Adobe
Lightroom
Classic

Artwork by Marina Weishaupt.

Reading preferences...

Katy Montanez, Rahul Agrawal, Arjun Haarith G, Anshul Patel, Zahid Syed, Satish Thareja, Kanheiya Agrawal, Chinoy Gupta, Ben Olsem, Thomas Knoll, Eric Chan, Max Wendt, Joshua Bury, Julieanne Kost, Gopinath Srinivasan, Rikk Flohr, Sumeet Choudhury, Suhas Muthmurdu, Daniel Lee, Pavan Kumar T S, Krishna Singh Karki, Bruce Kaskel, Lisa Ngo, Vikash Mehta, Krishna Maithreya Samboji, Ivan Mcclellan, Vishal Joshi, Nandakumar Puzhankara, Nitin Sharma, Robert Christensen, Aiana Verma, Divya Jain, Rishi Kumar, Vidhya S, Somashekhara V, Katrin Eismann, Aditi Shah, Soumi Mitra, Sai Teja Gopi, Tina Chang, Nikhil Kumar Singh, Shabreen Taj, Sriya Challapalli, Dan Gerber, Melissa Monroe

Lightroom Classic SDK

Programmers Guide

Addendum:

Externally Controlling Lightroom Classic

Adobe Photoshop Lightroom Classic SDK Programmers Guide Addendum: Externally Controlling Lightroom Classic

Contents

Preface **3**

1 Overview **3**

2 Creating Your Plug-in **3**

Plug-in Configuration 4

Useful Namespaces 4

3 Compiling Your Plug-in **4**

4 Installing Your Plug-in **5**

5 Getting Started: An Example Plug-in **5**

Info.lua 5

Start.lua 6

Stop.lua 12

Shutdown.lua 12

Preface

This guide demonstrates how to use the Lightroom Classic SDK to write a plug-in that enables external control of the application (for example, to allow a hardware device like a physical knob to control a slider in the Develop module).

The intended audience for this guide are hardware manufacturers who wish to integrate their controller device with Lightroom Classic, which involves authoring a companion Lightroom Classic plug-in which their device driver sends commands to. More generally, this guide is for developers who want to enable communication between a Lightroom Classic plug-in and another application.

1 Overview

You will need to provide two components: a Lightroom Classic plug-in and your device driver (or similar software). Your hardware connects to your driver, your driver communicates with your plug-in via socket connections, and your plug-in interacts with Lightroom Classic via the Lightroom Classic SDK.

For example, manipulating your device might cause your driver to tell your plug-in to set Exposure to the value 1.5, which your plug-in would then accomplish by calling:

```
LrDevelopController.setValue( "Exposure", 1.5 )
```

This communication can also go in the opposite direction: your Lightroom Classic plug-in could notify your driver whenever the value of a slider in the Develop module changes, for example.

2 Creating Your Plug-in

First, please review the Lightroom Classic SDK Programmer's Guide for general information about authoring a Lightroom Classic plug-in. Also be sure to see the API reference found inside the SDK download itself for complete documentation of all APIs, including several namespaces mentioned in this document that are not covered by the SDK Programmer's Guide (like `LrDevelopController`).

Your controller plug-in will just be a normal Lightroom plug-in, except it will establish a socket connection with your driver and await instructions. You can define whatever string-based messaging protocol you want for the communication between your plug-in and your driver. You can pick any available ports on **localhost** for your connections, or have ports be selected automatically by the OS. The simplest approach is to preselect your ports and hard-code them in your plug-in and driver. A more robust solution would be to have your driver find an available port, record it in a configuration file somewhere, and have your plug-in read this file at startup to determine which port to use.

Typically, you will want to open the socket connection automatically when Lightroom starts up and keep it open until

Lightroom Classic shuts down, requiring no user interaction, although you can also allow the user to start and stop it manually through custom menu commands that you provide. The following section explains how to accomplish all these things.

Plug-in Configuration

As covered in Chapter 2 of the Programmer's Guide, your plug-in is configured by its **`Info.lua`** manifest file. To make your plug-in automatically start when Lightroom Classic launches and then stop cleanly at shutdown, it should include the following:

- Set **`LrInitPlugin`** to be your main script that establishes your socket connection and continuously listens for and responds to messages from your driver. The simplest way to implement this is to bind to the socket and then enter a loop that waits for your shutdown script to signal exit. (See below for an example.)
- Set **`LrForceInitPlugin`** to **`true`** so your init script will automatically run at startup.
- Declare at least one menu item, since **`LrForceInitPlugin`** requires this. This can be anything -- start/stop scripts for manually controlling your plug-in, or even something as simple as a help dialog.
- Set **`LrShutdownApp`** to be a script that signals to your main script that it should exit its loop and close its socket connections.

An example of a plug-in that does all of these can be found at the end of this document.

Useful Namespaces

Your plug-in can use any namespaces in the SDK, but the following are likely to be particularly useful for a hardware controller:

`LrSocket`	Provides the ability to send and receive data from other processes using TCP sockets.
`LrApplicationView`	Provides access to the application's view state, including module, main view, secondary view, and zoom control.
`LrDevelopController`	Provides access to controls in the Develop module, including setting and getting slider values, resetting defaults, and selecting tools.
`LrSelection`	Provides access to selection-based commands, including changing which image is selected and setting its flags, color labels, and ratings.
`LrSlideshow`	Allows for starting and stopping of slideshows.
`LrTether`	Gives control over tethered shooting.
`LrUndo`	Provides access to undo/redo commands.

3 Compiling Your Plug-in

Lightroom Classic plug-ins are written in Lua which does not require compilation, although it does support it. If you

wish to obfuscate your plug-in code, just replace your source files with their compiled versions. Lightroom Classic uses version 5.1.5 of the Lua language.

The Lua compiler (luac) is available as a part of the Lightroom Classic SDK Guide, inside "Lua Compiler" folder under the appropriate platform. Run it on each of your source files, giving the same name for the output file as referenced in your **`Info.lua`** file.

For example:

```
luac.exe source/start.lua -o build/start.lua
```

Put all your files together into the same folder, including your **`Info.lua`**, give this folder the extension **`.lrplugin`**, and that folder is now your Lightroom Classic plug-in.

For more details about the structure of a Lightroom Classic plug-in, please see the **"Delivering a standard plug-in"** section of Chapter 2 in the SDK Programmer's Guide.

4 Installing Your Plug-in

Users can manually install plug-ins through Lightroom Classic's Plug-In Manager dialog, but the best approach is to have your device driver's installer also install your Lightroom Classic plug-in for them. This just involves copying your plug-in to a specific location where Lightroom Classic will find it.

For more details about automatically installing the plugins for use in Lightroom Classic, please see the "**Automatic plug-in loading**" sub-section inside "**Delivering a standard plug-in**" section of Chapter 2 in the Lightroom Classic SDK Programmer's Guide.

5 Getting Started: An Example Plug-in

The following is an example of a complete plug-in that:

- Automatically opens two sockets when Lightroom Classic starts up, one for receiving and one for sending. (If your hardware device doesn't display values or provide any sort of feedback, you probably only need the receiving side.)
- Listens for commands from the receiving socket formatted as "**`key = value`**" strings (ex: "**`Exposure = 1.2`**", or "**`rating = 3`**"), parses them, and executes the corresponding commands in Lightroom Classic.
- Notifies the sending socket whenever the current image's Develop settings change, formatted in a similar way.
- Continues receiving and sending in a loop until an exit is signalled, at which point it closes both socket connections and exits.
- Signals for the loop to exit when Lightroom Classic begins its shutdown process.
- Adds two menu commands under "**File** > **Plug-in Extras**" for manually starting and stopping the plug-in.

`Info.lua`

```
--[[----------------------------------------------------------------------------
ADOBE INC.

------------------------

Copyright 2024 Adobe
All Rights Reserved.
```

```
NOTICE: Adobe permits you to use, modify, and distribute this file in accordance
with the terms of the Adobe license agreement accompanying it. If you have received
this file from a source other than Adobe, then your use, modification, or distribution
of it requires the prior written permission of Adobe.

------------------------------------------------------------------------------]]
return {
	LrSdkVersion = 6.0,
	LrPluginName = "Controller Example",
	LrToolkitIdentifier = 'com.company_name.controller_example',
	LrInitPlugin = "start.lua", -- runs when plug-in initializes (this is the main script)
	LrForceInitPlugin = true, -- initializes the plug-in automatically at startup.
	LrShutdownApp = "shutdown.lua", -- tells the main script to exit and waits for it to finish.
	LrShutdownPlugin = "shutdown.lua",
	LrDisablePlugin = "stop.lua", -- tells the main script to exit.
	LrExportMenuItems = {
		{
			title = "Start",
			file = "start.lua",
		},
		{
			title = "Stop",
			file = "stop.lua",
		},
	},
	VERSION = { major=6, minor=9, revision=0, build="201905010000-a1b2c3d4", },
}
```

Start.lua

```
--[[------------------------------------------------------------------------
ADOBE INC.

_____________________

Copyright 2024 Adobe
All Rights Reserved.

NOTICE: Adobe permits you to use, modify, and distribute this file in accordance
with the terms of the Adobe license agreement accompanying it. If you have received
this file from a source other than Adobe, then your use, modification, or distribution
of it requires the prior written permission of Adobe.

------------------------------------------------------------------------------]]
local LrDialogs = import "LrDialogs"
local LrFunctionContext = import "LrFunctionContext"
local LrTasks = import "LrTasks"
local LrApplication = import "LrApplication"
local LrSelection = import "LrSelection"
local LrDevelopController = import "LrDevelopController"
local LrSocket = import "LrSocket"
local LrTableUtils = import "LrTableUtils"

--============================================================================
-- Port numbers

-- port zero indicates that we want the OS to auto-assign the port
local AUTO_PORT = 0

-- port number used to send change notifications
local sendPort = AUTO_PORT

-- port number used to receive commands
local receivePort = AUTO_PORT

--============================================================================
-- All of the Develop parameters that we will monitor for changes. For a complete
-- listing of all parameter names, see the API documentation for LrDevelopController.
```

```
local develop_params = {
	"Temperature",
	"Tint",
	"Exposure",
	"Contrast",
	"Highlights",
	"Shadows",
	"Whites",
	"Blacks",
	"Clarity",
	"Vibrance",
	"Saturation",
}

local develop_param_set = {}

for _, key in ipairs( develop_params ) do
	develop_param_set[ key ] = true
end

--------------------------------------------------------------------------------
-- Checks to see if observer[ key ] is equal to the given value. If the value has
-- changed, reports the change to the given sender.
-- Used to notify external processes when settings change in Lr.

local function updateValue( observer, sender, key, value )

	if observer[ key ] ~= value then

		-- for table types, check if any values have changed
		if type( value ) == "table" and type( observer[ key ] ) == "table" then
			local different = false
			for k, v in pairs( value ) do
				if observer[ key ][ k ] ~= v then
					different = true
					break
				end
			end
			for k, v in pairs( observer[ key ] ) do
				if value[ k ] ~= v then
					different = true
					break
				end
			end
			if not different then
				return
			end
		end

		observer[ key ] = value

		local data = LrTableUtils.tableToString {
			key = key,
			value = value,
		}

		if WIN_ENV then
			data = string.gsub( data, "\n", "\r\n" )
		end

		sender:send( data )
	end
end

--------------------------------------------------------------------------------
-- Given a key/value pair that has been parsed from a receiver port message, calls
-- the appropriate API to adjust a setting in Lr.
```

```
local function setValue( key, value )

	if value == "+" then 							-- ex: "Exposure = +"
		LrDevelopController.increment( key )
		return true
	end

	if value == "-" then 							-- ex: "Exposure = -"
		LrDevelopController.decrement( key )
		return true
	end

	if value == "reset" then 						-- ex: "Exposure = reset"
		LrDevelopController.resetToDefault( key )
		return true
	end

	if key == "label" then 							-- ex: "label = red"
		LrSelection.setColorLabel( value )
		return true
	end

	if key == "select" then 						-- ex: "select = next"
		if value == "next" then
			LrSelection.nextPhoto()
			return true
		elseif value == "previous" then
			LrSelection.previousPhoto()
			return true
		end
		return false
	end

	local numericValue = tonumber( value )

	if numericValue then

		if key == "rating" then 					-- ex: "rating = 3"
				LrSelection.setRating( numericValue )
				return true
		end

		if key == "flag" then 						-- ex: "flag = 1"
			if numericValue == -1 then
				LrSelection.flagAsReject()
				return true
			elseif numericValue == 0 then
				LrSelection.removeFlag()
				return true
			elseif numericValue == 1 then
				LrSelection.flagAsPick()
				return true
			end
			return false
		end

		if key and develop_param_set[ key ] then 	-- ex: "Exposure = 1.5"
			LrDevelopController.setValue( key, numericValue )
			return true
		end
	end
end

--------------------------------------------------------------------------------
-- Simple parser for messages sent from the external process over the socket
-- connection, formatted as "key = value". (ex: "rating = 2")

local function parseMessage( data )
	if type( data ) == "string" then
```

```
                local _, _, key, value = string.find( data, "([^ ]+)%s*=%s*(.*)" )
                return key, value
        end
end

--------------------------------------------------------------------------------
-- checks all Develop parameters for any changes that happened in Lr, reporting
-- them to the sender socket.

local function updateDevelopParameters( observer )
        local sender = observer._sender
        for _, param in ipairs( develop_params ) do
                updateValue( observer, sender, param, LrDevelopController.getValue( param ) )
        end
end

--------------------------------------------------------------------------------

local senderPort, senderConnected, senderObserver
local receiverPort, receiverConnected

--------------------------------------------------------------------------------
-- Called by both the send socket and the receive socket when they begin their
-- attempt to establish a connection to a port number.

local function maybeStartService()

        -- For the purpose of this demo, we are letting the OS select port numbers.
        -- So we will use a bezel message to tell the user what thse ports are so they
        -- can connect to them via Telnet (or similar) to send and receive messages.

        if senderPort and receiverPort then

                LrTasks.startAsyncTask( function()

                        -- Give them 10 seconds to connect.

                        for countDown = 10, 1, -1 do

                                if not _G.running then
                                        break
                                end

                                if senderConnected and receiverConnected then
                                        break
                                end

                                local msg = "Connect to port:"

                                if not receiverConnected then
                                        msg = string.format( "%s\nReceiver = %d", msg, receiverPort )
                                end

                                if not senderConnected then
                                        msg = string.format( "%s\nSender = %d", msg, senderPort )
                                end

                                msg = string.format("%s\n%d", msg, countDown )

                                LrDialogs.showBezel( msg, 1 )
                                LrTasks.sleep( 1 )
                        end
                end )
        end
end

--------------------------------------------------------------------------------

local function makeSenderSocket( context )
```

```
	-- A socket connection that sends messages from the plugin to the external process.

	local sender = LrSocket.bind {

		functionContext = context,
		address = "localhost",
		port = sendPort,
		mode = "send",
		plugin = _PLUGIN,

		onConnecting = function( socket, port )
			senderPort = port
			maybeStartService()
		end,

		onConnected = function( socket, port )
			senderConnected = true
		end,

		onMessage = function( socket, message )
				-- Nothing, we don't expect to get any messages back.
		end,

		onClosed = function( socket )
			-- If the other side of this socket is closed,
			-- tell the run loop below that it should exit.
			_G.running = false
		end,

		onError = function( socket, err )
			if err == "timeout" then
				socket:reconnect()
			end
		end,
	}

	-- This object is used to observe Develop parameter changes and
	-- report them all to the sender socket.

	senderObserver = {
		_sender = sender,
	}

	LrDevelopController.addAdjustmentChangeObserver( context, senderObserver, updateDevelopParameters )

	-- do initial update

	updateDevelopParameters( senderObserver )

	return sender
end

--------------------------------------------------------------------------------

local function makeReceiverSocket( context )

	-- A socket connection that receives messages from the external process and executes
	-- commands in Lightroom.

	local receiver = LrSocket.bind {

		functionContext = context,
		port = receivePort,
		mode = "receive",
		plugin = _PLUGIN,

		onConnecting = function( socket, port )
			receiverPort = port
```

```
                maybeStartService()
        end,

        onConnected = function( socket, port )
                receiverConnected = true
        end,

        onClosed = function( socket )
                -- If the other side of this socket is closed,
                -- tell the run loop below that it should exit.
                _G.running = false
        end,

        onMessage = function( socket, message )
                if type( message ) == "string" then

                        local key, value = parseMessage( message )

                        if key and value then

                                if setValue( key, value ) then

                                        -- For the purpose of this demo, also show a bezel.
                                        LrDialogs.showBezel( string.format( "%s %s",
                                                                            tostring( key ),
                                                                            tostring( value ) ), 4 )
                                end
                        end
                end
        end,

        onError = function( socket, err )
                if err == "timeout" then
                        socket:reconnect()
                end
        end,
    }

    -- automatically scroll sliders into view whenever they are adjusted

    LrDevelopController.revealAdjustedControls( true )

    return receiver

end

--------------------------------------------------------------------------------
-- Start everything in an async task so we can sleep in a loop until we are shut down.

LrTasks.startAsyncTask( function()

    -- A function context is required for the socket API below. When this context
    -- is exited all socket connections that have been created from it will be
    -- closed. We stay inside this context indefinitiely by spinning in a sleep
    -- loop until told to exit.

    LrFunctionContext.callWithContext( 'socket_remote', function( context )

        local sender = makeSenderSocket( context )
        local receiver = makeReceiverSocket( context )

        LrDialogs.showBezel( "Controller Demo Running" )

        -- Loop until this plug-in global is set to false, which happens when the external process
        -- closes the socket connection(s), or if the user selects the menu command "File >
        -- Plug-in Extras > Stop" , or when Lightroom is shutting down.

        _G.running = true
```

```
		while _G.running do
			LrTasks.sleep( 1/2 ) -- seconds
		end

		_G.shutdown = true

		if senderConnected then
			sender:close()
		end

		senderObserver = nil

		if receiverConnected then
			receiver:close()
		end

		LrDialogs.showBezel( "Remote Connections Closed", 4 )

	end )

end )
```

Stop.lua

```
--[[----------------------------------------------------------------------------
ADOBE INC.

---------------------

Copyright 2024 Adobe
All Rights Reserved.

NOTICE: Adobe permits you to use, modify, and distribute this file in accordance
with the terms of the Adobe license agreement accompanying it. If you have received
this file from a source other than Adobe, then your use, modification, or distribution
of it requires the prior written permission of Adobe.

------------------------------------------------------------------------------]]
_G.running = false			-- tell the run loop to exit
```

Shutdown.lua

```
--[[----------------------------------------------------------------------------
ADOBE INC.

---------------------

Copyright 2024 Adobe
All Rights Reserved.

NOTICE: Adobe permits you to use, modify, and distribute this file in accordance
with the terms of the Adobe license agreement accompanying it. If you have received
this file from a source other than Adobe, then your use, modification, or distribution
of it requires the prior written permission of Adobe.
------------------------------------------------------------------------------]]
return {
	LrShutdownFunction = function( doneFunction, progressFunction )
		local LrTasks = import "LrTasks"
		LrTasks.startAsyncTask( function()
			if _G.running then
				local LrDate = import "LrDate"
				local start = LrDate.currentTime()
				local estimatedWait = 0.5 -- seconds

				-- tell the run loop to exit
				_G.running = false

				-- wait for the run loop to exit, updating the
```

```
                        -- progress bar to give the user feedback
                        while not _G.shutdown do
                                local now = LrDate.currentTime()
                                local percent = (now - start) / estimatedWait
                                percent = math.min( 1, math.max( 0, percent ) )
                                progressFunction( percent )
                                LrTasks.sleep( 0.1 ) -- seconds
                        end
                end

                -- tell the app we're done and it's safe to shut down
                progressFunction( 1 ) -- 100% complete
                doneFunction()
        end )
    end,
}
```