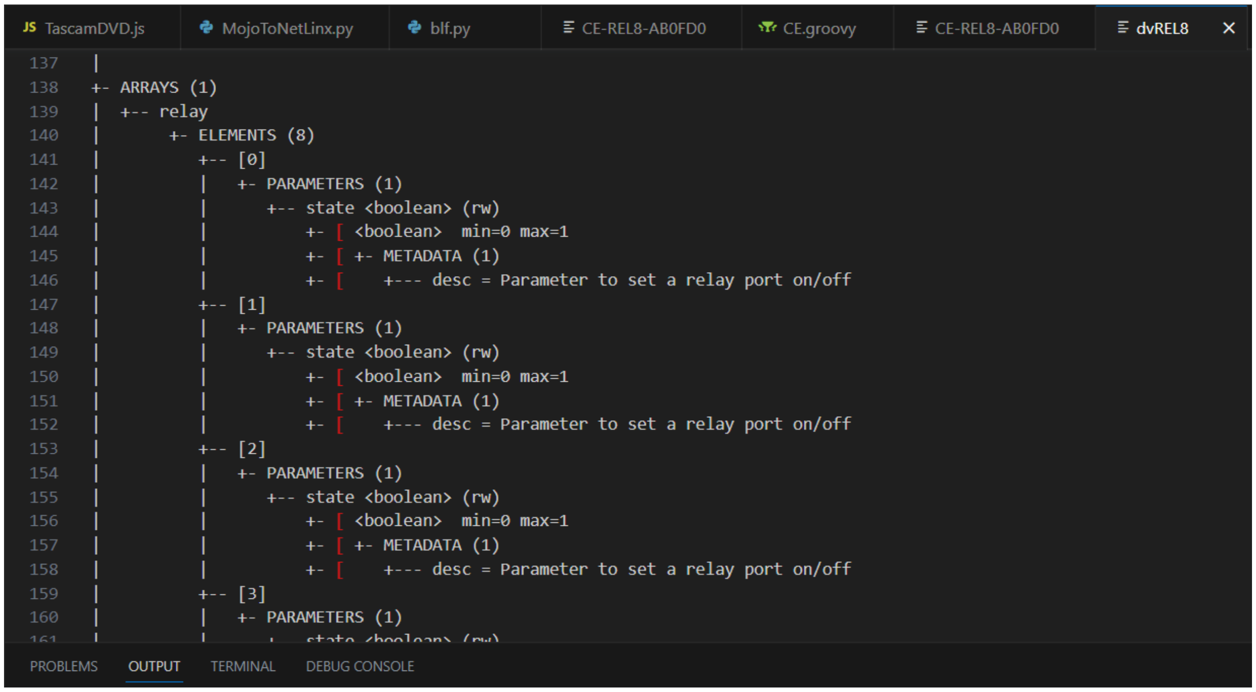


Right click on the device in question and choose Descriptor from the context menu. A file will open that is an interpretation of the Descriptor file, not the file itself.



```
--------------------------------------
com.amx.mojo.hcontrol.generic
--------------------------------------
|
+-- id:           com.amx.mojo.hcontrol.generic
+-- name:         CE-REL8
+-- manufacturer: Harman
+-- models:       [ALL]
+-- version:      1.11.107
+-- description:  HControl based 8 relay expansion box
+-- dynamic:      false
+-- oneBased:     false
|
+- PARAMETERS (1)
|  +-- version <string> (ro)
|      +- [ <string>  min=0
|
+- COMMANDS (1)
|  +-- locate
|
+-- OBJECTS (1)
|   +-- configuration
|       +- PARAMETERS (1)
|       |  +-- commands <string> (rw)
|       |      +- [ <string>  min=0
```

The file tends to start with configuration information that may not apply to the programming task at hand. The typical control-related information will be in the Arrays portion at that end of the file.



```
|
+- ARRAYS (1)
|  +-- relay
|       +- ELEMENTS (8)
|          +-- [0]
|          |   +- PARAMETERS (1)
|          |      +-- state <boolean> (rw)
|          |          +- [ <boolean>  min=0 max=1
|          |          +- [ +- METADATA (1)
|          |          +- [    +--- desc = Parameter to set a relay port on/off
|          +-- [1]
|          |   +- PARAMETERS (1)
|          |      +-- state <boolean> (rw)
|          |          +- [ <boolean>  min=0 max=1
|          |          +- [ +- METADATA (1)
|          |          +- [    +--- desc = Parameter to set a relay port on/off
|          +-- [2]
|          |   +- PARAMETERS (1)
|          |      +-- state <boolean> (rw)
|          |          +- [ <boolean>  min=0 max=1
|          |          +- [ +- METADATA (1)
|          |          +- [    +--- desc = Parameter to set a relay port on/off
|          +-- [3]
|          |   +- PARAMETERS (1)
|          |      +-- state <boolean> (rw)
```

From this example, we can see that the CE-REL8 has an array named ‘relay’ with 8 elements. Each relay has a parameter named ‘state’ that is a Boolean. If we want to engage the relay, we would set the ‘state’ parameter to true.

Here are some syntax examples in the supported languages. In each, assume we have a Device reference held in a variable named dvREL8.

Groovy syntax example:

```
dvREL8.relay[0].state = true
```
Python syntax example:

```
dvREL8.relay[0].state = True
```
JavaScript syntax example

```
dvREL8.relay[0].state = true
```
For this activity, there is very little difference between the syntax in any particular language.

More ways to get the Descriptor and related information

The MUSE controller offers an SSH connection that has many useful CLI commands.   To get the same interpretation of the descriptor is available in VS Code, the CLI command is:

```
admin@mojo>driver:describe dvREL8
```
To obtain the actual JSON document, use:

```
admin@mojo>driver:describe -j dvREL8
```
To get the list of commands or parameters that pertain to the device, use:

```
admin@mojo>device:commands
admin@mojo>device:parameters
```

To obtain a list of current devices, use:

```
admin@mojo>device:list
```

Event-based Programming

Adding a watch/listen

One of the most basic requirements of AV control system programming is reacting to user input from some form of User Interface. Typically, touchscreens and keypads provide the user the ability to inform the system of their desired functionality.

MUSE has two methods to be notified about changes: **.watch( )** and **.listen( )**

The **.watch( )** method is for parameters. Parameters are attributes that keep their state and can be queried for their value.

The **.listen( )** method is for events. Events, once fired, are gone. You cannot query the value of an event outside of the **.listen( )** callback.

In MUSE, to react to button presses you create a ***watch,*** because buttons are defined as parameters. The watch argument contains a callback function that will be called to process the button event.

Groovy Syntax:

```
dvTP.port[1].button[255].watch( {
    println("button is "+it.value)
    println("dvTP button "+it.id+" "+ ((it.value == true)?"pressed":"released"));
    println("it.id=="+it.id)
    println("it.path=="+it.path)
    })
```

Groovy provides a convenient mechanism for writing complex code within the watch. This Groovy Closure includes the passed parameter containing the event information. If unnamed, the passed parameter is named ***it***. In this example, ***it*** is a Parameter Update Structure that contains everything about the parameter change notification.

For ICSP events, the path contains the port and channel of the button that caused the event in the form port/<p>/button/<b>. So, if button 1 on port 1 was pressed, the event.path value would contain:

`port/1/button/1`

In languages that support regex, this lends itself to a convenient pattern match. For example, in Groovy you can use:

```
  def eventPathPattern = ~"port/[0-9]+/button/[0-9]+"
```

…then extract the event data with:

```
      def eventMatches = it.path =~ eventPortAndChannelPattern
      eventMatches.find()
```

`EventMatches` then becomes a collection with the port and channel numbers as the two elements

Python Syntax:

```
dvTP.port[1].button[1].watch(dvTP_port1_button1_watch)
```

…where `dvTP_port1_button1_watch` is a user defined function used as a callback.   When the event is processed, this function will be called and passed a parameter update structure containing the parameter change information.

Elsewhere in your code, you need to implement this function:

```
def dvTP_port1_button1_watch(theButtonPress):
    #do stuff with the data in theButtonPress
    print('theButtonPress.path==' +theButtonPress.path)
```

All the event parameters are available in each language. See The Parameter Update Structure for details.

JavaScript syntax:

```
dvTP.port[1].button[4].watch(dvTP_button1_watch)
```

…where dvTP_button1_watch is a user define function used as a callback.

Elsewhere in your code, you need to implement this function:

```
function dvTP_button1_watch(theButtonPress) {
    context.log('dvTP_button_watch(theButtonPress) called')
    context.log(path:'+theButtonPress.path+" is "+theButtonPress.newValue)
```

…where theButtonPress is a structure passed into the callback. All the event parameters are available in each language. See The Parameter Update Structure and The Event Structure for details.

Timelines

MUSE provides a centralized timing mechanism to use within the scripts. To ensure that events are processed in the order that they are received, events are processed one at a time. Consequently, invoking a *sleep* method on a thread that is processing an event will delay the processing of the next event until the *sleep* returns. Using a Timeline side-steps this issue by running the delayed code in the centralized timing thread.

To use a Timeline, an instance of a timeline needs to be retrieved from the context.

In Python, the syntax is:

```
tick = context.services.get("timeline")
```

In this case, the variable *tick* will contain the timeline object. To start the timeline, the syntax is:

```
tick.start([1000],True,-1)
```

…where [1000] is an array of times in milliseconds.

The Boolean is the 'relative' flag. If there are multiple times in the array, it determines whether the timing are treated as relative delays between triggers, or as an independent list of times that may trigger out of sequence relative to their order in the list.

The final parameter is the 'repeat' value. A value of -1 indicates that the timeline should run forever. A timeline started with a value of 0 will run once. The value indicates the number of repetitions.

To receive events from the timeline, use this syntax:

```
tick.expired.listen(tickListener)
```

Where *tickListener* is the function that will be called when the timer expires. This is not a fixed name. The script writer creates and names this function:

```
def tickListener(tlEvent):
      # do the timed stuff
```

The *tlEvent* parameter contains information specific to the timeline event that just triggered.

```
def tickListener(tlEvent):
      print("ticklistener expired: sequence=" + tlEvent.arguments.sequence +
        ", time=" + tlEvent.arguments.time +
        ", repetition=" + tlEvent.arguments.repetition)
      Repetition = tlEvent.arguments["sequence"]
```

Timelines have the following methods: start(timeArray,relative,repetition), stop(), pause(), restart()

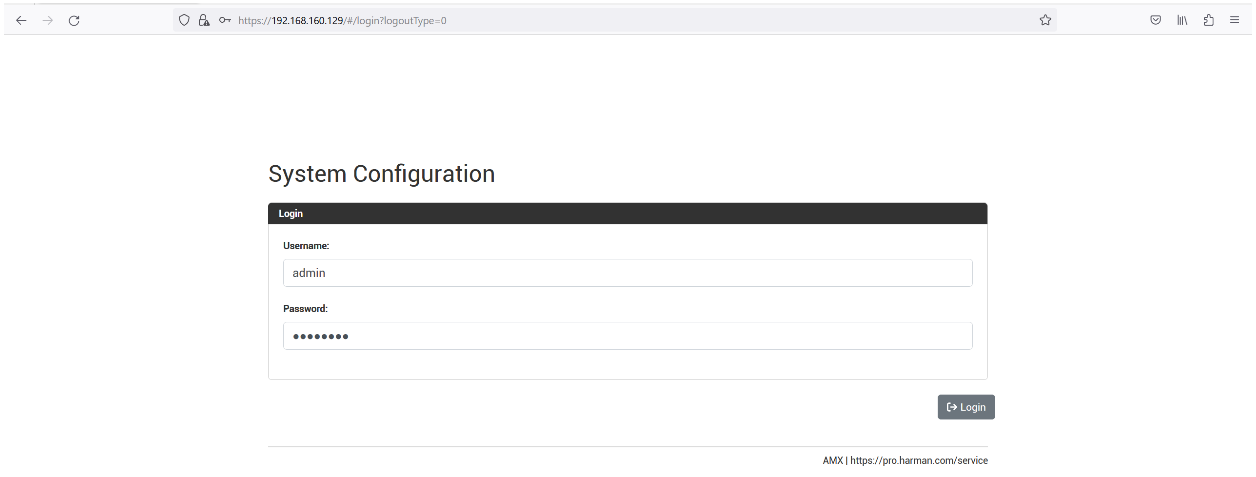
Duet Modules

Initial setup

Adding the Duet Extension

By default, the Duet Driver Provider is not installed on the MU-series controller. This Extension must be enabled to load and run Duet-based drivers.

Log in to your MUSE controller's configuration web page:


https://192.168.160.129/#/login?logoutType=0

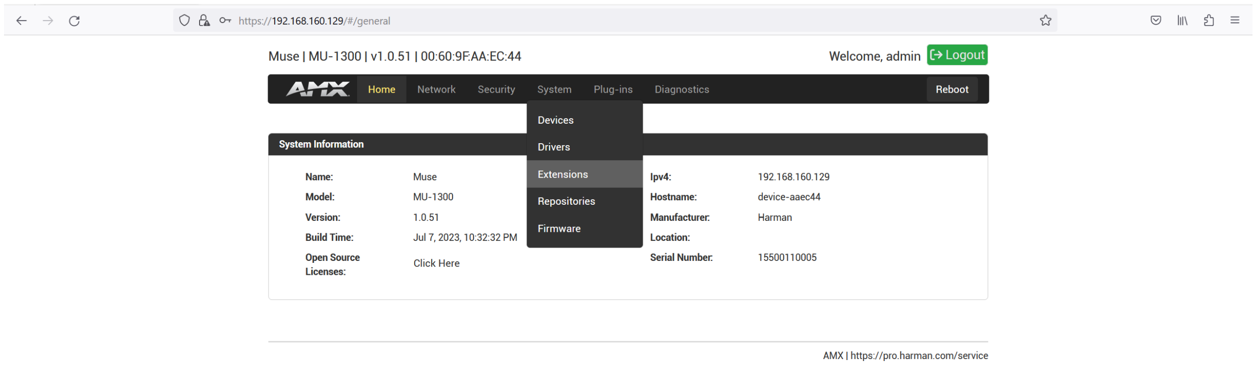
System Configuration

Login

Username: admin

Password: ••••••••

In the menu bar, choose System -> Extensions

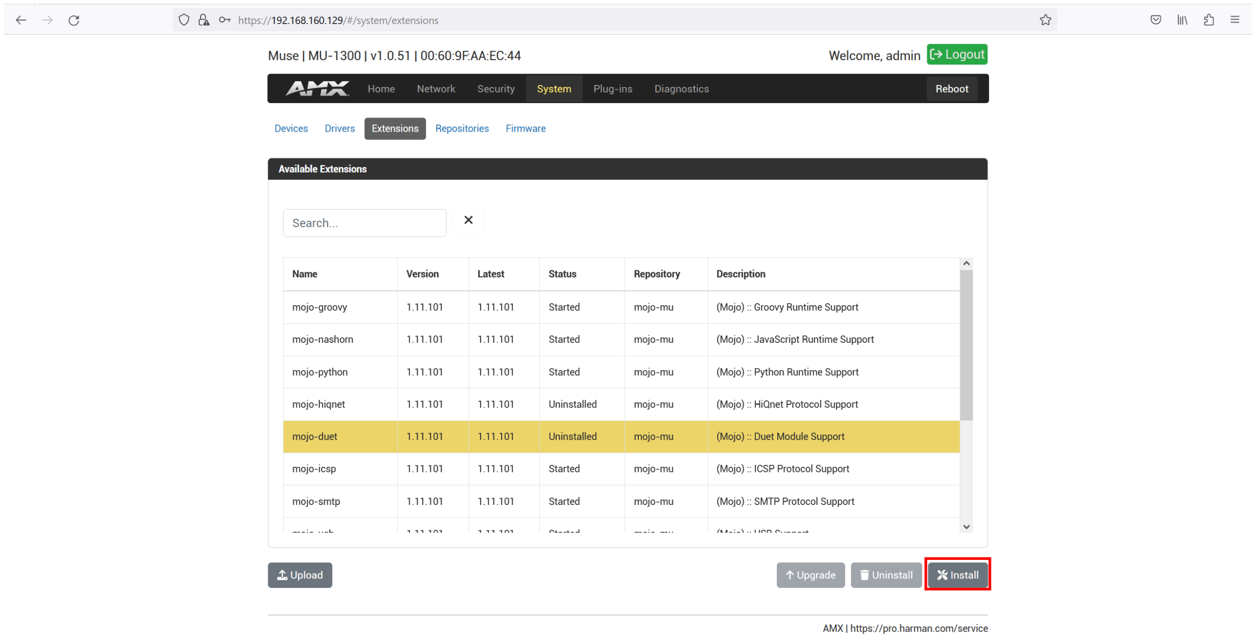

https://192.168.160.129/#/general

Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44

System Information

Name:	Muse	Ipv4:	192.168.160.129
Model:	MU-1300	Hostname:	device-aaec44
Version:	1.0.51	Manufacturer:	Harman
Build Time:	Jul 7, 2023, 10:32:32 PM	Location:	
Open Source Licenses:	Click Here	Serial Number:	15500110005

Find **mojo-duet** in the list of available extensions:


https://192.168.160.129/#/system/extensions

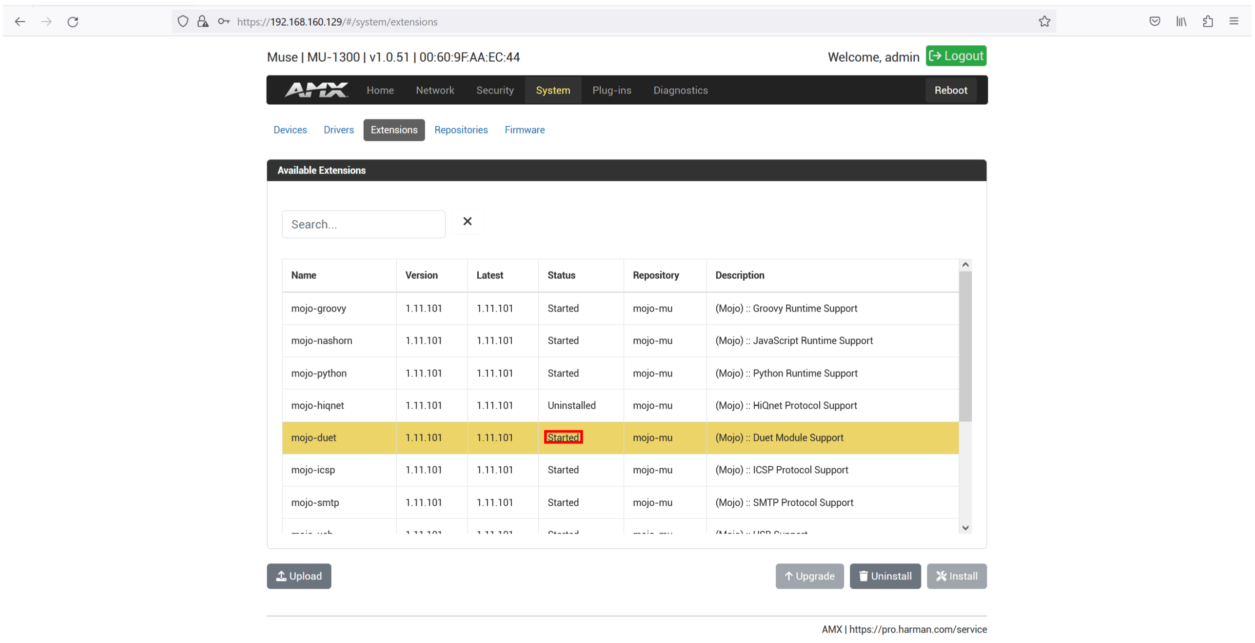
Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44

Available Extensions

Name	Version	Latest	Status	Repository	Description
mojo-groovy	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Groovy Runtime Support
mojo-nashorn	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: JavaScript Runtime Support
mojo-python	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Python Runtime Support
mojo-hiqnet	1.11.101	1.11.101	Uninstalled	mojo-mu	(Mojo) :: HiQnet Protocol Support
mojo-duet	1.11.101	1.11.101	Uninstalled	mojo-mu	(Mojo) :: Duet Module Support
mojo-icsp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: ICSP Protocol Support
mojo-smtp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: SMTP Protocol Support

We can see that the Extension's status is 'uninstalled'. With **mojo-duet** selected, click **Install**

After a few moments, the 'busy' screen will disappear, and the **mojo-duet** extension is enabled.



Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44

Welcome, admin

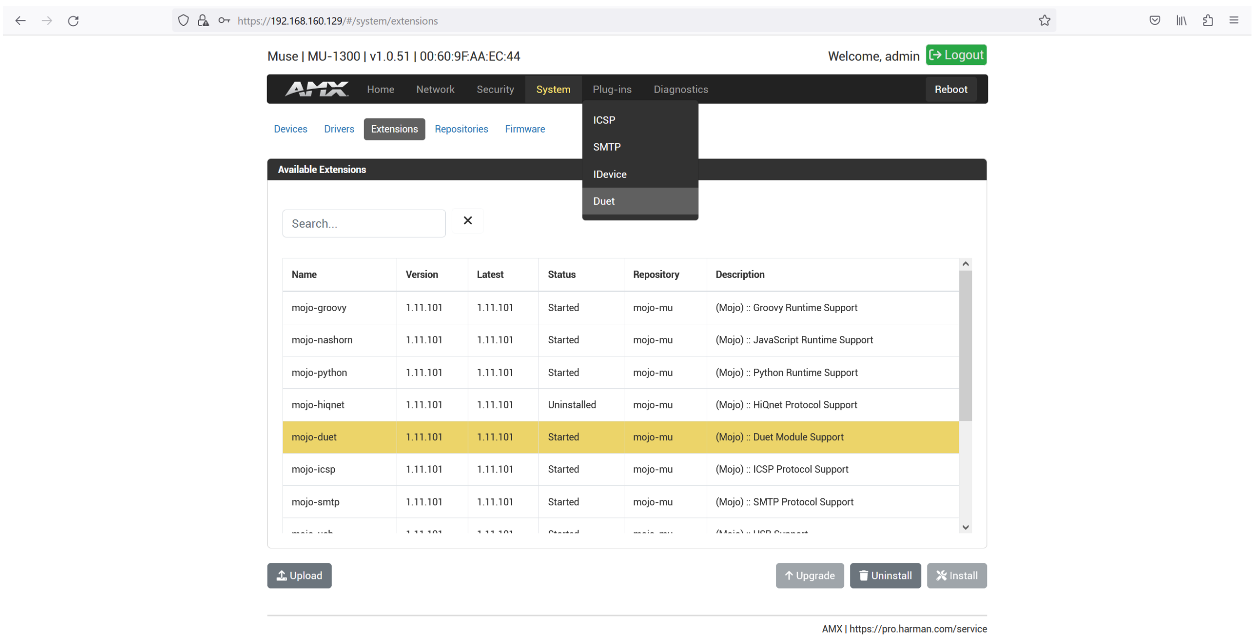
Available Extensions

Name	Version	Latest	Status	Repository	Description
mojo-groovy	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Groovy Runtime Support
mojo-nashorn	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: JavaScript Runtime Support
mojo-python	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Python Runtime Support
mojo-hiqnet	1.11.101	1.11.101	Uninstalled	mojo-mu	(Mojo) :: HiQnet Protocol Support
mojo-duet	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Duet Module Support
mojo-icsp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: ICSP Protocol Support
mojo-smtp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: SMTP Protocol Support

AMX | https://pro.harman.com/service

Loading a Duet module’s .jar file

Choose Plug-ins -> Duet in the MUSE controller’s web configuration menu bar



Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44

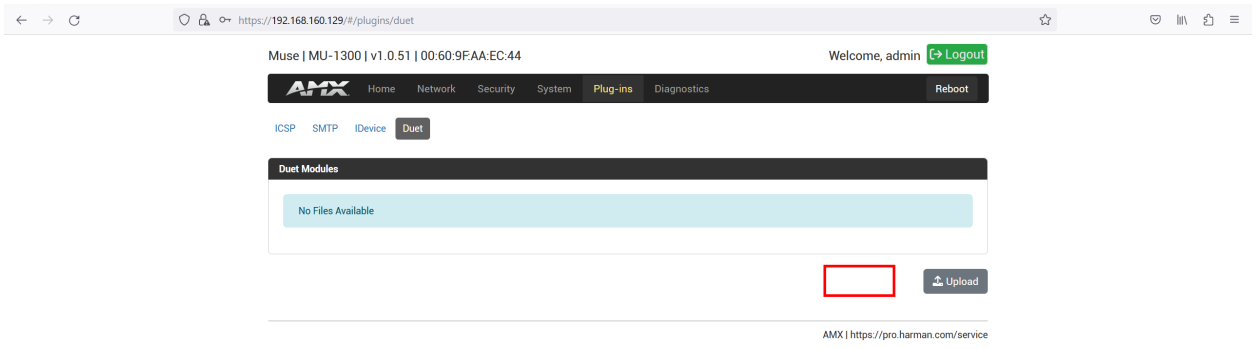
Welcome, admin

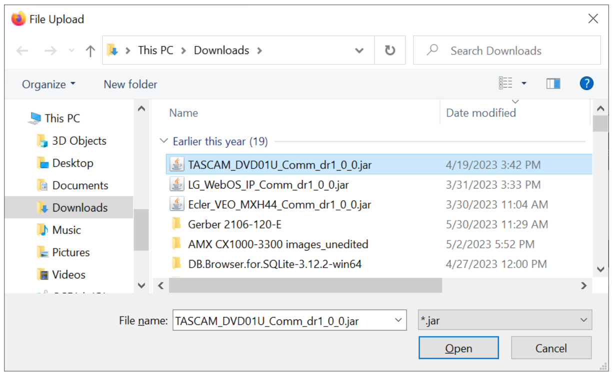
Available Extensions

Name	Version	Latest	Status	Repository	Description
mojo-groovy	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Groovy Runtime Support
mojo-nashorn	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: JavaScript Runtime Support
mojo-python	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Python Runtime Support
mojo-hiqnet	1.11.101	1.11.101	Uninstalled	mojo-mu	(Mojo) :: HiQnet Protocol Support
mojo-duet	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: Duet Module Support
mojo-icsp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: ICSP Protocol Support
mojo-smtp	1.11.101	1.11.101	Started	mojo-mu	(Mojo) :: SMTP Protocol Support

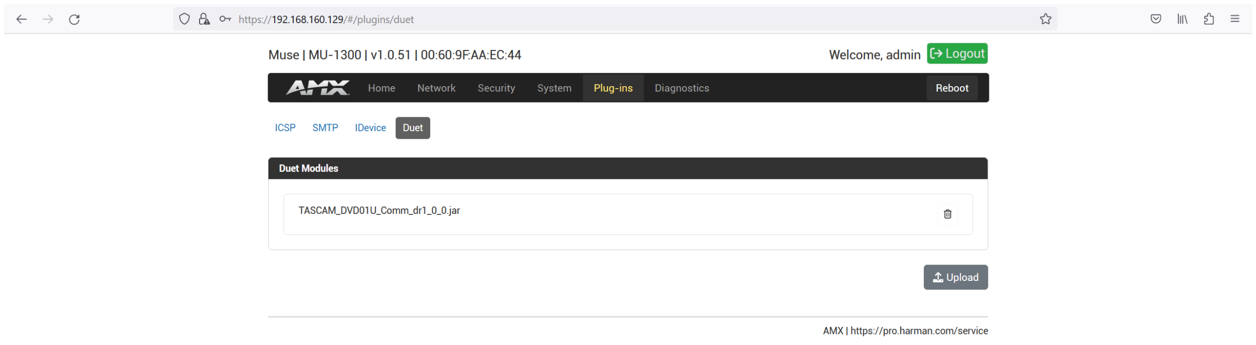
AMX | https://pro.harman.com/service

Click Upload to find and load a Duet .jar file



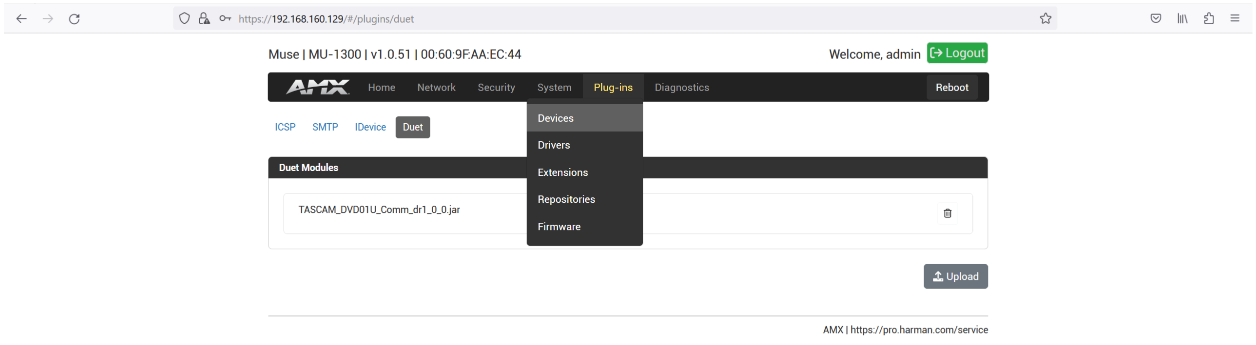

File Upload
This PC > Downloads
Search Downloads
Organize
New folder
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Name
Date modified
Earlier this year (19)
TASCAM_DVD01U_Comm_dr1_0_0.jar 4/19/2023 3:42 PM
LG_WebOS_IP_Comm_dr1_0_0.jar 3/31/2023 3:33 PM
Ecler_VEO_MXH44_Comm_dr1_0_0.jar 3/30/2023 11:04 AM
Gerber 2106-120-E 5/30/2023 11:29 AM
AMX CX1000-3300 images_unedited 5/2/2023 5:52 PM
DB.Browser.for.SQLite-3.12.2-win64 4/27/2023 12:00 PM
File name: TASCAM_DVD01U_Comm_dr1_0_0.jar
*.jar
Open
Cancel

The Duet Driver is now available for use

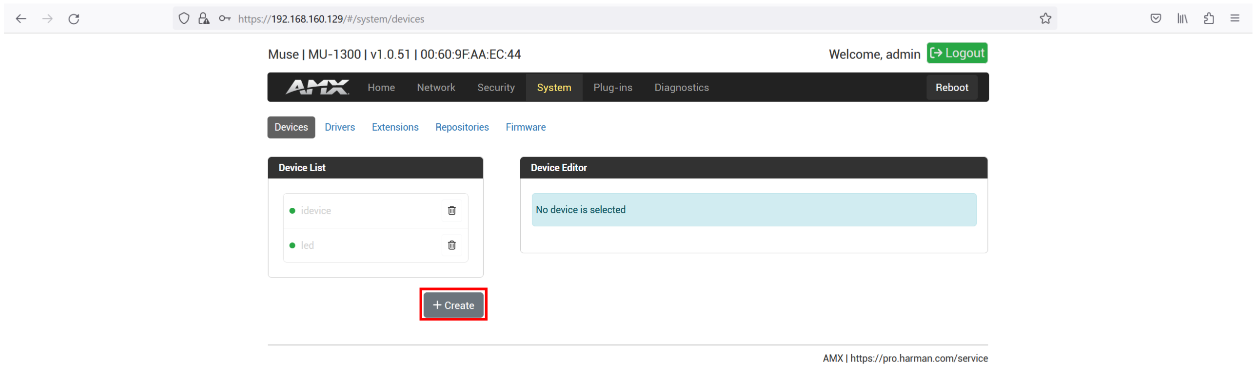

https://192.168.160.129/#/plugins/duet
Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44
Welcome, admin
Logout
Home
Network
Security
System
Plug-ins
Diagnostics
Reboot
ICSP
SMTP
IDevice
Duet
Duet Modules
TASCAM_DVD01U_Comm_dr1_0_0.jar
Upload
AMX | https://pro.harman.com/service

Make a Driver instance

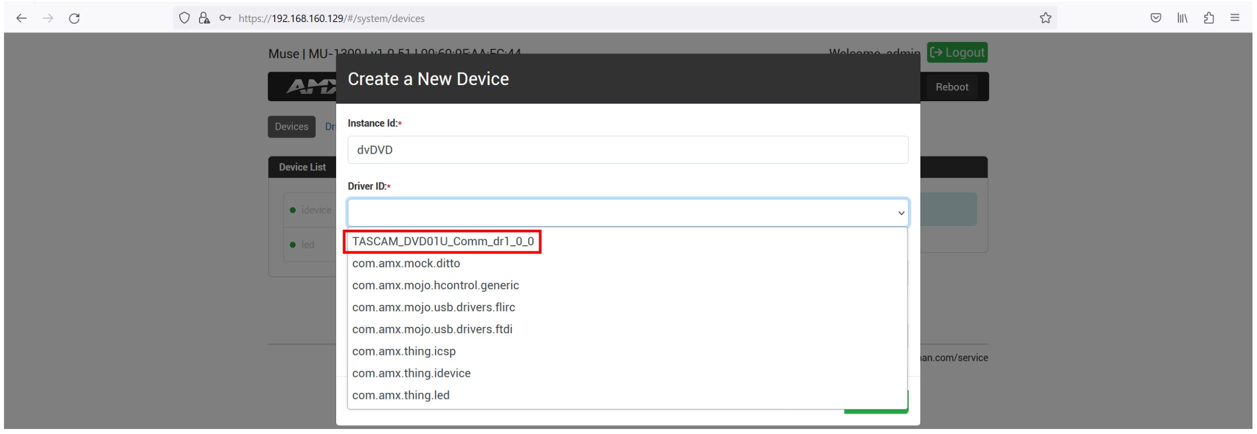
Choose System -> Devices to navigate to the Devices page



On the Devices page, click the Create button



In the resulting dialog box, name the Device and choose the Duet Module to use:


Create a New Device

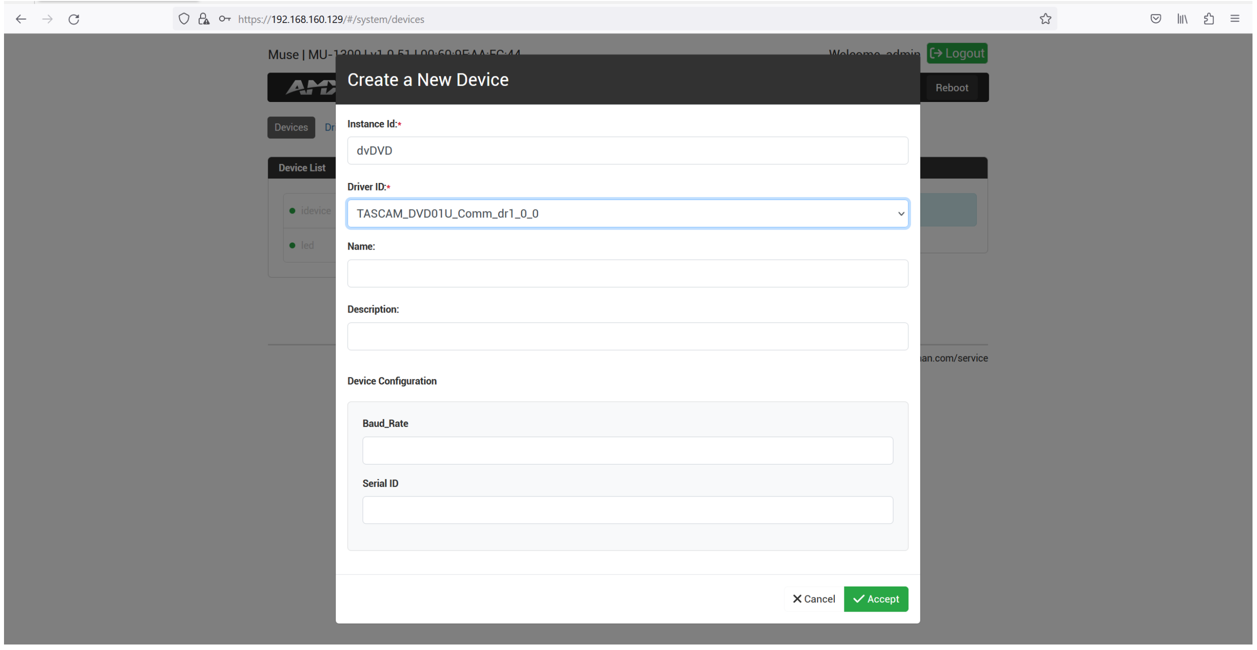
Instance Id:*

dvDVD

Driver ID:*

TASCAM_DVD01U_Comm_dr1_0_0
com.amx.mock.ditto
com.amx.mojo.hcontrol.generic
com.amx.mojo.usb.drivers.flirc
com.amx.mojo.usb.drivers.ftdi
com.amx.thing.icsp
com.amx.thing.idevice
com.amx.thing.led

Once selected, the dialog box will now contain new fields relevant to the Duet Module:


Create a New Device

Instance Id:*

dvDVD

Driver ID:*

TASCAM_DVD01U_Comm_dr1_0_0

Name:

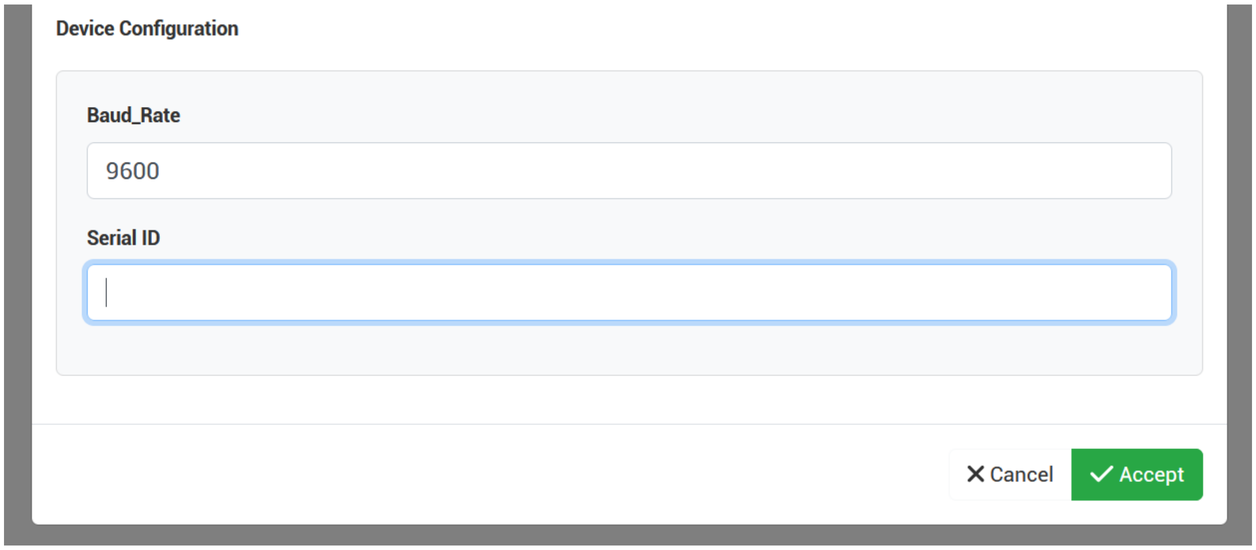
Description:

Device Configuration

Baud_Rate

Serial ID

Fill out the fields in the Device Configuration section to inform the Duet module about the setup



Device Configuration

Baud_Rate: 9600

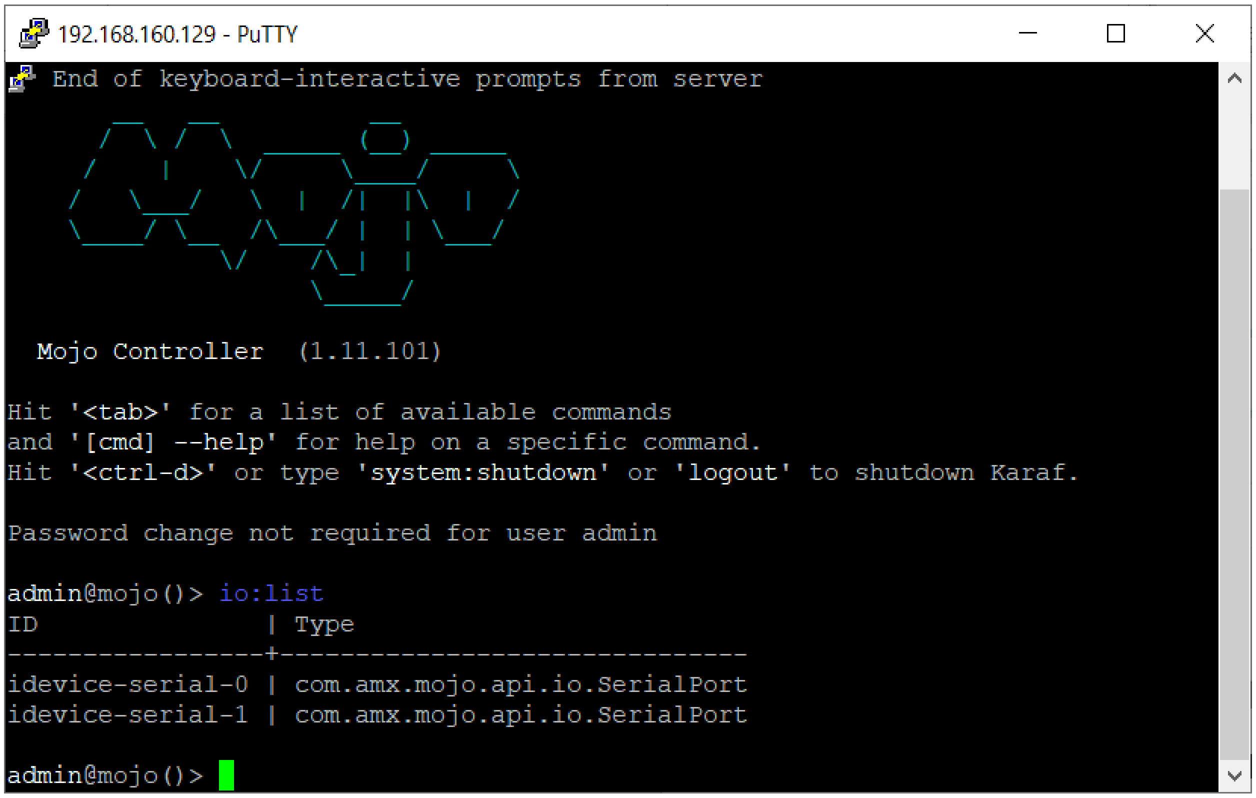
Serial ID:

For the Serial ID, the MUSE controller has a list of all available COM ports on the controller. In future, this will be populated in a pulldown. Until that time, we need to ask the MUSE controller for that list.

In an SSH session with the controller, log in and type the following command:

```
io:list
```

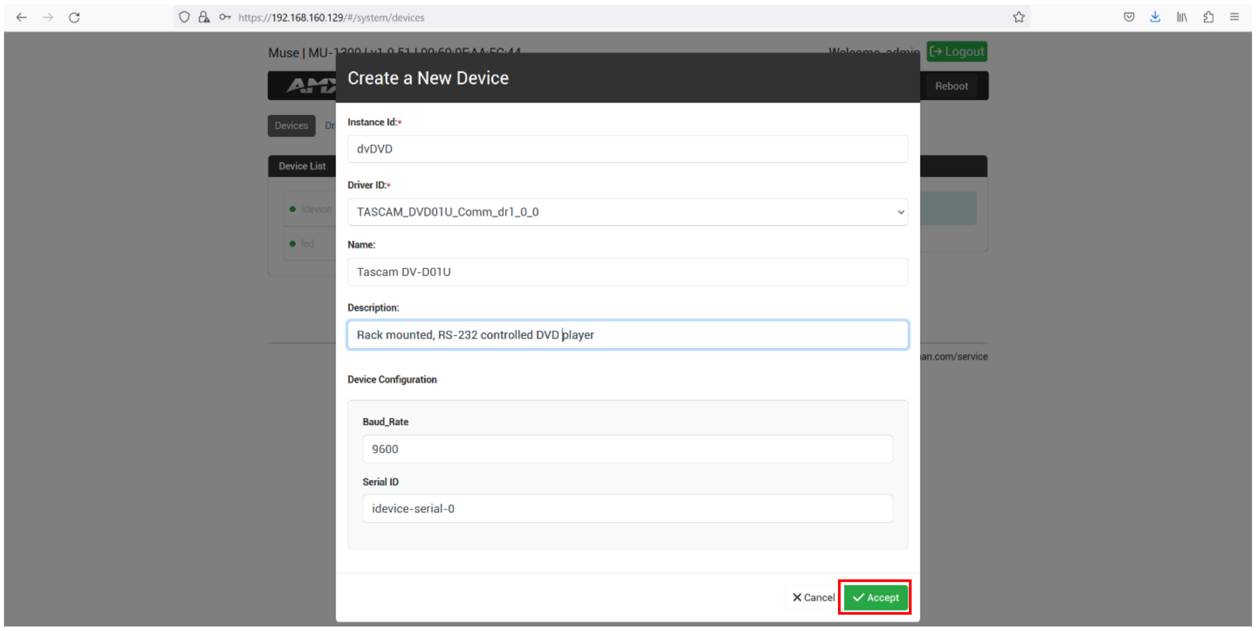
From this list, choose the COM port that the device is physically wired to.



```
End of keyboard-interactive prompts from server
     __   __          __
    /  \ /  \  _____ (__) _____
   /    |    \/     \_____/     \
  /  \___/    \  |  /|  |\  |  /
  \_____/ \__ /\__/ |  | \___/
           \/    /\_|  |
                 \_____/

  Mojo Controller  (1.11.101)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

Password change not required for user admin

admin@mojo()> io:list
ID                | Type
------------------+------------------------------
idevice-serial-0  | com.amx.mojo.api.io.SerialPort
idevice-serial-1  | com.amx.mojo.api.io.SerialPort

admin@mojo()>
```

Note: Mojo keeps the COM ports organized in an array. Consequently, the list is numbered in a zero-based index as most modern programming environments do. This differs from the actual silkscreen on the MUSE unit, which is one-based. In the example above, an MU-1300 is being used. There are two serial ports available on the controller. To use Serial Port 1 as labelled on the controller, choose:

```
 idevice-serial-0
```


Create a New Device

Instance Id:* dvDVD

Driver ID:* TASCAM_DVD01U_Comm_dr1_0_0

Name: Tascam DV-D01U

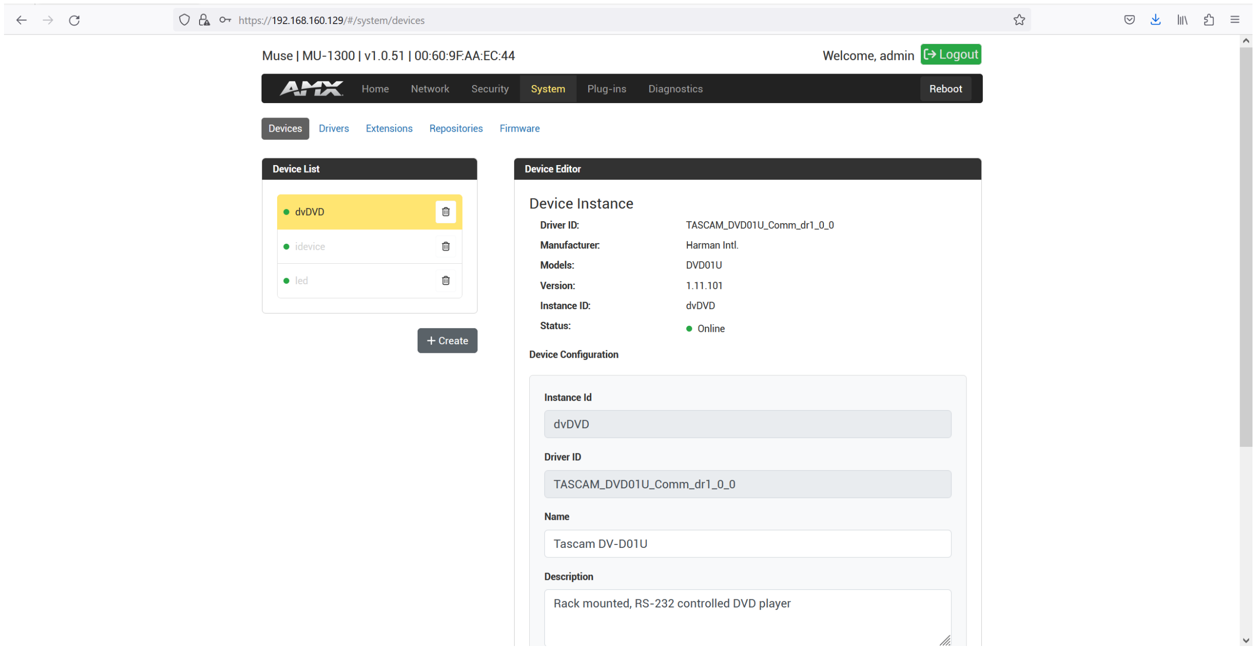
Description: Rack mounted, RS-232 controlled DVD player

Device Configuration

Baud_Rate: 9600

Serial ID: idevice-serial-0

Click Accept to bind the driver to the COM port.


Muse | MU-1300 | v1.0.51 | 00:60:9F:AA:EC:44

Device List: dvDVD, idevice, led

Device Editor

Device Instance

Driver ID: TASCAM_DVD01U_Comm_dr1_0_0

Manufacturer: Harman Intl.

Models: DVD01U

Version: 1.11.101

Instance ID: dvDVD

Status: Online

Device Configuration

Instance Id: dvDVD

Driver ID: TASCAM_DVD01U_Comm_dr1_0_0

Name: Tascam DV-D01U

Description: Rack mounted, RS-232 controlled DVD player

Using a module instance

Once the module instance is created, the module itself will start. This may include connecting to the device, polling, and other startup behaviors. The module instance will be listed by name in the Device List portion of the MUSE Controller's web server and the device:list CLI command:

```
admin@mojo()> device:list
ID       | Scope     | Status | Driver ID                   | Version
---------+-----------+--------+-----------------------------+--------
dvDVD    | <GLOBAL> | Online | TASCAM_DVD01U_Comm_dr1_0_0 | 1.12.12
dvTP     | <GLOBAL> | Online | com.amx.thing.icsp          |
idevice  | <GLOBAL> | Online | com.amx.thing.idevice       |
led      | <GLOBAL> | Online | com.amx.thing.led           |
admin@mojo()>
```

The CLI provides some focused insight to the control of the device through the module. Two helpful CLI commands are: device:commands and device:parameters.   The essential difference between commands and parameters is persistence. Parameter values stay and can be queried. Commands happen and then are gone.

Asking the CLI what command our DVD player has results in this output:

```
  admin@mojo()> device:commands dvDVD
--------------------------------------------------
DEVICE COMMANDS (dvDVD)
--------------------------------------------------
> "discDevice/0/cycleDiscTray"
> "discDevice/0/cycleRepeat"
> "discDevice/0/queryDiscProperties"
> "discDevice/0/queryDiscProperty" (1 arguments)
> "discDevice/0/queryTitleProperties"
> "discDevice/0/queryTitleProperty" (1 arguments)
> "discDevice/0/setPlayPosition" (2 arguments)
> "discTransport/0/cycleScanSpeed"
> "discTransport/0/getTrackInfo"
> "discTransport/0/queryTrackProperties"
> "discTransport/0/queryTrackProperty" (1 arguments)
> "menu/0/moveCursor" (1 arguments)
> "menu/0/pressButton" (1 arguments)
> "menu/0/selectItem"
> "module/channel" (3 arguments)
> "module/command" (2 arguments)
> "module/getInstanceProperty" (1 arguments)
> "module/intLevel" (3 arguments)
> "module/passThru" (1 arguments)
> "module/reinitialize"
> "module/setInstanceProperty" (2 arguments)
> "power/0/cycle"
--------------------------------------------------

Use device:invoke to execute a command

admin@mojo()>
```

The commands can be executed through the CLI as stated by this output.

For syntax help with this or any other CLI command, use the **man** or **help** CLI command:

```
admin@mojo()> man device:invoke
DESCRIPTION
        device:invoke

    Invoke a command on a specific runtime device.
```

```
 Arguments are presented as name/value pairs separated by '='.  e.g. day=tuesday time=12:00 where
day and time are the name of the arguments

SYNTAX
        device:invoke id command [arguments]

ARGUMENTS
        id
                Instance ID
                (required)
        command
                Command Name
                (required)
        arguments
                Command Arguments (name=value format)

admin@mojo()>
```

So, to toggle the power on this DVD Player, we can say:

```
admin@mojo()> device:invoke dvDVD power/0/cycle
--------------------------------------------------
DEVICE INVOKE COMMAND
--------------------------------------------------
 INSTANCE  : dvDVD
 COMMAND   : power/0/cycle
--------------------------------------------------
Invoked Successfully!


admin@mojo()>
```

For parameters, the device:parameters command lists both the available parameters and their current value.

```
admin@mojo()> device:parameters dvDVD
--------------------------------------------------
DEVICE PARAMETERS (dvDVD)
--------------------------------------------------
> "configuration/device/classname" = com.amx.thing.duet
> "configuration/device/container" =
> "configuration/device/description" = This is a Duet module driver
> "configuration/device/descriptorlocation" = descriptor.json
> "configuration/device/devicestate" = Running
> "configuration/device/manufacturer" = Harman Intl.
> "configuration/device/name" = Duet Driver
> "configuration/device/protocolversion" =
> "configuration/device/serialnumber" =
> "configuration/device/softwareversion" =
> "configuration/device/venue" =
> "configuration/device/version" =
> "discDevice/0/discCapacity" = 1
> "discDevice/0/titleCounterNotification" = false
> "discTransport/0/discTransport" = PLAY
> "discTransport/0/trackCounterNotification" = false
> "module/dataInitialized" = true
> "module/debugState" = 2
> "module/deviceDateTime" = Thu Jan 01 00:00:00 UTC 1970
> "module/deviceOnline" = true
> "module/fwVersion" = 1.0.0
> "module/passBack" = false
> "module/version" = 1.0.1
> "power/0/power" = ON
--------------------------------------------------

admin@mojo()>
```

Parameter values can be set and get individually with the device:set and device:get CLI commands:

```
admin@mojo()> device:set dvDVD discTransport/0/discTransport PLAY
--------------------------------------------------
DEVICE PARAMETER SET
--------------------------------------------------
 INSTANCE : dvDVD
 PARAMETER : discTransport/0/discTransport
--------------------------------------------------
NEW PARAMETER VALUE: PLAY (1)


admin@mojo()>
```

This particular parameter has caused the DVD to start playing. For a full list of values, we can consult the descriptor file for dvDVD. Typing driver:describe dvDVD gives the entire control set for the dvDVD device. The discTransport parameter specifically contains:

```
+- ARRAYS (4)
|  +-- discTransport
|  |    +- ELEMENTS (1)
|  |       +-- [0]
|  |           +- PARAMETERS (2)
|  |           |  +-- discTransport <enum> (rw)
|  |           |  |   +- [ <enum> [ INVALID, PLAY, STOP, PAUSE, PREVIOUS, NEXT, SCAN_FWD,
SCAN_REV, SLOW_FWD ]
```

If we stopped the DVD player and wanted to query the discTransport parameter, we can type:

```
admin@mojo()> device:get dvDVD discTransport/0/discTransport
--------------------------------------------------
DEVICE PARAMETER/METADATA GET
--------------------------------------------------
```

```
 INSTANCE           : dvDVD
 PARAMETER/METADATA : discTransport/0/discTransport
--------------------------------------------------
PARAMETER VALUE: STOP (2)


admin@mojo()>
```

These commands and parameters give us insight into how to use our device in control scripts. Duet modules have the concept of Components (typically arrays of Components) and methods/properties.   For the above example, the discTransport component array has one element (index 0).   This component has a property called discTransport, which is called to set or get the current state of the DVD.

Groovy syntax example:

```
      dvDVD.discTransport[0].discTransport = "PLAY"
```

Python syntax example:

```
        dvDVD.discTransport[0].discTransport = "PLAY"
```

JavaScript syntax example

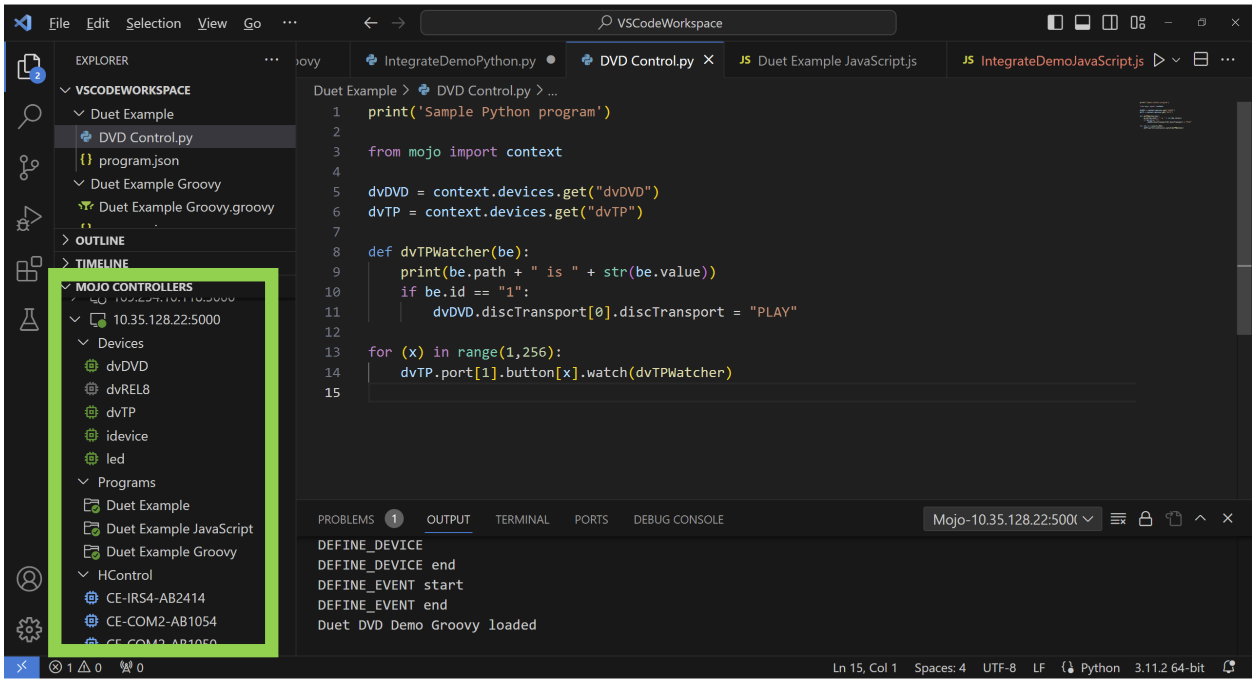
```
        dvDVD.discTransport[0].discTransport = "PLAY";
```

…as before, the individual bits of Mojo-related syntax barely differ between language.

Debugging

There are two places that contain substantial debugging information for MUSE. VS Code and the CLI.

VS Code



```python
print('Sample Python program')

from mojo import context

dvDVD = context.devices.get("dvDVD")
dvTP = context.devices.get("dvTP")

def dvTPWatcher(be):
    print(be.path + " is " + str(be.value))
    if be.id == "1":
        dvDVD.discTransport[0].discTransport = "PLAY"

for (x) in range(1,256):
    dvTP.port[1].button[x].watch(dvTPWatcher)
```

```
DEFINE_DEVICE
DEFINE_DEVICE end
DEFINE_EVENT start
DEFINE_EVENT end
Duet DVD Demo Groovy loaded
```

A great deal of information can be obtained from the MOJO CONTROLLER section that was added by the Mojo VS Code extension.

We can see that we are connected to and authenticated into a controller at 10.35.128.22. The green pip indicates the state of the connection.   A yellow pip indicates the controller has been seen on the network, but authentication has not taken place. No pip means the status is unknown. To attempt a connection, use the refresh icon to initiate it. Follow any login prompts or messages that appear.

Once connected, we can see:

- Currently configured devices and their status
- Currently running programs
- HControl devices discovered near the controller

From the Devices portion of the tree, we can remove devices or ask for the descriptor. Each of these activities is available by context menu by right-clicking on the device in question

From the Programs portion of the tree, you can:

- Retrieve the script from the controller
- Restart the script
- Delete the script
- Enable/disable the script
- Attach to / Detach from the standard output of a script (print( ) and log( ) statements go here)

Attaching is the primary means of debugging a particular script. The script author can place print( ) or log( ) statements throughout the code to track the progress and expected code flow. These print statements can contain information about variable states and other helpful information.

Right-click on the running script and choose Attach to see the standard output.

Groovy syntax example:

```
println('DEFINE_DEVICE')

dvTP = context.devices.get("dvTP");
dvDVD = context.devices.get("dvDVD")

println('DEFINE_DEVICE end')

println('DEFINE_EVENT start')

dvTP.port[1].button[17].watch( {
    if(it.value) { //PUSH:
        dvDVD.discTransport[0].discTransport = "PLAY"
    }
    })

dvTP.port[1].button[18].watch( {
    if(it.value) { //PUSH:
        dvDVD.discTransport[0].discTransport = "STOP";
    }
    })

println("Duet DVD Demo Groovy loaded")
```

…would produce the following output when restarted:

```
Program "Duet Example Groovy" attached, you will see outputs here.
DEFINE_DEVICE
DEFINE_DEVICE end
DEFINE_EVENT start
Duet DVD Demo Groovy loaded
```

Python syntax example:

```
print('Duet Example Python')
from mojo import context

print('Grab device objects')
dvDVD = context.devices.get("dvDVD")
dvTP = context.devices.get("dvTP")

print('Define my button callback')
def dvTPWatcher(be):
    print(be.path + " is " + str(be.value))
    if be.id == "1":
        dvDVD.discTransport[0].discTransport = "PLAY"

print('set a watch for my buttons')
```

```
for (x) in range(1,256):
    dvTP.port[1].button[x].watch(dvTPWatcher)

print('Duet Example Python script loaded')
```

…would produce the following output when run.

```
Duet Example Python
Grab device objects
Define my button callback
set a watch for my buttons
Duet Example Python script loaded
```

JavaScript syntax example

```
context.log('JavaScriptDemo START')

context.log('DEFINE_VARIABLE')

var dvTP;
var dvDVD;

function dvTP_button_event(event) {
    context.log('dvTP_button_event(event) called')
    context.log('event:'+event.path+" is "+event.newValue)

    if(event.newValue){ // PUSH:
        var button
        button = parseInt(event.id)
        switch(button) {
            case 9:
              dvDVD.discTransport[0].discTransport = "PLAY";
              break;
            case 10:
                dvDVD.discTransport[0].discTransport = "STOP";
              break;
            default:
              console.log("unhandled button event "+event.id)
          }

    }
}
context.log('DEFINE_DEVICE')

dvTP   = context.devices.get("dvTP")
dvDVD = context.devices.get("dvDVD")

context.log('DEFINE_EVENT')

for(let i=1;i<=255;i++)
```

```
{
   // set watches on all dvTP buttons
   dvTP.port[1].button[i].watch(dvTP_button_event);
}

context.log('Duet Example JavaScript loaded')
```

would produce the following output:

```
Program "Duet Example JavaScript" attached, you will see outputs here.
[INFO] JavaScriptDemo START
[INFO] DEFINE_VARIABLE
[INFO] DEFINE_DEVICE
[INFO] DEFINE_EVENT
[INFO] Duet Example JavaScript loaded
```

The CLI

A far more powerful interface for debugging is the CLI. This is available from SSH, the Virtual COM port available on the USB-C connector, and the Diagnostics → Shell portion of the MUSE web server.

As discussed earlier in this document, device parameters can be retrieved and set, commands sent, devices listed, and dozens of other useful functions performed.

For the entire list of commands, type *help* in the CLI. This will display a document containing all the published commands. Hit CTRL-C to exit this document.

For details on a specific command, use the *man or help* command. For example:

```
admin@mojo()> man network:ip
DESCRIPTION
        network:ip

    Set and get network config

SYNTAX
        network:ip [options]

OPTIONS
        -e, --ethinterface
                Set ethernet interface specifier [eth0 or eth1] (defaults to eth0)
                (defaults to eth0)
        --help
                Display this help message
        -m, --mode
                Set mode: dhcp: static: ipv6d: ipv6s. {DHCP | Static | Dynamic Ipv6 or Static Ipv6
[Eth1 can only be static] (defaults to empty)

                (defaults to )
        -s, --subnet
                Set subnet mask [Valid only when mode is static] (defaults to empty)

                (defaults to )
        -g, --gateway
                Set gateway [Valid only when mode is static] (defaults to empty)

                (defaults to )
        -i, --ip
                Set IP address [Valid only when mode is static] (defaults to empty)

                (defaults to )
        -en, --enable
                Enable eth1 [When enabling eth1 ip, subnet, and gateway values must be entered.]

        -d, --disable
                Disable eth1
```

```
admin@mojo()>
```

…displays all the syntax and option information for retrieving network information about the MUSE controller.

For debugging, the most useful command in the CLI is *log:tail* .   This will show any exceptions that occur, any debug information that has been printed to the log, and other real-time data that is useful.

With our Tascam DVD module running, log:tail will show transactions to and from the serial port attached to the DVD. This behavior is defined by the Duet module itself.

```
00:11:10.169 WARN  | TimerDaemon | Duet:TASCAMDVD01UDiscDevice | 190 |  TASCAMDVD01UDiscDevice: Comm ----------> device: BUFFER '023E504C5963465744202020202020313703'
00:11:10.262 WARN  | Thread-61 | Duet:TASCAMDVD01UDiscDevice | 190 |  TASCAMDVD01UDiscDevice: Comm <---------- device: INCOMING DATA '023E504C5973465744202020202020323703'
```

In this case, we can see a transaction that occurred when the module decided to poll the DVD player for status. In this case, the module author decided to print the string, which is a mix of printable and non-printable ASCII characters, as an ASCII stream of the hexadecimal values of each character. Knowing this, we can see that the conversation was:

Controller to DVD: STX>PLYcFWD   17ETX   (please start playing the disc)

DVD to Controller: STX >PLYsFWD   27 ETX   (play command acknowledged)

Appendix A – The Thing API

The Mojo Universal Scripting Engine (MUSE) contains a framework for communicating with AMX devices, modules, and other AMX specific facilities: the Thing API.

Context is the handle for the Mojo engine. It has logging and device access features. The structure of context is:

```
context
•  devices
   •  get(name)      – get a specific device by its name
   •  has(name)      – check if a specific device is defined
   •  ids()          – get the list of defined devices
•  log
   •  level          – set/get the current logging threshold
   •  trace(msg)   – issue a log message at TRACE level
   •  debug(msg)   – issue a log message at DEBUG level
   •  info(msg)    – issue a log message at INFO level
   •  warning(msg) – issue a log message at WARNING level
   •  error(msg)   – issue a log message at ERROR level
•  services
   •  get(serviceName) – get a service by name
•  run(globals())  - a call specific to Python to continue operation once
         the script has reached its end
```

Syntax Examples: (identical for each language unless noted)

Get the device named AMX-10001 and assign it to the script variable dvTP:

```
dvTP = context.devices.get("AMX-10001")
```

Check if a device named dvLighting has been defined:

Python:

```
if context.devices.has("dvLighting"):
      # do something
```

Groovy/JavaScript:

```
if(context.device.has("dvLighting") {
      // do something
}
```

Send a message to the logger at the WARNING threshold:

```
context.log.warning("dvLighting is not defined")
```

Get Request an instance of the timeline service that triggers 1/second forever:

```
myTimeline = context.services.get("timeline")
```

```
mytimeline.start([1000],false,-1)
```

The services available vary based on what is currently installed/enabled.   At the time of this writing, there are three services: timeline, smtp, and netlinxClient.

For the syntax and usage of the services, use the following CLI command:

```
doc:list
```

…to obtain the current list of services available and:

```
doc:show serviceName
```

…to show the syntax and usage description.

The Event structure

Any callback or lambda functions for a .listen passed the event structure. This contains the specific information that triggered the event.

Field	Data type	Description
path	String	The property of the device that this event refers to
id	String	A shortened version path. For ICSP, only the button number is conveyed
arguments	array	The data payload of the event, dependent on the specific event
arguments	array	The data payload of the event, dependent on the specific event
oldValue	variant	The data value before the event was processed
source	Object Ref	The object reference for the specific parameter that was updated

The most commonly encountered event within MUSE is for receiving data from a serial port. Here are some language specific examples.

Python:

```
def dvCOM3_Event(ev):
    print('PY_ev.source    =='+str(ev.source))
    print('PY_ev.path      =='+str(ev.path))
    print('PY_ev.id        =='+str(ev.id))
    print('PY_ev.arguments:')
    print('[\'data\'] =='+str(ev.arguments['data'].decode()))

idevice.serial[3].receive.listen(dvCOM3_Event)
```

…has the following output when a serial message is received:

```
PY_ev.source    ==<mojo.PythonThing object at 0xffff956e9880>
PY_ev.path      ==serial/3/receive
PY_ev.id        ==receive
PY_ev.arguments:
['data'] ==hello from python
```

Groovy:

```
idevice.serial[3].receive.listen({ serialEvent ->
    println("serial data event on COM3")
    println('G_serialEvent.source    =='+serialEvent.source)
    println('G_serialEvent.path      =='+serialEvent.path)
    println("G_serialEvent.id        =="+serialEvent.id)
    println('G_ev.arguments:')
    println('data =='+new String(serialEvent.arguments.get("data"),"UTF-8"))

})
```

…has the following output when a serial message is received:

```
G_serialEvent.source    ==com.amx.mojo.groovy.GroovyThing@1b5a766a
G_serialEvent.path      ==serial/3/receive
G_serialEvent.id        ==receive
G_ev.arguments:
data ==hello from Groovy
```

JavaScript:

```
function dvCOM3_RX(serialEvent){
  context.log("serial data event on COM3")
  context.log('JS_serialEvent.source    =='+serialEvent.source)
  context.log('JS_serialEvent.path      =='+serialEvent.path)
  context.log("JS_serialEvent.id        =="+serialEvent.id)
  context.log('JS_ev.arguments:')
  context.log('data =='+String(serialEvent.arguments.data))
}

idevice.serial[3].receive.listen(dvCOM3_RX)
```

…has the following output when a serial message is received:

```
[INFO] serial data event on COM3
[INFO] JS_serialEvent.source    ==null
[INFO] JS_serialEvent.path      ==serial/3/receive
[INFO] JS_serialEvent.id        ==receive
[INFO] JS_ev.arguments:
[INFO] data ==hello from JavaScript
```

The Parameter Update Structure

Any callback or lambda functions for a .watch( ) attached to a parameter is passed this structure. This contains the specific information that triggered the parameter change event.

Field	Data type	Description

path	String	The specific parameter that has been updated
id	String	The last element of the path
value	variant	The current value of the parameter
newValue	variant	The current value of the parameter
oldValue	variant	The previous value of the parameter
normalized	float	A float value between 0 and 1, inclusive, based on the range of the value
source	Object Ref	The object reference for the specific parameter that was updated

Strangely the most commonly encountered parameter update within the MUSE environment is a button press. Button events can be queried, so are actually parameters.

Groovy:

```
println('Sample Groovy program')

dvTP = context.devices.get("dvTP")

dvTP.port[1].button[2].watch( { parameterChange ->
    println('parameterChange.id        =='+parameterChange.id)
    println('parameterChange.source    =='+parameterChange.source)
    println('parameterChange.path      =='+parameterChange.path)
    println('parameterChange.value     =='+parameterChange.value)
    println('parameterChange.newValue  =='+parameterChange.newValue)
    println('parameterChange.oldValue  =='+parameterChange.oldValue)
    println('parameterChange.normalized=='+parameterChange.normalized)
} )
```

…has the following output when button 2 on port 1 is pressed:

```
parameterChange.id        ==2
parameterChange.source    ==com.amx.mojo.groovy.GroovyThing@36a1b33c
parameterChange.path      ==port/1/button/2
parameterChange.value     ==true
parameterChange.newValue  ==true
parameterChange.oldValue  ==false
parameterChange.normalized==1.0
```

Python:
```
print('Sample Python program')

from mojo import context

dvTP = context.devices.get("dvTP")

def dvTPWatcher(parameterChange):
    print(parameterChange.path + " is " + str(parameterChange.value))
    print('parameterChange.id        =='+parameterChange.id)
    print('parameterChange.source    =='+str(parameterChange.source))
    print('parameterChange.path      =='+str(parameterChange.path))
    print('parameterChange.value     =='+str(parameterChange.value))
    print('parameterChange.newValue  =='+str(parameterChange.newValue))
    print('parameterChange.oldValue  =='+str(parameterChange.oldValue))
    print('parameterChange.normalized=='+str(parameterChange.normalized))

dvTP.port[1].button[3].watch(dvTPWatcher)

context.run(globals())
```

…has the following output when button 3 on port 1 is pressed:
```
port/1/button/3 is True
parameterChange.id        ==3
parameterChange.source    ==<mojo.PythonThing object at 0xffffb1acc970>
parameterChange.path      ==port/1/button/3
parameterChange.value     ==True
parameterChange.newValue  ==True
parameterChange.oldValue  ==False
parameterChange.normalized==1.0
```

JavaScript:

```
function dvTP_button_change(buttonChange) {
   context.log('dvTP_button_change(buttonChange) called')
   context.log('button:'+buttonChange.path+" is "+ buttonChange.newValue)
   context.log("JS_buttonChange.id        =="+ buttonChange.id)
   context.log('JS_buttonChange.source    =='+ buttonChange.source)
   context.log('JS_buttonChange.path      =='+ buttonChange.path)
   context.log('JS_buttonChange.value     =='+ buttonChange.value)
   context.log('JS_buttonChange.newValue  =='+ buttonChange.newValue)
   context.log('JS_buttonChange.oldValue  =='+ buttonChange.oldValue)
   context.log('JS_buttonChange.normalized=='+ buttonChange.normalized)


idevice.serial[3].receive.listen(dvCOM3_RX)
```

…has the following output when button 1 on port 1 is pressed:

```
[[INFO] dvTP_button_change(buttonChange) called

[INFO] button:port/1/button/1 is true
[INFO] JS_buttonChange.id        ==1
[INFO] JS_buttonChange.source    ==true
[INFO] JS_buttonChange.path      ==port/1/button/1
[INFO] JS_buttonChange.value     ==true
[INFO] JS_buttonChange.newValue  ==true
[INFO] JS_buttonChange.oldValue  ==false
[INFO] JS_buttonChange.normalized==1
```

The Parameter structure – set/get

When manipulating the value of a particular Thing parameter

Field	Data type	Description
value	variant	The new value which cause the parameter change callback
normalized	float	A float value between 0 and 1 inclusive, based on the value’s range
min	variant	The minimum value of a numerical parameter
max	variant	The maximum value of a numerical parameter
defaultvalue	variant	The default value of a parameter
type	float	The data type of this specific parameter
enums	enumeration	For an enumeration, the specific data points available

Syntax Examples:

In Python, to retrieve all the properties of the parity setting on COM3:

```
    print("parity value:"       + str(idevice.serial[3].parity.value))
    print("parity normalized:"  + str(idevice.serial[3].parity.normalized))
    print("parity min:"         + str(idevice.serial[3].parity.min))
    print("parity max:"         + str(idevice.serial[3].parity.max))
    print("parity defaultvalue:"+ str(idevice.serial[3].parity.defaultvalue))
    print("parity type:"        + str(idevice.serial[3].parity.type))
    print("parity enums:"       + str(idevice.serial[3].parity.enums))
```

Returns:

```
parity value:NONE
parity normalized:0.0
parity min:0
parity max:4
parity defaultvalue:NONE
parity type:enum
parity enums:['NONE', 'EVEN', 'ODD', 'MARK', 'SPACE']
```

In JavaScript, to retrieve all the properties of the parity setting on COM3:

```
    context.log("parity value:"       + idevice.serial[3].parity.value)
    context.log("parity normalized:"  + idevice.serial[3].parity.normalized)
    context.log("parity min:"         + idevice.serial[3].parity.min)
    context.log("parity max:"         + idevice.serial[3].parity.max)
    context.log("parity defaultvalue:"+ idevice.serial[3].parity.defaultvalue)
    context.log("parity type:"        + idevice.serial[3].parity.type)
    context.log("parity enums:"       + idevice.serial[3].parity.enums)
```

Returns:

```
[INFO] parity value:NONE
[INFO] parity normalized:0
[INFO] parity min:0
[INFO] parity max:4
[INFO] parity defaultvalue:NONE
[INFO] parity type:enum
[INFO] parity enums:[NONE,EVEN,ODD,MARK,SPACE]
```

In Groovy, to retrieve all the properties of the parity setting on COM3:

```
        println("parity value:"       + idevice.serial[3].parity.value)
        println("parity normalized:"  + idevice.serial[3].parity.normalized)
        println("parity min:"         + idevice.serial[3].parity.min)
        println("parity max:"         + idevice.serial[3].parity.max)
        println("parity defaultvalue:"+ idevice.serial[3].parity.defaultvalue)
        println("parity type:"        + idevice.serial[3].parity.type)
        println("parity enums:"       + idevice.serial[3].parity.enums)
```

Returns:

```
parity value:NONE
parity normalized:0.0
parity min:0
parity max:4
parity defaultvalue:NONE
parity type:enum
parity enums:[NONE, EVEN, ODD, MARK, SPACE]
```