



LUA DEVICE DRIVER PROGRAMMING GUIDE

V1.0

1ST SEPTEMBER 2020

LUA DEVICE DRIVER PROGRAMMING GUIDE

CONTENTS

CONTENTS	2	LuaTCPServerMessageReceivedCallback	17
GENERAL OVERVIEW	4	LuaTCPClientMessageReceivedCallback	17
DEVICE DRIVER OVERVIEW	4	LuaUDPServerMessageReceivedCallback	18
Device Driver Types	4	LuaUIUpdatedCallback	18
Official Drivers	4	LuaTimerCallback	18
Custom Drivers	5	XTCPServerSetup	18
Components of a Device Driver	5	XTCPServerClose	19
Driver Basic Information	5	XTCPServerReplyMessage	19
Device Specific Parameters	5	XTCPClientSetup	19
Script Body	5	XTCPClientClose	19
UI Control Panels	5	XTCPClientIsConnected	20
INSTALLING DEVICE DRIVERS	6	XTCPClientSendMessage	20
Official Drivers	6	XUDPServerSetup	20
Custom Drivers	6	XUDPServerClose	20
LUA DEVICE DRIVER BUILDER	7	XUDPClientSendMessage	21
DEVICE DRIVER DEVELOPMENT	10	XUIUpdateRequest	21
Driver Basic Information	10	XStartTimer	21
Device Specific Parameters	12	XStopTimer	21
Driver Lua Scripting	14	XConsoleLog	22
LuaDriverInitialization	16	User Interface Builder	22
LuaTCPServerConnectionCreatedCallback	17	DRIVER TESTING AND DEBUGGING	23
LuaTCPServerConnectionClosedCallback	17	DRIVER IMPORTING AND EXPORTING	26
		PROJECT FILES AND DRIVERS	28

GENERAL OVERVIEW

The Lua Device Driver Builder software is integrated into Xilica Designer and provides a tool for equipment manufacturers, system integrators, distributors, and end-users to build control interfaces within Xilica Designer for any 3rd party device with Ethernet control. This tool provides two major capabilities: communication protocol handling, and control panel UI design and packaging. Drivers designed by manufacturers are available as an installable plug-in to the Xilica Designer software and allows for drag & drop access to a given device's controllable parameters.

The install package of a fully tested third-party developed driver may be made publicly available at www.xilica.com.

Installed drivers are shown in the Component Library of the Xilica Designer software. Double-clicking on the device will download and install the driver, and it will then be available to drag & drop into a project. A double-click on the device icon in the project window will reveal the module's control interface. Elements from an interface can be selected and dragged into an XTouch panel, XWP device, or Project Controller window.

In addition to official drivers, users can create their own local custom drivers, whether from scratch or by using an existing driver as a starting point. Such custom drivers can be exported and made available for sharing as well.

DEVICE DRIVER OVERVIEW

Each device driver should be created for a specific make and model of hardware. Multiple instances of the same driver can be utilized in a design and distinguished simply by assigning a unique IP address for each one.

Note: global variables within the script of a given module are specific to that instance of the module only and are not accessible to other instances of the same module.

Varying instances of the same module may be uniquely customized with device-specific parameters. For example, common parameters among multiple instances may be assigned unique device IDs.

Device Driver Types

Official Drivers

This type of driver is developed either by a third-party manufacturer or by Xilica, are officially supported and will be made available to all Xilica Designer users. When new drivers are made available to the public, Xilica Designer will detect the availability of new drivers during the application startup and will download the driver information automatically. New drivers will be listed in the Component Library within Xilica Designer.

Note: once new drivers are detected, a message will prompt the user to restart Xilica Designer after which the new driver will be made available for use.

Custom Drivers

In addition to official drivers, users can define their own custom drivers, either by creating a new driver from scratch or by using an existing driver as a starting point.

After a custom driver is developed, it can then be exported and then re-imported by other Xilica Designer users.

Components of a Device Driver

Each device driver contains the following four components:

Driver Basic Information

General information of a device driver such as name and description. Users can define what I/O points the device driver will have. These defined I/O points are mainly for documentation purposes, (when dragged into the project view, the device icon will include corresponding I/O points to document the wiring of the project). For Dante I/O points, users may choose to setup Dante flows in the project view, and the corresponding Dante routing will be setup automatically once online with the hardware.

Device Specific Parameters

As a single Driver definition can apply to multiple instances (multiple devices in the same project), each individual device might have different information associated with it. One good example would be the Login ID and Password. In order to properly handle different and potentially unique parameters of each individual instance of the same Driver, we support the definition of Device Specific parameters.

Script Body

The Script Body consists of the actual Lua script programming code. Different callback functions will need to be scripted so that when the driver starts, different callback functions will be called.

UI Control Panels

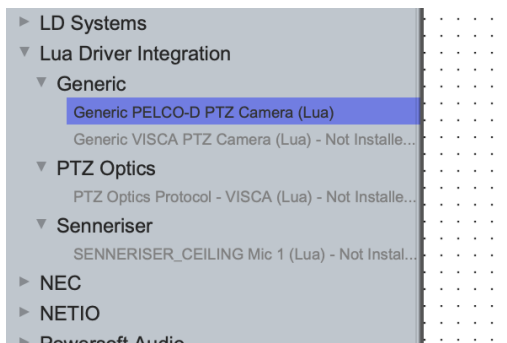
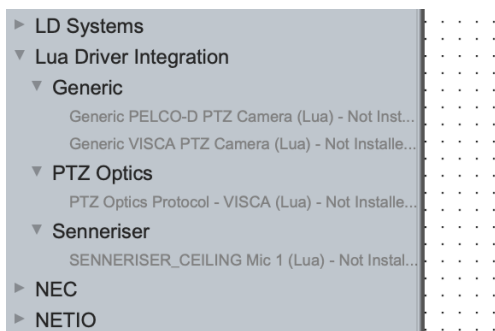
A drag and drop GUI builder (similar to XTouch panel interface design) enables the ability for develop the user interface of a driver. Various UI objects can be placed in multiple interface pages. Each UI object is named and can be accessed from the Lua script by name.

INSTALLING DEVICE DRIVERS

Official Drivers

When a new Official driver is detected by the Xilica Designer software, it will first load in the driver's basic information, including the name and description. At this stage the drivers are listed in the Component Library but will be greyed out to indicate that it is not yet installed. To install a driver, double-click it and it will automatically download and install, providing the machine is connected to the Internet. Once downloaded and installed the driver can now be brought into the project window.

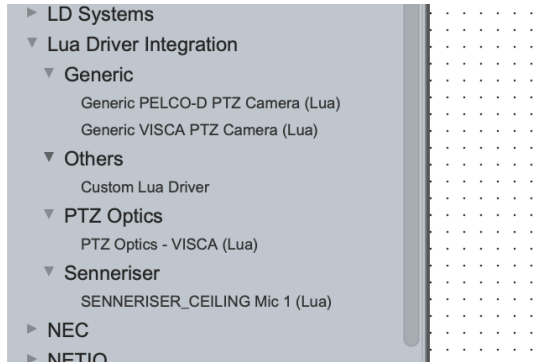
In the following example, a double-clicking on "Generic PELCO-D PTZ Camera (Lua)" will download and install the driver.



Custom Drivers

For user-defined Custom drivers (or drivers that have been imported), there is no need to re-install as they were developed locally. These drivers will be automatically listed in the Component Library.

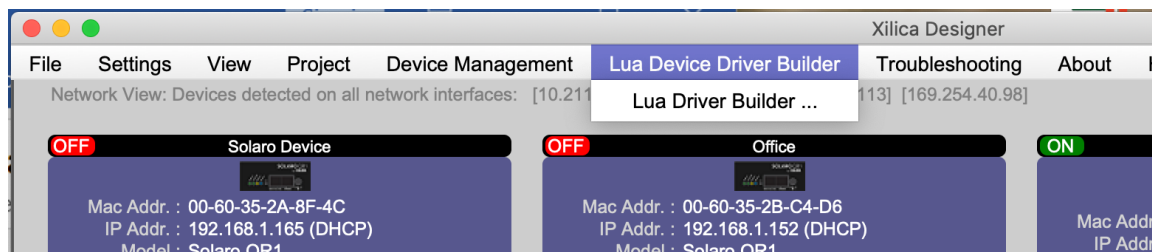
In the following example, the “Custom Lua Driver” device is automatically installed and available.



LUA DEVICE DRIVER BUILDER

A Lua Device Driver Builder is integrated into the Xilica Designer software for programmers to create new drivers or modify existing ones. To start the device driver builder, select the “Lua Device Driver Builder” -> “Lua Driver Builder ...” item from top menu bar. This will bring up the Lua Device Driver List Dialog.

Note: this option is only available when all projects are closed. You will need to save and exit any open projects before you can access the Lua Device Driver Builder.



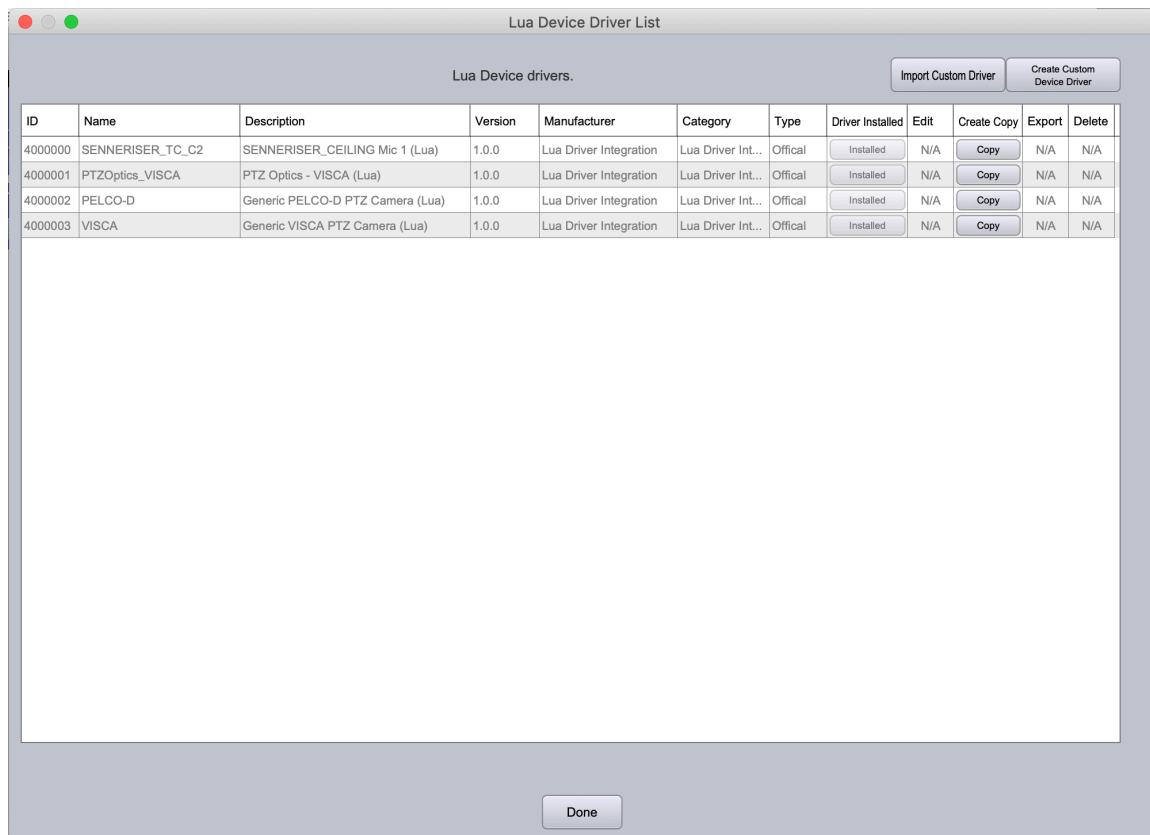
In this dialog, all drivers (Official or Custom) will be shown. If an Official driver has not yet been installed, you can click on the “Install” button to download and install it. Installing the driver here is the same as installing it from the Component Library by double-clicking.

For Official drivers, you have the following options:

- Install the driver (if not yet installed)

- Create a local copy of the driver. You can then modify the driver as desired.

Once you have created a local copy, this copy will be independent of the original driver. The original Official driver will be unchanged.



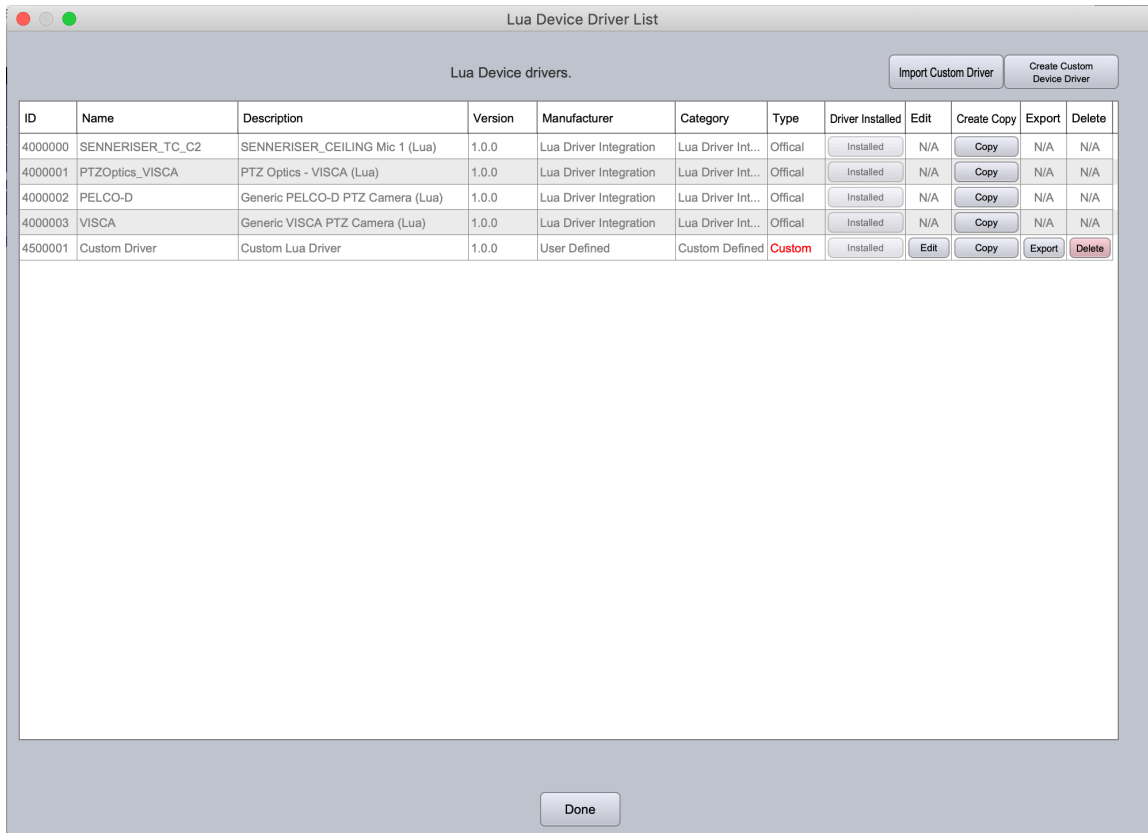
To create a custom, local driver from scratch, click the “Create Custom Device Driver” button. This will create a new driver with a default callback function available.

Note: if you are new to programming in Lua we recommend you make a copy from Official drivers to use as an example and get familiar with the programming environment and syntax.

Once a custom driver is created, it will be listed in the dialog marked as “Custom” and the following options will be available.

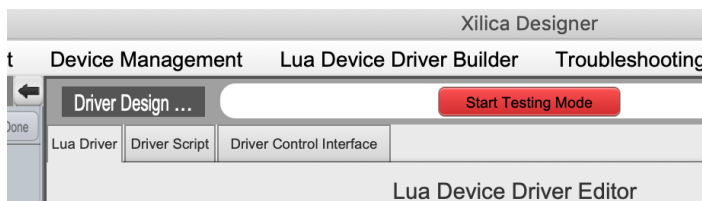
- Edit – edit/modify the driver
- Copy – create a copy from the driver
- Export – export the driver for another user to import into their Xilica Designer software.

- Delete – delete the driver



Once you have decided to create or edit a driver, the builder editor will be displayed. It consists of 3 different tabs.

- "Lua Driver" tab: for basic driver information editing
- "Driver Script" tab: for editing the Lua scripts underlying the driver
- "Driver Control Interface" tab: for editing the user interface for the driver



Device Driver Development

The following describes the process of driver development for each of the four device driver components.

Driver Basic Information

The Driver ID is a unique ID generated by Xilica Designer. Users cannot modify the Driver ID. When a driver is exported and re-imported to a different instance of Xilica Designer, the ID will be re-generated.

Device Name and Device Description will be checked for uniqueness within Xilica Designer's detected drivers. If a name conflict is found, it will suggest the next available name for you.

Most of the items listed in the Driver's Basic Information are for documentation purposes, however the Manufacturer and Device Type will be used to organize the driver in the Xilica Designer Component Library. In Component Library, devices are categorized by manufacturer and then device type, and therefore the Manufacturer and Device Type information controls where your drivers are located in the Library.

In the second section of this screen, users can select the number of analog input/output and number of Dante input/output points for the driver. When the driver is dragged into the project view, the icon will contain the number of I/O points that have been defined. Analog I/O it is for documentation purposes only. For Dante I/O, once defined, the driver will be Dante enabled.

Note: For user-defined custom drivers, the following fields are used to determine its identity and uniqueness:

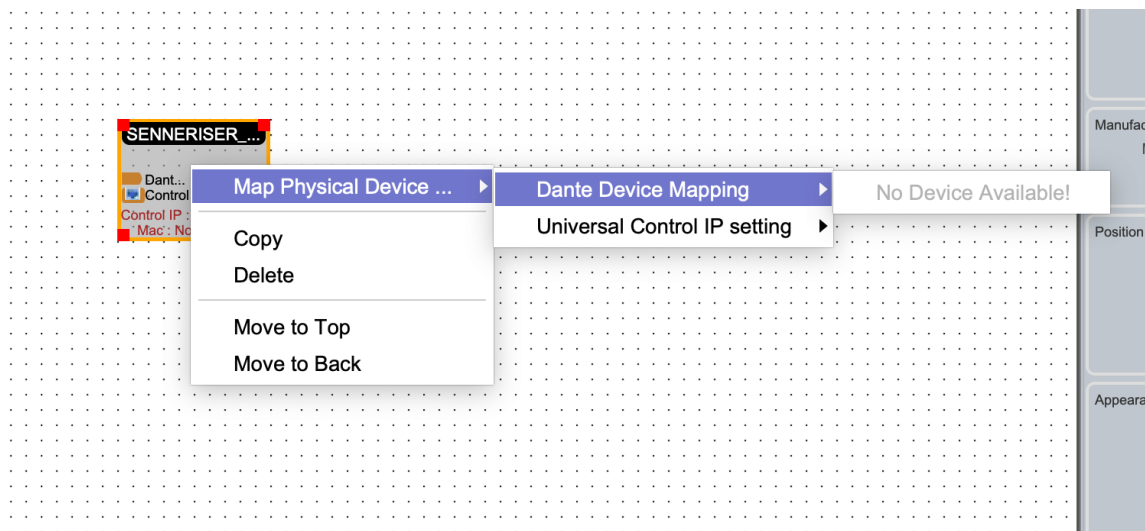
- Driver Description
- Driver Manufacturer
- Author
- Creation Date

Lua Device Driver Editor

Driver ID :	4500005
Device Name :	SENNERISER_TC_C2-1
Description :	SENNERISER_CEILING Mic 1 (Lua)-1
Manufacturer :	Lua Driver Integration
Device Type :	Senneriser
Driver Version :	1.0.0
Author :	Eddie Tam
Create Date :	2020-07-05 00:43:12
Last Modified Date :	2020-07-05 00:43:12

# of Input	0
# of Output	1
# Of Dante Input	1
# Of Dante Output	1

When deploying Dante-enabled drivers in a project, you will need to map it to the corresponding Dante device available in your network (similar to other Dante enabled devices in Xilica Designer). You will also need to map the control IP address for the device as shown below:



Device Specific Parameters

For each individual device, we can define parameters that will be passed to the driver during initialization. We support 4 different data types:

- Numeric
- Boolean
- String
- Binary

For each of these parameters, you can provide a default value, though these values can be overwritten when mapping the driver to a physical device.

Device Specific Parameters				Add Parameter
	Name	Default Value	Data Type	
1	DeviceNumber	1	Numeric	Delete
2	UserName	User1	String	Delete
3	Password	abc	String	Delete
4	BinaryData	01 02 03 ff	Binary (Hex - 'ff ff')	Delete
5	LogicData	1	Boolean (0/1)	Delete

After these parameters are defined for a driver, mapping to a physical device (i.e. mapping its control to an IP address) will bring up the mapping dialog. The mapping dialog displays the parameter list where these values can be edited as needed for this particular device, as shown below:

Device Control IP Setup

Please enter the IP address for device control network interface

Device IP:

Device Specific Parameters

DeviceNumber :	Double	<input type="text" value="1"/>
UserName :	String	<input type="text" value="User1"/>
Password :	String	<input type="text" value="abc"/>
BinaryData :	Binary Data (Hex)	<input type="text" value="01 02 03 ff"/>
LogicData :	Boolean (0/1)	<input type="text" value="1"/>

Once the appropriate parameter values are entered and the project is loaded to hardware (online), the parameter values will be passed to the Lua initialization script (LuaDriverInitialization). We will explain more about the initialization function Section 8. The parameters are being passed to the initialization script as a table and you may access individual elements by their respective name.

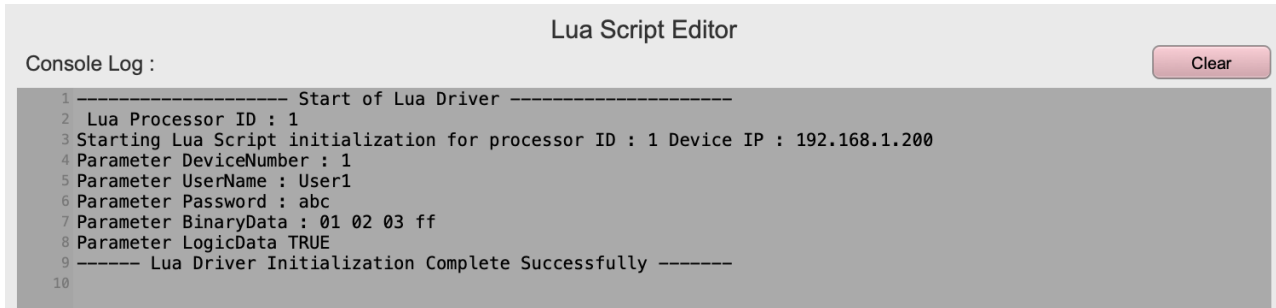
In the Lua script you can obtain the parameter value from the table "deviceParamTable". The following example code will explain how you can access different types of values from the table:

```

Lua Script Editor
Lua Script Body :
2 -- LuaDriverInitialization - Lua Initialization function that will be called when
3 -- we put the driver ONLINE
4 -----
5 function LuaDriverInitialization (id, ip, deviceParamTable)
6
7 G_driverID = id
8 G_deviceIP = ip
9 -- Setup additional device parameter into global variables
10
11 -- Example as below. You can access the table by the device parameter name defined in driver
12 G_deviceNumber = deviceParamTable['DeviceNumber']
13 G_username = deviceParamTable['UserName']
14 G_password = deviceParamTable['Password']
15 G_binaryData = deviceParamTable['BinaryData']
16 G_logicData = deviceParamTable['LogicData']
17
18 XConsoleLog(G_driverID, 'Starting Lua Script initialization for processor ID : ' ..
19             string.format('%d', G_driverID) .. ' Device IP : ' .. G_deviceIP .. '\n')
20
21 XConsoleLog(G_driverID, 'Parameter DeviceNumber : ' .. string.format('%d', G_deviceNumber) .. '\n')
22 XConsoleLog(G_driverID, 'Parameter UserName : ' .. G_username .. '\n')
23 XConsoleLog(G_driverID, 'Parameter Password : ' .. G_password .. '\n')
24 XConsoleLog(G_driverID, 'Parameter BinaryData : ' .. string.format('%02x %02x %02x %02x',
25             string.byte(G_binaryData, 1), string.byte(G_binaryData, 2),
26             string.byte(G_binaryData, 3), string.byte(G_binaryData, 4)) .. '\n')
27 if (G_logicData) then
28     XConsoleLog(G_driverID, 'Parameter LogicData TRUE\n')
29 else
30     XConsoleLog(G_driverID, 'Parameter LogicData FALSE\n')
31 end
32
33 -- Start Network Server Initialization here
34
35 -- Start timer initialization here
36
37 return
38 end
39
40

```

The result of this initialization script execution is shown below:



The screenshot shows a window titled "Lua Script Editor" with a "Console Log" section. A "Clear" button is visible in the top right corner of the log area. The log content is as follows:

```
1 ----- Start of Lua Driver -----  
2 Lua Processor ID : 1  
3 Starting Lua Script initialization for processor ID : 1 Device IP : 192.168.1.200  
4 Parameter DeviceNumber : 1  
5 Parameter UserName : User1  
6 Parameter Password : abc  
7 Parameter BinaryData : 01 02 03 ff  
8 Parameter LogicData TRUE  
9 ----- Lua Driver Initialization Complete Successfully -----  
10
```

Driver Lua Scripting

For driver Lua scripting, each driver is required to implement specific call-back functions that will be executed under different events of a driver. These types of events can be classified into the following categories:

Initialization

LuaDriverInitialization (drvierId, ip, deviceParamTable)

Network events

LuaTCPServerConnectionCreatedCallback (serverId, connectionId)

LuaTCPServerConnectionClosedCallback (serverId, connectionId)

LuaTCPServerMessageReceivedCallback (serverId, connectionId, receivedData)

LuaTCPClientMessageReceivedCallback (clientId, receivedData)

LuaUDPServerMessageReceivedCallback (serverId, receivedData)

User Interface events

LuaUIUpdatedCallback (objectName, strData)

Timer events

LuaTimerCallback ()

The following functions are used to manage the network server, UI handling, and Timer handling.

Note that in all these calls, the first parameter is a "driverId". This ID is a unique identifier to indicate which specific instance the call should route to. This driver ID is the parameter being passed to the LuaInitialization function. All subsequent function calls need to specify this ID to address the proper instance.

Note: We recommend defining the driverId as a global variable so all calls will reference this global variable.

Network service

serverId	XTCPServerSetup (driverId, serverPort, idleTimeout)
bool	XTCPServerClose (driverId, serverId)
bool	XTCPServerReplyMessage (driverId, serverId, clientId, sendData)
clientId	XTCPClientSetup (driverId, remoteIP, remotePort, idleTimeout)
bool	XTCPClientClose (driverId, clientId)
bool	XTCPClientIsConnected (driverId, clientId)
bool	XTCPClientSendMessage (driverId, clientId, sendData)
serverId	XUDPServerSetup (driverId, serverPort)
bool	XUDPServerClose (driverId, serverId)
bool	XUDPClientSendMessage (driverId, remoteIP, remotePort, localPort, sendData)

User Interface service

bool	XUIUpdateRequest (driverId, objectName, UIdata)
------	---

Timer Service

void	XStartTimer (driverId, timeoutInterval)
void	XStopTimer (driverId)

Console Log Service

void	XConsoleLog (driverId, message)
------	---------------------------------

In the “Driver Script” tab, there is an editor for all script function calls. When a new driver is created from scratch, the sample function call signature will be provided as a starting point.

```

Driver Design Mode
Start Testing Mode

Lua Driver | Driver Script | Driver Control Interface
Lua Script Editor

Lua Script Body :
1  -----
2  -- LuaDriverInitialization - Lua Initialization function that will be called when
3  -- we put the driver ONLINE
4  -----
5  function LuaDriverInitialization (id, ip, deviceParamTable)
6
7      G_driverID = id
8      G_deviceIP = ip
9      -- Setup additional device parameter into global variables
10
11      -- Example as below, You can access the table by the device parameter name defined in driver
12      G_username = deviceParamTable['UserName']
13      G_password = deviceParamTable['Password']
14      XConsoleLog(G_driverID, 'Starting Lua Script initialization for processor ID : ' .. string.format('%d', G_driverID) .. ' Device IP : ' .. G_deviceIP .. '\n')
15
16      -- Start Network Server Initialization here
17
18      -- Start timer initialization here
19
20  return
21  end
22
23  -----
24  -- LuaTCPServerConnectionCreatedCallback - Lua TCP server detected a new TCP
25  -- connection. This function will be called with the connection ID.
26  -----
27  function LuaTCPServerConnectionCreatedCallback(serverId, connectionId)
28      G_connectionID = connectionId
29      result = XConsoleLog(G_driverID, 'In LuaTCPServerConnectionCreatedCallback' .. ' server ID : ' .. string.format('%d', serverId) .. ' Connection ID : ' .. string.format('%d', connectionId) .. '\n')
30      return
31  end
32
33  -----
34  -- LuaTCPServerConnectionClosedCallback - Lua TCP server detected a TCP
35  -- connection has been disconnected. This will only be able to detected
36  -- if your server has an idle timeout. If there is no idle timeout
37  -- there is no way to detect that a remote connection has been terminated
38  -----
39  function LuaTCPServerConnectionClosedCallback(serverId, connectionId)
40      G_connectionID = 0
41      result = XConsoleLog(G_driverID, 'In LuaTCPServerConnectionClosedCallback' .. ' server ID : ' .. string.format('%d', serverId) .. ' connection ID : ' .. string.format('%d', connectionId) .. '\n')
42      return
43  end
44
45  -----
46  -- LuaTCPServerMessageReceivedCallback - Callback function when a TCP server
47  -- received message from the network. connectionId is the connection
48  -- that the message is coming from
49  -----
50  function LuaTCPServerMessageReceivedCallback(serverId, connectionId, receivedData)
51      result = XConsoleLog(G_driverID, 'In LuaTCPServerMessageReceivedCallback' .. ' server ID : ' .. string.format('%d', serverId) .. ' connection ID : ' .. string.format('%d', connectionId) .. '\n')
52      result = XConsoleLog(G_driverID, receivedData .. '\n')
53      return
54  end
55
56  -----
57  -- LuaTCPClientMessageReceivedCallback - Callback function when a TCP client
58  -- connection received data from remote end.
59  -----
60  function LuaTCPClientMessageReceivedCallback(clientId, receivedData)
61      result = XConsoleLog(G_driverID, 'In LuaTCPClientMessageReceivedCallback' .. ' client ID : ' .. string.format('%d', clientId) .. '\n')
62      result = XConsoleLog(G_driverID, receivedData .. '\n')
63      return
64  end
65
66  -----
67  -- LuaUDPServerMessageReceivedCallback - Callback function when a UDP server
68  -- received message from the network.
69  -----
70  function LuaUDPServerMessageReceivedCallback(serverId, receivedData)
71      result = XConsoleLog(G_driverID, 'In LuaUDPServerMessageReceivedCallback' .. ' server ID : ' .. string.format('%d', serverId) .. '\n')
72      result = XConsoleLog(G_driverID, receivedData .. '\n')
73      return
74  end
75
76  -----
77  -- LuaUIUpdatedCallback - Callback function when UI changes has been detected
78  -----

```

For each new driver, the driver developer must implement a script for the following events:

LuaDriverInitialization

LuaDriverInitialization (driverId, ip, deviceParamTable)

driverId	Unique Global driver ID to identify the driver environment
ip	IP address of this driver instance
deviceParamTable	A table containing all user-entered device-specific parameters

This script will be called when a driver starts and will be called once and only once for each driver session. We recommend using this initialization script to perform the following:

- Setting up all global variables, especially the unique driver ID, as this needs to be accessed throughout all other script function calls.
- Accessing the device-specific parameter from the deviceParamTable. The value can be retrieved from the table by accessing its array elements (e.g. to get the device username – deviceParamTable[UserName] where UserName variables can be defined when you build the driver.
- Setting up communication sessions. i.e. TCP server setup, TCP client setup, UDP server setup.
- Setting up timers if you require periodic sending or polling of data from device.

LuaTCPServerConnectionCreatedCallback

LuaTCPServerConnectionCreatedCallback (serverId, connectionId)

serverId Indicates which server has an incoming TCP connection
 connectionId A unique Id within the specific server

This script will be called when a previously setup server receives an incoming connection. Once a TCP session is established, a unique connection ID will be used to identify this connection.

LuaTCPServerConnectionClosedCallback

LuaTCPServerConnectionClosedCallback (serverId, connectionId)

serverId Indicates which server has incoming TCP disconnected
 ConnectionId Indicates which particular connection is closed

This script will be called when a TCP connection has been disconnected.

LuaTCPServerMessageReceivedCallback

LuaTCPServerMessageReceivedCallback (serverId, connectionId, receivedData)

serverId Indicates which server has an incoming message
 connectionId Indicates which connection has an incoming message
 receivedData Message as string data in Lua

This script will be called when a previously setup server and specific connection session have received data. The incoming string data can store binary data in Lua.

LuaTCPClientMessageReceivedCallback

LuaTCPClientMessageReceivedCallback (clientId, receivedData)

clientId Indicates which TCP client has an incoming message
 receivedData Message as string data in Lua

This script will be called when a previously setup TCP client connection has received data.

LuaUDPServerMessageReceivedCallback

LuaUDPServerMessageReceivedCallback (serverId, receivedData)

serverId Indicate which UDP server has incoming message

receivedData Message as string data in Lua

This script will be called when a previously setup UDP server has incoming message received.

LuaUIUpdatedCallback

LuaUIUpdatedCallback (objectName, strData)

objectName Object name in ASCII format that has been updated

strData New data value in string format

This script will be called when a UI object value has been changed, e.g., a fader has been moved. User should obtain the strData and then send it to the remote device to update the device status.

LuaTimerCallback

LuaTimerCallback ()

This script will be called when a timer expires. After a timer has been setup, this script will be called periodically. You can make use of this script to perform value polling. You can also call XStopTimer function within the script to stop the timer from triggering again.

Within your Lua script, in addition to standard Lua programming functionality, we also provide the following function calls for communication and other functions. The supporting function calls can be classified into the following groups:

- TCP/UDP communication setup
- UI manipulation
- Timer manipulation
- Console log message for debugging

Note: In order for each of these function calls to access the proper Lua environment of the individual device instance, each function call needs to provide the unique driver ID (which is provided in the Lua Initialization script call) as the first argument to the function call.

XTCPServerSetup

serverId XTCPServerSetup (driverId, serverPort, idleTimeout)

driverId Id to identify the individual driver instance

serverPort Server port for the TCP server

idleTimeout Idle timeout value (in ms) for the server to disconnect idle connection

return serverId which is a unique ID for the server

This function call will setup a TCP server at the port specified. If setup is successful it will return a non-zero serverId. This ID will be used in future scripts to identify the server being setup. The server being setup will wait for an incoming TCP connection. (If the setup fails, it will return 0.)

<i>Bool</i>	<i>XTCPServerClose (driverId, serverId)</i>
driverId	Id to identify the individual driver instance
serverId	Indicates which server to close
return	True if successful, otherwise false.

This function will close a previously created TCP server.

XTCPServerReplyMessage

<i>bool</i>	<i>XTCPServerReplyMessage (driverId, serverId, clientId, sendData)</i>
driverId	Id to identify the individual driver instance
serverId	Indicates which server to send message to
clientId	Indicates which client connection to send message to
sendData	String (binary) data to send to network
return	True if successful, otherwise false.

This function will send the message to a TCP server connection indicated by the serverId and clientId.

XTCPClientSetup

<i>clientId</i>	<i>XTCPClientSetup (driverId, remoteIP, remotePort, idleTimeout)</i>
driverId	Id to identify the individual driver instance
remoteIP	Remote device IP address to connect to
remotePort	Remote TCP port to connect to
idleTimeout	Indicate idle timeout (in ms) to disconnect the connection (0 - means no timeout)
return	A unique clientId that you can use to send data to within the script.

This function call will setup a TCP client connection to a remote device with a specified IP and port. If idleTimeout is set, the connection will be closed when idle for the time specified. Once the connection times out, you can simply send another message to it and it will perform a re-connection to the remote server.

When this setup call is made it will not actually create the TCP connection to the remote until you send a message to this client.

XTCPClientClose

<i>bool</i>	<i>XTCPClientClose (driverId, clientId)</i>
driverId	Id to identify the individual driver instance
serverId	Indicate which server to close
return	True if successful, otherwise false.

This function call will close a previously created UDP server.

XTCPClientIsConnected

<i>bool</i>	<i>XTCPClientIsConnected (driverId, clientId)</i>
<i>driverId</i>	Id to identify the individual driver instance
<i>clientId</i>	Client to check for connection
<i>return</i>	True if the connection still active, otherwise false

This function call will check whether a client connection is currently connected. If it is not connected, you can re-connect simply by sending a message to the client. The reason to provide such a check is to enable the ability to send specific messages when a TPC client connection is formed. This is useful in the case where a TCP connection requires a login.

XTCPClientSendMessage

<i>bool</i>	<i>XTCPClientSendMessage (driverId, clientId, sendData)</i>
<i>driverId</i>	Id to identify the individual driver instance
<i>clientId</i>	Indicates which client connection to send message to
<i>sendData</i>	String (binary) data to send to network
<i>return</i>	True if successful, otherwise false.

This function call will send a message to a specific TCP client connection that has been previously setup. If the device has not connected yet, this send message will connect to remote before sending the message.

XUDPServerSetup

	<i>serverIdXUDPServerSetup (driverId, serverPort)</i>
<i>driverId</i>	Id to identify the individual driver instance
<i>serverPort</i>	Server port for the UDP server
<i>return</i>	<i>serverId</i> which is a unique ID for the server

This function call will setup a UDP server at the server port.

XUDPServerClose

<i>bool</i>	<i>XUDPServerClose (driverId, serverId)</i>
<i>driverId</i>	Id to identify the individual driver instance
<i>serverId</i>	Indicate which server to close
<i>return</i>	True if successful, otherwise false.

This function will close a previously created UDP server.

XUDPClientSendMessage

<i>bool</i>	<i>XUDPClientSendMessage (driverId, remoteIP, remotePort, localPort, sendData)</i>
driverId	Id to identify the individual driver instance
remoteIP	Remote device IP address to connect to
remotePort	Remote TCP port to connect to
localPort	Local port to use (0 – means O/S will choose a port for you)
sendData	String (binary) data to send to network
return	True if successful, otherwise false.

This function call will send message to a remote UDP server. As UDP is connectionless, it is not required to setup a client before sending message to a remote UDP server.

XUIUpdateRequest

<i>bool</i>	<i>XUIUpdateRequest (driverId, objectName, Uldata)</i>
driverId	Id to identify the individual driver instance
objectName	UI object ASCII name to updated. If more than one UI object has the same name, all UI objects with that name will be updated
Uldata	Date to update to UI objects.
return	True if successful, otherwise false.

This function call will update the UI object identified by the objectName. There could be more than one UI object with the same name. This call will update all objects with that name.

XStartTimer

<i>Void</i>	<i>XStartTimer (driverId, timeoutInterval)</i>
driverId	Id to identify the individual driver instance
timeout	Interval Timeinterval in millisecond
return	None.

This function call will start a periodic timer. When a timeout occurs, it will call the corresponding Lua script. You can only setup a single timer in a Lua environment. If you run XStartTimer again, it will replace the previously setup timeoutInterval.

XStopTimer

<i>Void</i>	<i>XStopTimer (driverId)</i>
driverId	Id to identify the individual driver instance
return	None.

This function call will stop a previously setup timer.

XConsoleLog

<code>void</code>	<code>XConsoleLog (driverId, message)</code>
<code>driverId</code>	Id to identify the individual driver instance
<code>message</code>	String message send to console log
<code>return</code>	None.

This function call will send the message to the console log. You can see the console log message during the debugging session of the driver. During normal run time, this call has no effect.

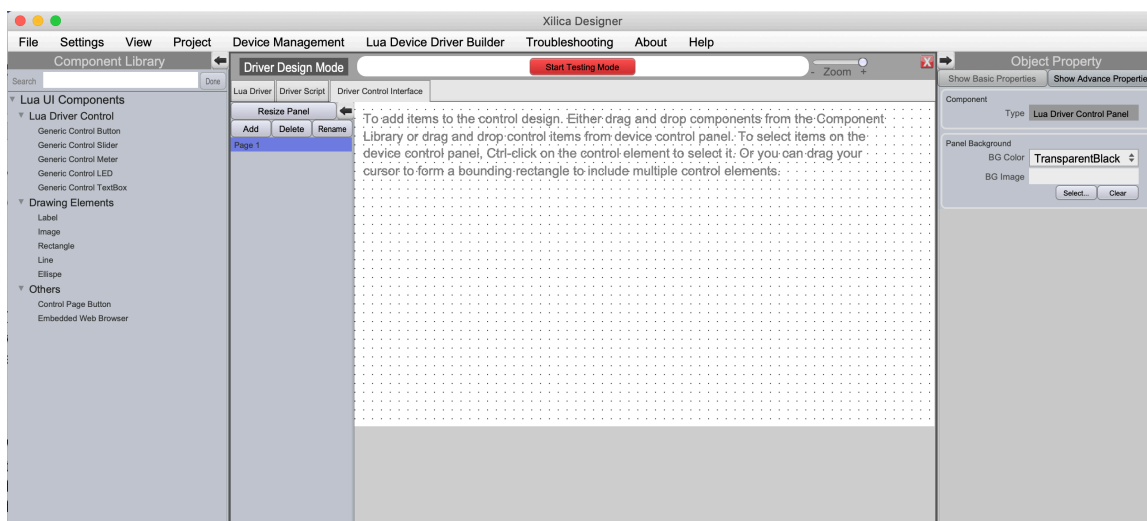
User Interface Builder

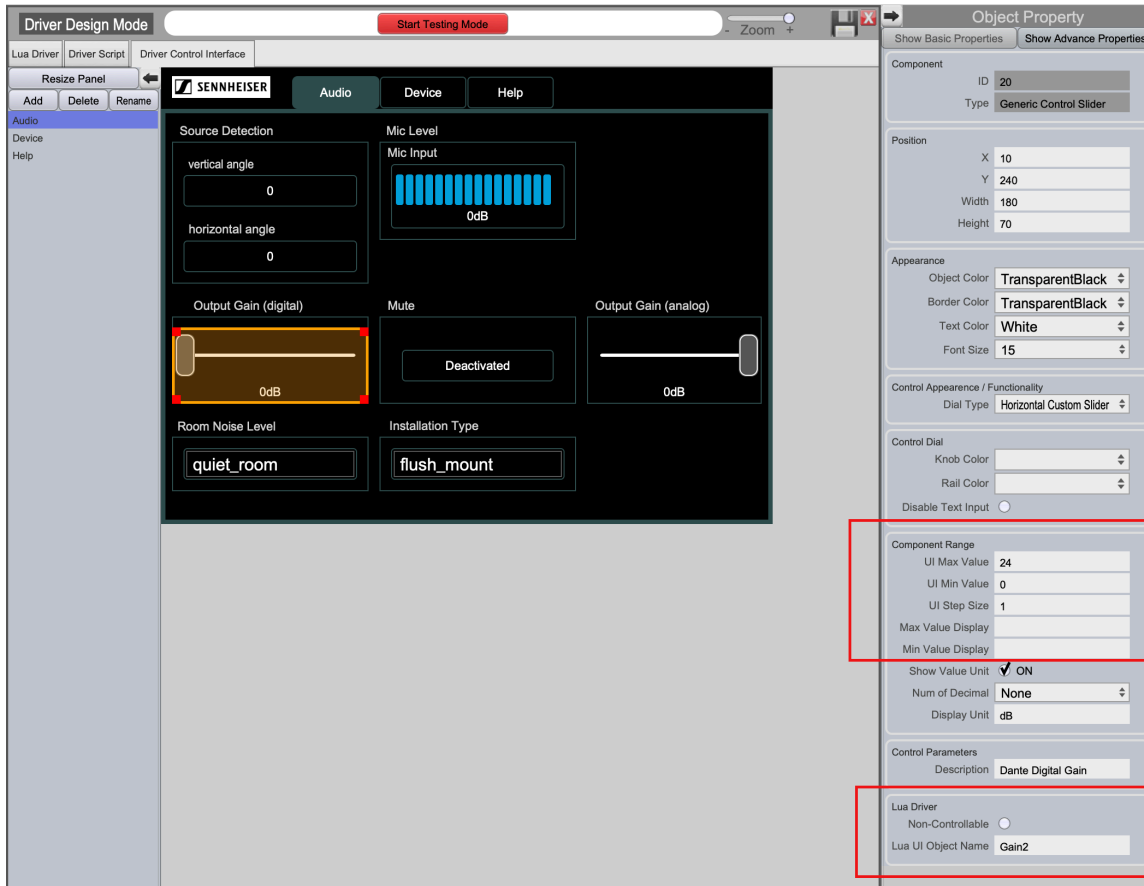
Each driver is associated with its own user interface. Creating a driver user interface is similar to building a user interface for an XTouch device. You can drag in faders, buttons, meters, etc. into the interface editor.

Each UI object you put into the interface can be named, and in the script, you can access this UI object by its name.

In addition, you can setup multiple pages and link these pages using page change buttons.

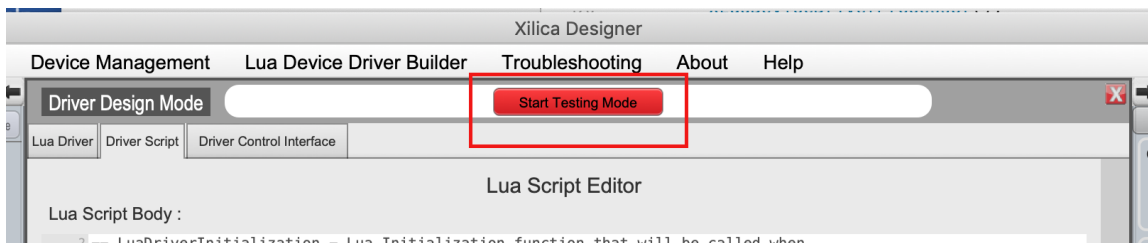
In the interface, you can name more than one object with same name. In this case any object value change in the UI will trigger a UI update request event for the Lua event. If the Lua script decides to change the value of a named UI object, all objects with same name will be updated simultaneously.



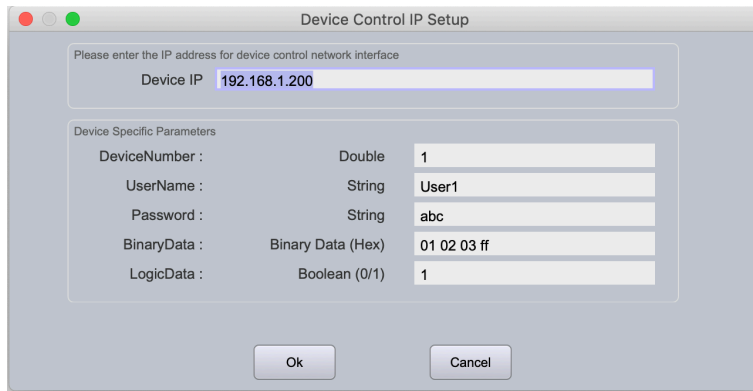


DRIVER TESTING AND DEBUGGING

Once you have finished setting up the driver information, script body, and user interface, you are then ready to test your driver. To test the driver, simply click the “Start Testing Mode” button on the middle top of the menu bar.



The driver must be saved before testing can begin. To start testing, a dialog will be displayed to ask you to map to a physical device by entering the target IP address as well as the related parameters for your testing session.



The image shows a dialog box titled "Device Control IP Setup". It contains a text input field for "Device IP" with the value "192.168.1.200". Below this is a section titled "Device Specific Parameters" with five rows of input fields:

Parameter	Type	Value
DeviceNumber	Double	1
UserName	String	User1
Password	String	abc
BinaryData	Binary Data (Hex)	01 02 03 ff
LogicData	Boolean (0/1)	1

At the bottom of the dialog are "Ok" and "Cancel" buttons.

Once the information is entered and you click Ok, the driver will be ONLINE. All editing parameters will become read-only, and a console log area will be displayed on top of the script editor. All console messages generated in the script will be displayed in the log area. If you find the log to be too long, you can press the "Clear" button to clear the old log.

In the script area, you can modify your script directly online. Pressing the "Apply Script" button will re-start the script initialization. All opened connections will be closed and then re-opened according to the new initialization script.

Driver Testing Mode
Stop Testing Mode
✕

Lua Driver
Driver Script
Driver Control Interface

Lua Script Editor

Console Log :
Clear

```

1 ----- Start of Lua Driver -----
2 Lua Processor ID : 1
3 Starting Lua Script initialization for processor ID : 1 Device IP : 192.168.1.200
4 Parameter DeviceNumber : 1
5 Parameter UserName : User1
6 Parameter Password : abc
7 Parameter BinaryData : 01 02 03 ff
8 Parameter LogicData TRUE
9 ----- Lua Driver Initialization Complete Successfully -----
10

```

Lua Script Body :
Apply Script

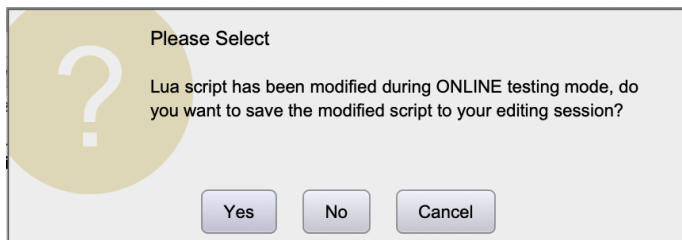
```

2 -- LuaDriverInitialization - Lua Initialization function that will be called when
3 -- we put the driver ONLINE
4 -----
5 function LuaDriverInitialization (id, ip, deviceParamTable)
6
7 G_driverID = id
8 G_deviceIP = ip
9 -- Setup additional device parameter into global variables
10
11 -- Example as below. You can access the table by the device parameter name defined in driver
12 G_deviceNumber = deviceParamTable['DeviceNumber']
13 G_username = deviceParamTable['UserName']
14 G_password = deviceParamTable['Password']
15 G_binaryData = deviceParamTable['BinaryData']
16 G_logicData = deviceParamTable['LogicData']
17
18 XConsoleLog(G_driverID, 'Starting Lua Script initialization for processor ID : ' ..
19 string.format('%d', G_driverID) .. ' Device IP : ' .. G_deviceIP .. '\n')
20
21 XConsoleLog(G_driverID, 'Parameter DeviceNumber : ' .. string.format('%d', G_deviceNumber) .. '\n')
22 XConsoleLog(G_driverID, 'Parameter UserName : ' .. G_username .. '\n')
23 XConsoleLog(G_driverID, 'Parameter Password : ' .. G_password .. '\n')
24 XConsoleLog(G_driverID, 'Parameter BinaryData : ' .. string.format('%02x %02x %02x %02x',
25 string.byte(G_binaryData, 1), string.byte(G_binaryData, 2),
26 string.byte(G_binaryData, 3), string.byte(G_binaryData, 4)) .. '\n')
27 if (G_logicData) then
28 XConsoleLog(G_driverID, 'Parameter LogicData TRUE\n')
29 else
30 XConsoleLog(G_driverID, 'Parameter LogicData FALSE\n')
31 end
32
33 -- Start Network Server Initialization here
34
35 -- Start timer initialization here
36
37 return
38 end
39
40 -----
41 -- LuaTCPServerConnectionCreatedCallback - Lua TCP server detected a new TCP
42

```

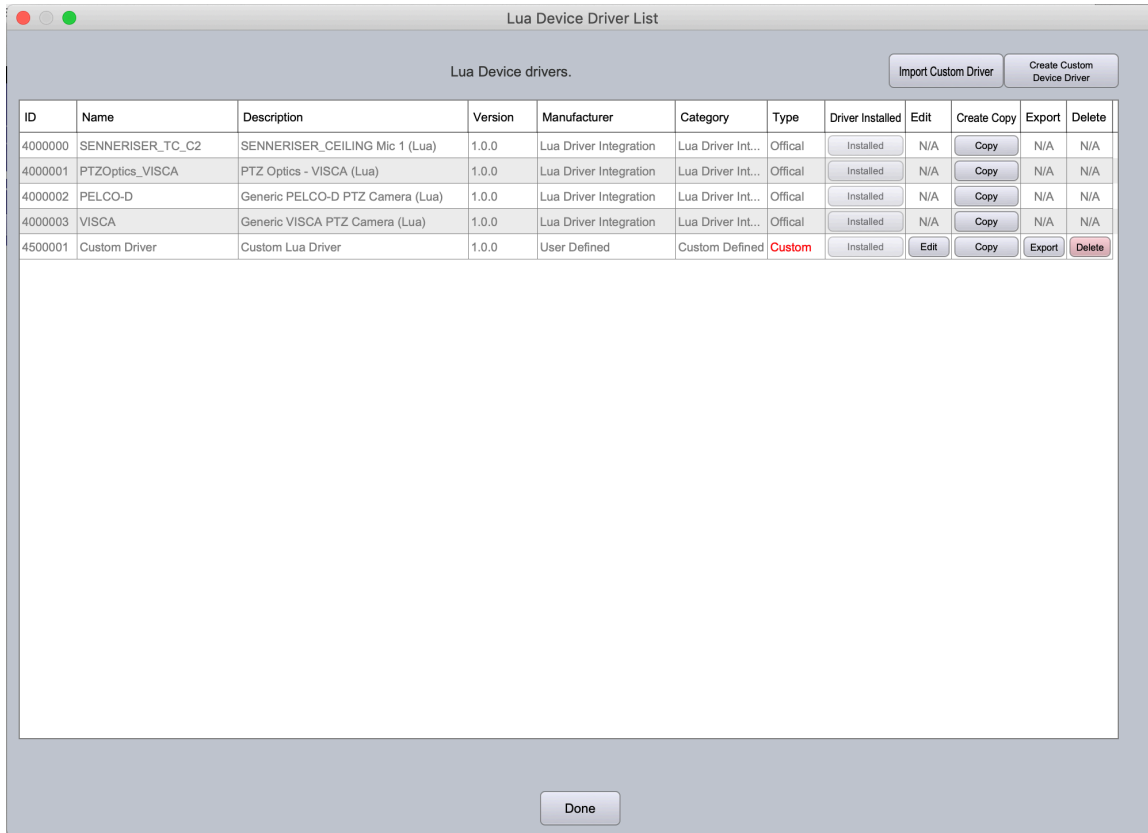
The control interfaces will also go into ONLINE controlling mode. You can change fader and button states and the related script will be executed, and you can check the log from those scripts. If changes have been made from the remote end, your Lua script can also drive the UI object state as well.

When you have finished with the testing session, press “Stop Testing Mode” to stop testing and go back to editing mode. If you have modified the script during testing, a dialog will be displayed to ask you whether you want to save the script modification or discard it.



DRIVER IMPORTING AND EXPORTING

Custom drivers can be exported by clicking the corresponding “Export” button. The export is an XML file containing all information (including driver info, the script, and control interfaces). You can then share this file with others.



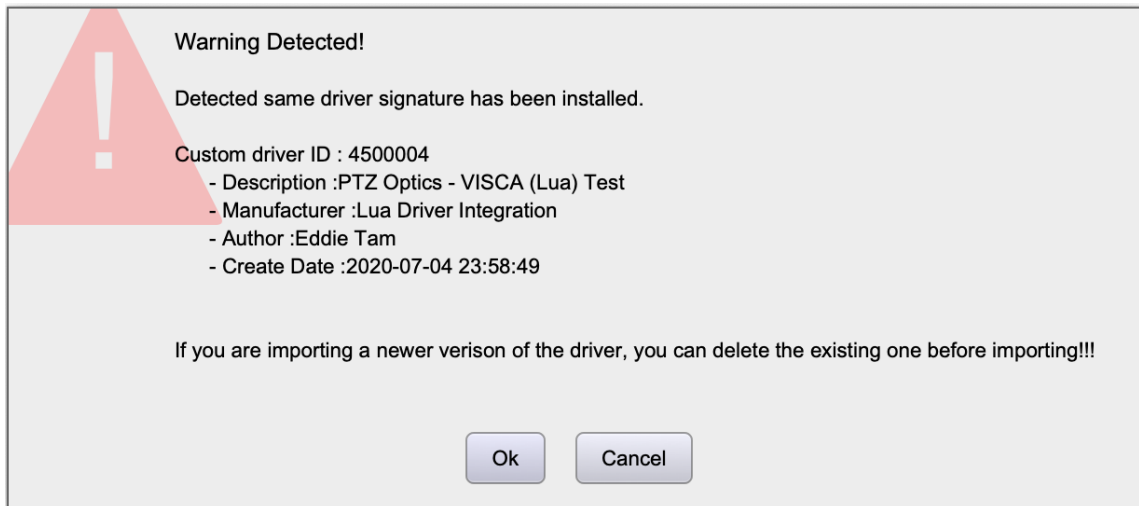
For others to use the driver, they can click the "Import Custom Driver" button to import the driver to their Xilica Designer. Once imported, the devices will automatically be listed in the Component Library.

Once a driver has been installed, you cannot re-install the same driver again. You need to delete the old one before you can re-install it. If you try to re-install the same driver, a warning dialog will be displayed.

Note: For user-defined custom drivers, the following fields are used to define the driver's identity:

- Driver Description
- Driver Manufacturer
- Author
- Creation Date

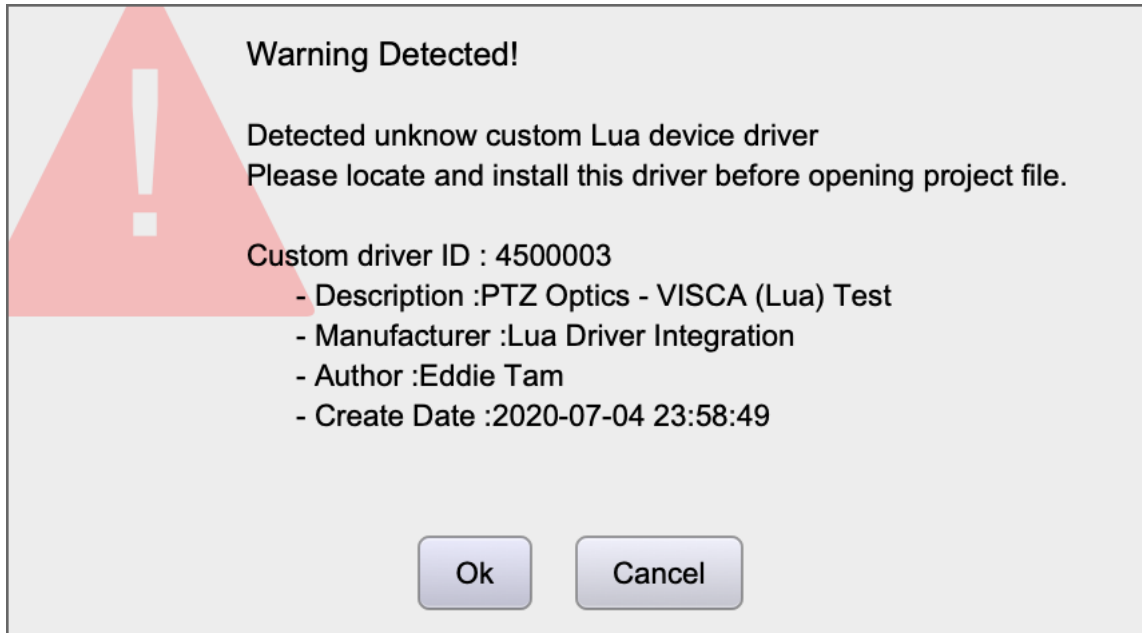
If any of these parameters change the driver will be considered a different driver.



PROJECT FILES AND DRIVERS

If you receive project files containing user-defined Custom drivers, you will NOT be able to open the project file. A dialog will be displayed indicating the specific information of the missing driver. You must contact the driver owner and install the custom driver(s) before the project file can be opened.

During driver import, a different driver ID will be automatically assigned locally to your Xilica Designer installation.



For a project file that includes devices with Official drivers, Xilica Designer will confirm that these drivers have been installed. If not, simply install the drivers Designer's project editor. All Official drivers will be listed on the left-hand side. Locate the driver and double click on it to install. Once installed, you will be able to open the project file.

