# EAN-Script Development

**2023-04-27**

Exports: Export Summary Sheet

EULA: End User License Agreement

Web: sightlineapplications.com

Sales: sales@sightlineapplications.com

Support: support@sightlineapplications.com

Phone: +1 (541) 716-5137

1 Overview ..... 1
1.1 Developing On-Board Applications ..... 1
1.1.1 Lua ..... 1
1.1.2 C/C++ ..... 1
1.2 Additional Support Documentation ..... 2
1.3 SightLine Software Requirements ..... 2
2 Example Scripts ..... 2
2.1 Install Directory ..... 2
2.2 Script Summary ..... 2
3 Basic Setup ..... 4
4 Development Environment ..... 5
5 Uploading Scripts ..... 6
6 Enabling / Disabling Scripts ..... 7
6.1 Running Scripts at Startup ..... 7
7 Script Interface ..... 8
7.1 LUA Script and VideoTrack ..... 8
7.2 Create Script Function ..... 8
8 Key Script Interfaces ..... 9
8.1 SLPostAnalyze ..... 9
8.2 SLUnLoad (New in 2.24.xx) ..... 10
8.3 SLLoad (New in 2.24.xx), ..... 11
8.4 SLNewCmdCallback (New in 2.24.xx) ..... 11
9 Using Lua to Initialize Hardware ..... 11
9.1 Create a Lua Script ..... 12
9.2 Load Script to Target Hardware ..... 13
9.3 Configure Acquisition Parameters ..... 13
9.4 Enable Lua Script ..... 13
10 Troubleshooting ..... 14
10.1 Additional Script Debugging ..... 14
10.2 Questions and Additional Support ..... 15
Appendix A - Lens Control Script ..... 15
A1 Overview ..... 15
A2 Testing on SightLine Hardware ..... 15
A3 Basic Troubleshooting ..... 17
A4 Serial Ports ..... 18
A5 Camera Index ..... 18
Appendix B - Reticle Selection Script ..... 18
B1 Overview ..... 18
B2 Configuring Reticle Scripts ..... 18
B3 Reticle Color Mapping ..... 19
B4 Reticle Index ..... 20

**CAUTION:** Alerts to a potential hazard that may result in personal injury, or an unsafe practice that causes damage to the equipment if not avoided.

**IMPORTANT:** Identifies crucial information that is important to setup and configuration procedures.

*Used to emphasize points or reminds the user of something. Supplementary information that aids in the use or understanding of the equipment or subject that is not critical to system use.*

# 1 Overview

This document provides information and steps for developing and running custom scripts in Lua for the 1500-OEM, 3000-OEM, 4000-OEM, and 1750-OEM video processing boards.

## 1.1 Developing On-Board Applications

SightLine provides two primary ways for customers to develop their own on-board applications: C/C++ and Lua. Each technology has benefits and costs for solving a problem. It is impossible to prescribe the right technology for every scenario. This section helps provide general guidelines to assist in understanding the tradeoffs.

### 1.1.1 Lua

Lua is recommended for light-weight applications that need to perform simple data processing and interaction with the onboard video processing application, VideoTrack. Applications such as dynamic on-screen displays based on telemetry data, or simple command and control from serial ports are effective uses for Lua. Lua scripts are executed in-line with our video processing and cannot be synchronized with the processing of video frames. Issues such as increased latency and other performance impacts can arise from Lua scripts that can be complex.

### 1.1.2 C/C++

If an application requires complex data handling, frequent real-time access to IO, or should be run in parallel with VideoTrack, SightLine recommends creating C/C++ applications that can be run on the ARM processor. Information on creating embedded C/C++ applications can be found in EAN-ARM-Development-1500-3000-OEM or EAN-ARM-Development.

When reviewing options, contact Support to discuss your application.

**Table 1: Lua and C/C++ Comparison Table**

| Benefits | Drawbacks |
|---|---|
| **Lua**<br>• Simple to deploy.<br>• Frame synchronized execution.<br>• Can leverage numerous examples from SightLine or the internet. | **Lua**<br>• Not as widely used as C/C++.<br>• Access to IO is complex and difficult.<br>• Real-time debugging is not available.<br>• Networking is not yet supported. |
| **C/C++**<br>• Wide acceptance within the embedded programming industry.<br>• Can be easy to test on a PC before deploying on target hardware.<br>• Real-time debugging.<br>• Complete access to IO, file system, etc.<br>• Can leverage numerous examples from SightLine or the internet.<br>• Can run in parallel to existing applications.<br>• Portable to numerous platforms. | **C/C++**<br>• Deploying application to launch at run time can be error prone, e.g., file location, system permission, etc.<br>• Existing setup procedure is complex[1] (VMWare, CCStudio, mapped drives, NFS booting). |

[1] These tools and procedures are complex but used industry wide with TI embedded systems.

### 1.2 Additional Support Documentation

Additional Engineering Application Notes (EANs) can be found on the Documentation page of the SightLine Applications website.

The Panel Plus User Guide provides a complete overview of settings and dialog windows. It can be accessed from the Help menu of the Panel Plus application.

The Interface Command and Control (IDD) describes the native communications protocol used by the SightLine Applications product line. The IDD is also available as a PDF download on the Software Downloads page.

### 1.3 SightLine Software Requirements

ⓘ **IMPORTANT:** Starting with 3.6.x software and above, only the 4000 and 1700 platforms will be supported. The 1500 and 3000 platforms will continue to be supported in 3.5.x software. Some features in 3.6.x and above may not be available on 1500 and 3000 platforms.

ⓘ **IMPORTANT:** The Panel Plus software version should match the firmware version running on the board. Firmware and Panel Plus software versions are available on the Software Download page.

## 2 Example Scripts

The sample applications/scripts installer (*SLA ARM Examples*) can be downloaded from the SightLine Applications website. Run the installer before setting up the development environment.

*LUA 5.1 is currently supported.*

### 2.1 Install Directory

The example scripts are intended to serve as a starting point for any script development. A summary of each example script is shown below. The example scripts are in *C:\SightLine Applications\SLA-Examples-ARM<<version number>>\SLAScripts\Scripts*.

### 2.2 Script Summary

*Example scripts in the install directory that are used internally are not shown in this list.*

**Table 2: Script Summary**

| Available Scripts | Description |
|---|---|
| **Camera Control** | |
| *cameraControl_Atom1024.lua* | Opens a serial port and sends commands to configure the camera for 8-bit or 14-bit mode. |
| *cameraControl_Boson.lua* | Enables a GPIO to pull camera out of reset then opens a serial port and sends commands to configure the camera for 8-bit or 16-bit mode. |
| *cameraControl_CC3Rst.lua* | Enables a GPIO to pull camera out of reset. |
| *cameraControl_FPC.lua* | Enables serial communications to the camera through the FPC board, which is normally done in the camera *init* code. |
| *cameraControl_GP1Low.lua* | Generic example that enables GPIO for output then toggles the line low. |
| *cameraControl_internal.lua* | Common functionality for use by other scripts, such as querying acquisition parameters, working with GPIO and serial ports. |
| *cameraControl_JAI.lua* | Configures serial port parameters such as baud rate, then sends configuration commands to the camera. |

*(Script Summary table continued)*

| | |
|---|---|
| *cameraControl_Sample.lua* | A template that includes methods on how to use *CameraControlProperties* data structure and serial port communication. |
| *cameraControl_Sony720P60.lua* | Configures serial port parameters such as baud rate, then sends commands to the camera to set it into 720P mode, power the camera off, then on again. This script supported on 3000-OEM only. |
| **Lens Control** | |
| *lensctrl.lua* | A complex script that provides example implementations of auto focus algorithms and other lens control functions. More details can be found in the Appendix A. |
| *ophirctrl_internal.lua* | Interface to the Ophir lens for auto focus. This does not to replace the built-in auto focus on the Ophir, but it provides an example of how to implement an auto focus algorithm. |
| *hitachictrl_internal.lua* | Interface to the Hitachi camera for lens control and focus mechanism. Testing was done with the Hitachi DI-SC123R camera. This does not replace the built-in auto focus on the Hitachi but provides an example of how to implement an auto focus algorithm and sending Hitachi commands over serial to the camera. |
| *sonyctrl_internal.lua* | Interface to the Sony camera for lens control and focus mechanism. Testing was done with the Sony FCB-EH6300 camera. This does not to replace the built-in auto focus on the Sony, but it provides an example of how to implement an auto focus algorithm and sending VISCA commands over serial to the camera. |
| *ophirlens_ctrl.lua* | The Ophir lens example provides support for basic lens control functions like zoom and focus. Users can choose between the example auto focus or native camera implementation. The example implementation is for a one-push auto focus method. It will not continue to track focus through zoom and scene changes. |
| *gpio.lua* | Toggles the GPIO based on MTI detections. See EAN-GPIO-and-I2C for more information on available GPIO. The example highlights how to get telemetry from the SightLine software, and how to set GPIO from a script. |
| *getParams.lua* | Examples of how to query for data/state. |
| *setSystemValue.lua* | This script is an example of how to interpret SetSystemValue (0x92) commands when systemValueType is set for CUSTOMER USE (18). Values 0 - 3 can be either left unused or can contain data that is interpreted by the script. This script can be used to write customer input to a file, change network performance using traffic control (tc), and make the file system writeable (3000-OEM). |
| *slrs232_internal.lua* | Wrapper for SightLine Application Inc. serial port class. |
| *snapFocus.lua* | This script uses focus metric and takes a group of snapshots when the focus is greater than the focus of a past window of frames. It also demonstrates how to kick off a script task using a SightLine command. Focus metric telemetry must be enabled using CoordinateReportingMode (0x0B). More user information is included in comments in the script. |
| **NUC / DPR** | |
| *autoTrackScript.lua* | Example program that toggles GPIO on MTI detections and show how detection can transition to tracking automatically. See EAN-GPIO-and-I2C for more information on available GPIO. This example uses 175, which is valid for the 1500-OEM. |
| *error_internal.lua* | Provides utility for generating user warnings and errors to assist in debugging. |
| *filesystem_example.lua* | An example for how to interact with files (open, close, write, copy). Currently file details (name, and directory location) are set as globals. This code can easily be modified to have these as values that are passed in. |
| *gpioSnapShot.lua* | Example program that takes snapshots on GPIO175 toggle to high then low. See EAN-GPIO-and-I2C for more information on available GPIO on 1500-OEM and 3000-OEM hardware. |
| *helloworld.lua* | Sends a command to VideoTrack to draw a text overlay with the following text: *Hello World*. Removes the overlay when the script is unloaded. |
| *klvOverlay.lua* | Draws KLV overlays on the screen. |
| *klvstatic.lua* | This script pushes static metadata values to VideoTrack for KLV output with streaming video. |
| *mtisnap.lua* | Injects MISB ST 0601 metadata values for Platform Designation (TAG 10), Image Source Sensor (TAG 11), and Platform Tail Number (TAG 4), which are then sent with the MPEG2-TS stream as KLV. |
| *reticles.lua* | Used to generate custom on screen display reticles. There are four distinct types of reticles to choose for each camera. Configuration details for reticles script are loaded from a secondary file. See Appendix B. |

*(Script Summary table continued)*

| | |
|---|---|
| *reticlesConfig_internal.lua* | Support file for *reticles.lua.* |
| *setSystemValue.lua* | This script is an example of how to interpret SetSystemValue (0x92) commands when systemValueType is set for CUSTOMER USE (18). Values 0 - 3 can be either left unused or can contain data that is interpreted by the script. This script can be used to write customer input to a file, change network performance using traffic control (tc), and make file system writeable (3000-OEM). |
| *sla_internal.lua* | Used during development, this script defines the SightLine Command interfaces. Used as an API in IDEs, it is not a script that runs on the target. |
| *snapFocus.lua* | Enables focus telemetry and takes snapshots when the focus is at a maximum over the past window of frames. |
| *snapshot.lua* | Retrieves the current version of SightLine software, starts a track at coordinates (320,240), takes a snapshot, and then prints out tracking positions. |
| *startRecord.lua* | Configures recording destination (uSD),and then starts recording video for Net0 on the 10th frame and ends recording on the 1000th frame. |
| *telemdata.lua* | Sends commands to VideoTrack to display the registration and stabilization telemetry data as on-screen overlays. It also displays the hex codes for any SightLine command messages received by the system. This diagnostic tool provides examples of message parsing. |
| *telemlogger.lua* | Similar to *telemdata.lua.* Instead of sending commands to draw the information on top of the video, telemetry data and messages are logged to a file on the microSD card. |
| *TemperatureDetect.lua* | Logs the telemetry data and messages to a file on the microSD card. |
| *TemperatureDetect_Basic.lua* | Uses pixel statistics to detect ranges of temperature from a calibrated infrared camera. If the criteria are met, it will draw green and red overlay boxes around targets and take a snapshot of the target. |
| *template.lua* | A minimal set of functions that allow it to be called from VideoTrack. Shows how to use error and warning printing. |
| *tracksnap.lua* | Example program that gets the version, starts a track, and takes a snapshot. |
| *windowPTZ.lua* | Example program that draws rectangles on the screen representing the capture and display areas. |

## 3 Basic Setup

The following are the major components of the development and deployment of LUA scripts:

- Any development environment can be used to write scripts. This is usually some form of text editor or more advance Interface Development Environment (IDE).
- The SightLine Firmware upgrade utility is used to upload scripts from the PC to the target hardware.
- SightLine Panel Plus is used to debug the script through error messages. Panel Plus may also be used to see custom graphics or other functions that are being executed by the LUA script.
- The Panel Plus External Programs dialog is used to load and unload scripts. Keeping this dialog open during development allows the developer to easily iterate during script development or try new scripts.

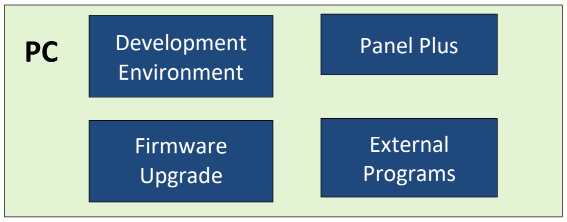


**Figure 1: Common Windows Layout During Script Development**

## 4 Development Environment

Lua scripts can be developed with almost any IDE. However, to get syntax highlighting, static analysis, and autocomplete capabilities, SightLine recommends using the ZeroBrane IDE from ZeroBrane Studio.

*If a script uses another script internally, include the keyword internal in the script file name so it will not show up in the Panel Plus list of programs.*

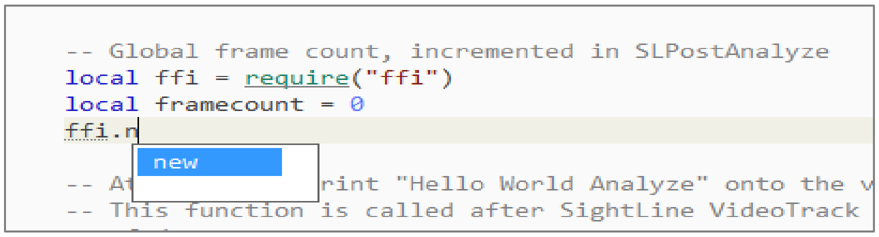
In this example, the application has been installed to *C:\ZeroBraneStudio*. The steps below reference the install location as *ZBS*.

1. Download and install ZeroBrane IDE from ZeroBrane Studio.
2. Copy *sla_internal.lua* from the installed sample directory /SightLine Applications/SLA-Examples-ARM/SLAScripts to the ZBS\api\lua directory. This will define the API used by SightLine Command and Control protocol.
3. Open ZeroBrane Studio and choose the menu option project directory: *Project* » *Project Directory* » *Choose*.
4. Select the directory with the example scripts. *C:\SightLine Applications\SLA-Examples-ARM <<version number>>\SLAScripts\Scripts*. (See Install Directory)
5. Choose the menu option *Edit* » *Preferences* » *Settings:User*. This opens a user settings file. Add the following line to end of the file:

   *api = {'sla_internal'}*

*For SightLine firmware version 2.23.xx add the following line to the end of the file: api = {'sla'} instead.*

6. Restart the ZeroBrane Studio application.
7. The *helloworld.lua* example can now be edited. To verify autocomplete is working, type *ffi.n* after the definition of *framecount*. It should show *new* as an option.



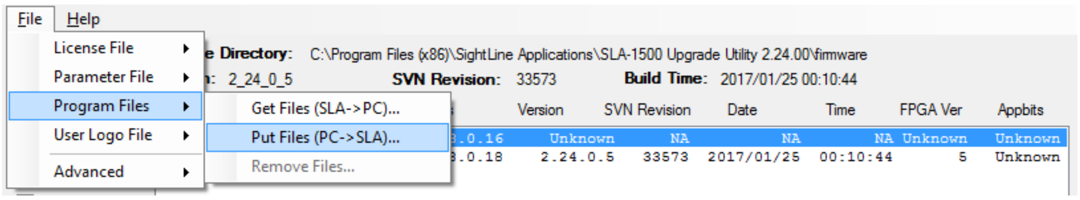
```
-- Global frame count, incremented in SLPostAnalyze
local ffi = require("ffi")
local framecount = 0
ffi.n

-- At           rint "Hello World Analyze" onto the v
-- This function is called after SightLine VideoTrack
```

*ZeroBrane Studio includes a static analyzer. To use this, go to the menu option Project » Analyze after editing a file. This is strongly recommended before sending files to the SightLine hardware. If using a different IDE that does not contain a static analyzer, there are other third-party tools available on the web.*

## 5 Uploading Scripts

User developed scripts can be uploaded to the OEM hardware using the SightLine upgrade utility.

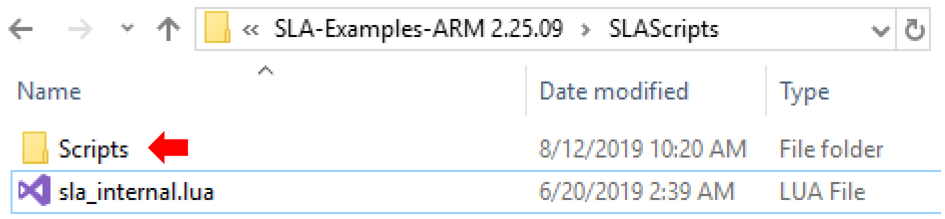
1. Click the *Find IP Address* to get a list of devices on the network.
2. Select the target hardware from the list of devices.
3. From the upgrade utility menu go to *File* » *Program Files* » *Put Files (PC->SLA) …*



e Directory: C:\Program Files (x86)\SightLine Applications\SLA-1500 Upgrade Utility 2.24.00\firmware
n: 2_24_0_5 SVN Revision: 33573 Build Time: 2017/01/25 00:10:44

| | Version | SVN Revision | Date | Time | FPGA Ver | Appbits |
|---|---|---|---|---|---|---|
| 3.0.16 | Unknown | NA | NA | NA | Unknown | Unknown |
| 3.0.18 | 2.24.0.5 | 33573 | 2017/01/25 | 00:10:44 | 5 | Unknown |

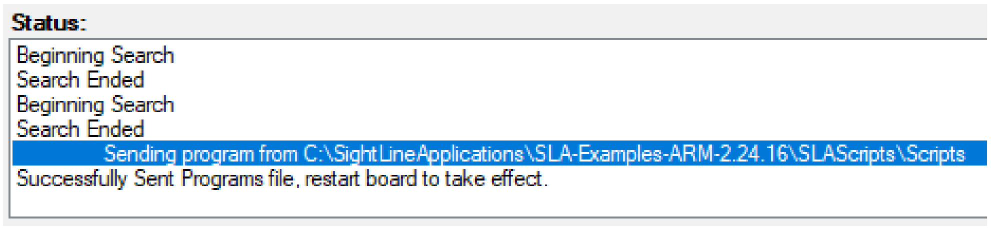
4. Select the directory containing the developed Lua scripts or the *Example* scripts directory. This uploads all the Lua files (*.*lua*) from that directory to the SightLine hardware.



| Name | Date modified | Type |
|---|---|---|
| Scripts | 8/12/2019 10:20 AM | File folder |
| sla_internal.lua | 6/20/2019 2:39 AM | LUA File |

*The upgrade utility can also be used to retrieve all the scripts from the hardware.*

5. Confirm success by checking the *Status* window.



```
Status:
Beginning Search
Search Ended
Beginning Search
Search Ended
        Sending program from C:\SightLineApplications\SLA-Examples-ARM-2.24.16\SLAScripts\Scripts
Successfully Sent Programs file, restart board to take effect.
```

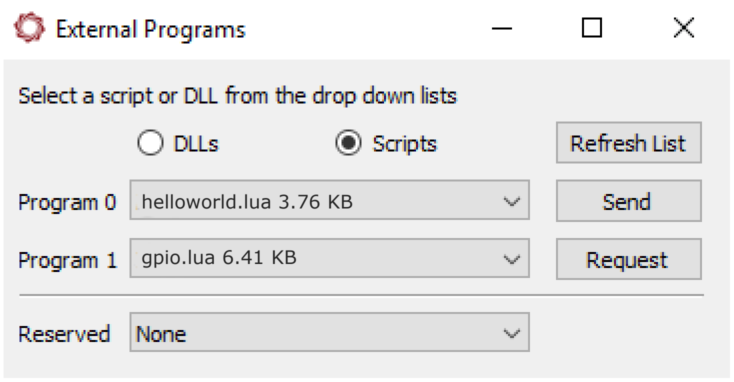
*Ignore the Restart board alert.* Use the *Clear* button if the Status window is too full of message.

6. Once the scripts are uploaded, they must be enabled (see the next section).

## 6 Enabling / Disabling Scripts

Scripts can only run if they are enabled. This can be done through the command and control protocol using the ExternalProgram (0x8F) command. It can also be done from the *External Programs* dialog window in Panel Plus, main menu » *File* » *Programs*.

To enable a script, select it in the drop-down menu and click *Send*. To disable a script, select *None* and click *Send*.


Select a script or DLL from the drop down lists

DLLs [✓ Scripts]

Program 0: helloworld.lua 3.76 KB

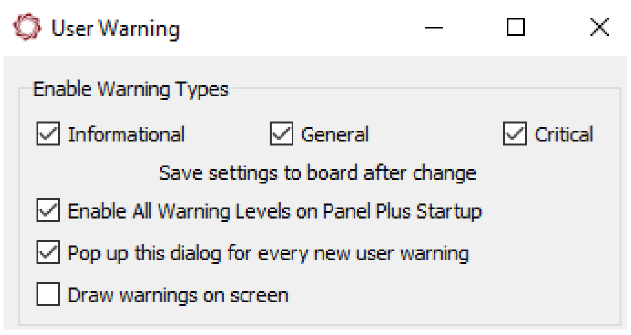
Program 1: gpio.lua 6.41 KB

Reserved: None

**Figure 2: External Programs**

*Sending the message again or clicking the Send button again will cause the scripts to be reloaded. This is especially useful when debugging or incrementally adding functionality to the script.*

**IMPORTANT:** Problems with scripts show up as a user warning. When loading scripts, it is important to monitor the *User Warning* dialog window. From the Panel Plus main menu, go to *View* » *User Warning*.


Enable Warning Types

☑ Informational ☑ General ☑ Critical

Save settings to board after change

☑ Enable All Warning Levels on Panel Plus Startup

☑ Pop up this dialog for every new user warning

☐ Draw warnings on screen

**Figure 3: User Warning Dialog**

### 6.1 Running Scripts at Startup

To run the script at startup, enable the script and then save the settings to the board, main menu » *Parameters* » *Save to board*.

## 7 Script Interface

The script interface to VideoTrack is documented in the IDD. The IDD provides details on structures, functions, and other protocols available through the script interface.

### 7.1 LUA Script and VideoTrack

It is important that understand that the LUA script is running on the same ARM processor as the VideoTrack application. It runs in close coordination with the VideoTrack application.

All the major functions in the LUA script are performed in *callback functions* that are called by VideoTrack in response to events, for example:

- A new SLA command was received by VideoTrack from Panel Plus:

  Forward a copy of this command to the LUA script so it can process the command.
- A new frame of video was acquired/analyzed by VideoTrack:

  Inform the LUA script so that it can do an operation, e.g., start recording based on the number of frames.

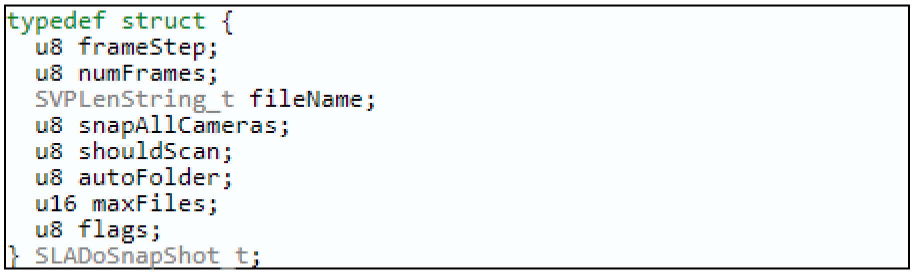
The LUA script can also generate SLA commands and send them to the VideoTrack application. For example, in the autofocus example code the LUA script polls the VideoTrack application to receive the current focus metric. It does this by calling GetParameters (0x28) with a FocusStats (0x55) ID, and then waits for a response from video track, which contains the focus metric.

It is important to understand that VideoTrack does not respond to some commands in certain cases, e.g., if the LUA script is controlling a lens through a LUA serial port, then the VideoTrack application will not respond to a CurrentLensStatus (0x6D) command (to get focus and zoom position). This is because VideoTrack application is not currently controlling a lens and cannot respond to the command.

### 7.2 Create Script Function

The following example describes how to create a script function/definition from the IDD. Figure 4 shows how the DoSnapShot (0x60) command is defined in the IDD.

*This may change with software versions.*



```
typedef struct {
  u8 frameStep;
  u8 numFrames;
  SVPLenString_t fileName;
  u8 snapAllCameras;
  u8 shouldScan;
  u8 autoFolder;
  u16 maxFiles;
  u8 flags;
} SLADoSnapShot_t;
```

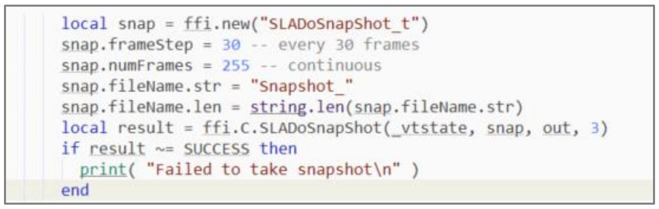
**Figure 4: DoSnapShot (0x60) Command Example in IDD**

All variable names and valid values are called out in the IDD. Variable names are case sensitive. Optional values (varies) do not need to be defined.

Message ID 0x60

| Byte Offset | Name | Description |
|---|---|---|
| 4 | *frameStep* | Frame Step - step between frames, e.g., 2 = every other. |
| 5 | *numFrames* | Number of snapshots to take (1 to 254), 255 = continuous, 0 = Stop. Ignored if snapAllCameras is used. |
| 6 | *Filename.len* | String length |
| 7-… | *Filename.str* | Base file name of saved files. |
| varies | *snapAllCameras* | Mask of cameras to snap, e.g., use 0x5 to snap cameras 0 and 2. For multicamera, only single snapshot is allowed. |
| varies | *shouldScan* | When using file auto-numbering. Begin file numbering after highest-numbered existing filename. |
| varies | *autoFolder* | Creates new files every maxFiles. |
| varies | *maxFiles* | Maximum number of files per folder (2000 is default, 20000 is max). The folder may auto increment sooner under certain conditions, e.g., snapshots being taken too fast. |

Using the above information, the Lua code segment shown in Figure 5 can be created to take a snapshot.



```
local snap = ffi.new("SLADoSnapShot_t")
snap.frameStep = 30 -- every 30 frames
snap.numFrames = 255 -- continuous
snap.fileName.str = "Snapshot_"
snap.fileName.len = string.len(snap.fileName.str)
local result = ffi.C.SLADoSnapShot(_vtstate, snap, out, 3)
if result ~= SUCCESS then
  print( "Failed to take snapshot\n" )
end
```

**Figure 5: Lua Code Segment for Snapshot Command**

## 8 Key Script Interfaces

Key script interfaces are shown in Table 3. These methods are called by VideoTrack when executing a Lua script.

**Table 3: Lua Script Interfaces**

| Script Interfaces | Software Version |
|---|---|
| SLUnLoad | 2.24.xx |
| SLLoad | 2.24.xx |
| SLNewCmdCallback | 2.24.xx |
| SLPostAnalyze | 2.23.xx |

### 8.1 SLPostAnalyze

This function is called by VideoTrack immediately after the analysis of a frame is complete. The analysis step includes registration, tracking, detection, etc. All scripts should implement this function.

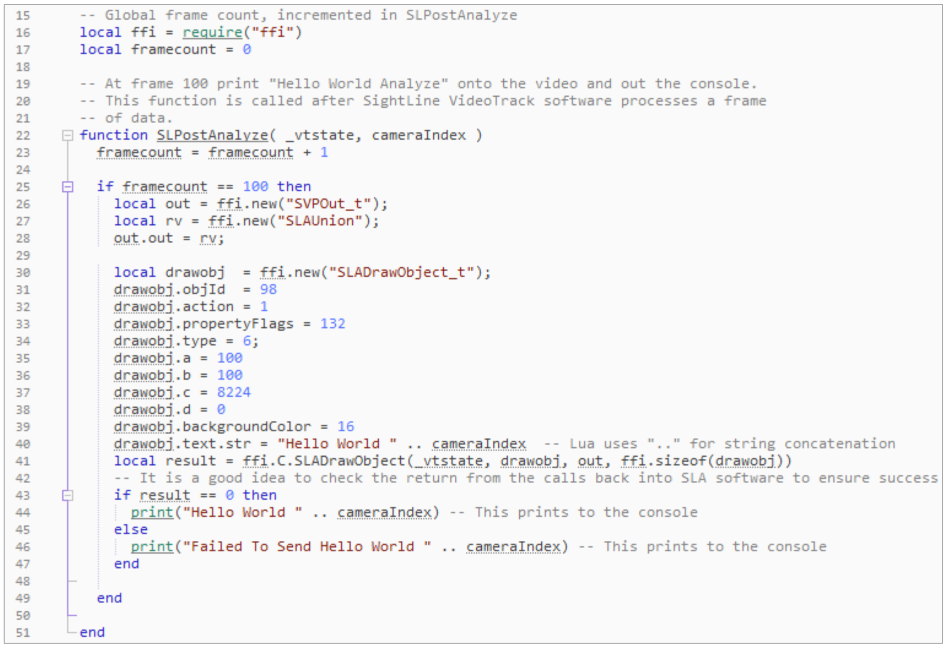
*SLPostAnalyze function is not called if a camera is disconnected or powered off (No Video Source Available message).*

In the *helloworld.lua* example, this function maintains a global framecount, and then uses it to determine when to draw *Hello World* on the video. This function is passed two parameters:

- *_vtstate*: This is a handle to the VideoTrack context and is needed in calls back to the VideoTrack program such as *ffi.C.SLADrawObject* shown in the example.

- *cameraIndex*: Provides the camera index for which the script is being called. Allows users to take actions on a specific camera or keep camera specific data separate.

*On the 3000-OEM, 4000-OEM and 1750-OEM the SLPostAnalyze function is called for all actively processed cameras. It is up to the user to decide how to use the cameraIndex. For example, to draw Hello World for camera two, return at the top of SLPostAnalyze if the camera index is not equal to 2.*



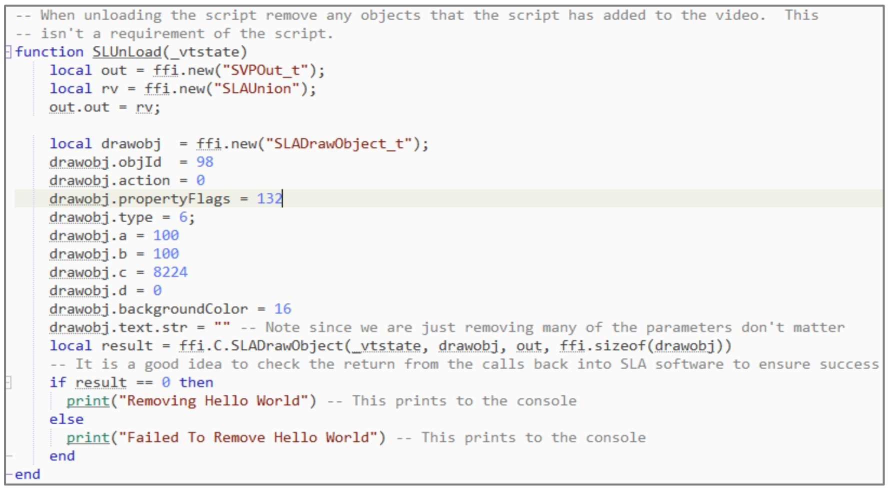
```
-- Global frame count, incremented in SLPostAnalyze
local ffi = require("ffi")
local framecount = 0

-- At frame 100 print "Hello World Analyze" onto the video and out the console.
-- This function is called after SightLine VideoTrack software processes a frame
-- of data.
function SLPostAnalyze( _vtstate, cameraIndex )
  framecount = framecount + 1

  if framecount == 100 then
    local out = ffi.new("SVPOut_t");
    local rv = ffi.new("SLAUnion");
    out.out = rv;

    local drawobj  = ffi.new("SLADrawObject_t");
    drawobj.objId  = 98
    drawobj.action = 1
    drawobj.propertyFlags = 132
    drawobj.type = 6;
    drawobj.a = 100
    drawobj.b = 100
    drawobj.c = 8224
    drawobj.d = 0
    drawobj.backgroundColor = 16
    drawobj.text.str = "Hello World " .. cameraIndex  -- Lua uses ".." for string concatenation
    local result = ffi.C.SLADrawObject(_vtstate, drawobj, out, ffi.sizeof(drawobj))
    -- It is a good idea to check the return from the calls back into SLA software to ensure success
    if result == 0 then
      print("Hello World " .. cameraIndex) -- This prints to the console
    else
      print("Failed To Send Hello World " .. cameraIndex) -- This prints to the console
    end

  end

end
```

**Figure 6: SLPostAnalyze Usage from Hello World Example**

## 8.2 SLUnLoad (New in 2.24.xx)

This function is called by VideoTrack whenever the script is disabled. This can happen if the board is shutting down or loading a new script. In the *helloworld.lua* example, this function removes the drawObject with id 98 created in *SLPostAnalyze.* This function is passed only one parameter:

_vtstate: This is a handle to the VideoTrack context and is needed in calls back to the VideoTrack program such as *ffi.C.SLADrawObject* shown in the example.



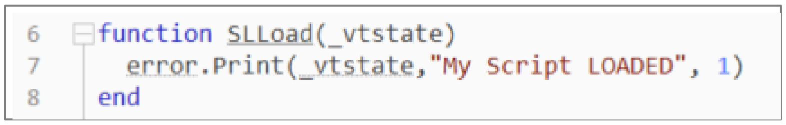
```
-- When unloading the script remove any objects that the script has added to the video.  This
-- isn't a requirement of the script.
function SLUnLoad(_vtstate)
    local out = ffi.new("SVPOut_t");
    local rv = ffi.new("SLAUnion");
    out.out = rv;

    local drawobj  = ffi.new("SLADrawObject_t");
    drawobj.objId  = 98
    drawobj.action = 0
    drawobj.propertyFlags = 132
    drawobj.type = 6;
    drawobj.a = 100
    drawobj.b = 100
    drawobj.c = 8224
    drawobj.d = 0
    drawobj.backgroundColor = 16
    drawobj.text.str = "" -- Note since we are just removing many of the parameters don't matter
    local result = ffi.C.SLADrawObject(_vtstate, drawobj, out, ffi.sizeof(drawobj))
    -- It is a good idea to check the return from the calls back into SLA software to ensure success
    if result == 0 then
      print("Removing Hello World") -- This prints to the console
    else
      print("Failed To Remove Hello World") -- This prints to the console
    end
end
```

**Figure 7: SLUnLoad Usage from Hello World Example**

### 8.3 SLLoad (New in 2.24.xx),

This function is called by VideoTrack when loading the script.

*_vtstate*: This is a handle to the VideoTrack context and is needed in calls back to the VideoTrack program.



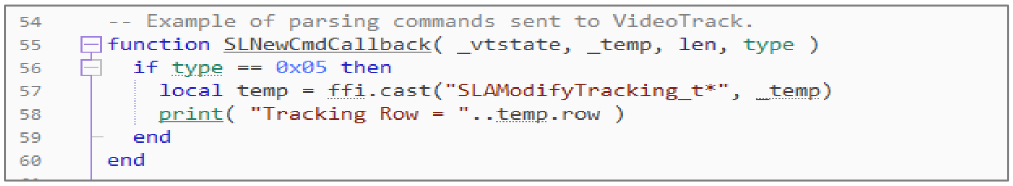
```lua
function SLLoad(_vtstate)
  error.Print(_vtstate,"My Script LOADED", 1)
end
```

**Figure 8: SLLoad Example**

### 8.4 SLNewCmdCallback (New in 2.24.xx)

This function is called by VideoTrack when it receives a command. The following four parameters are passed to this function:

- *_vtstate:* This is a handle to the VideoTrack context and is needed in calls back to the VideoTrack program.
- *_temp:* This is a handle to the structure containing the message data.
- *len:* The length of the message data.
- *type:* This is the message type.



```lua
    -- Example of parsing commands sent to VideoTrack.
function SLNewCmdCallback( _vtstate, _temp, len, type )
  if type == 0x05 then
    local temp = ffi.cast("SLAModifyTracking_t*", _temp)
    print( "Tracking Row = "..temp.row )
  end
end
```

**Figure 9: SLNewCmdCallback Example**

## 9 Using Lua to Initialize Hardware

Lua scripts can also be used to perform additional operations during the initialization of cameras. This is achieved by setting a custom option in the Acquisition Settings dialog window in Panel Plus and loading a Lua script that contains a special function that is called by VideoTrack during initialization.

These can also be set using the SetAcquisitionParameters (0x37) and the ExternalProgram (0x8F) commands in the IDD.

Additional operations may include toggling GPIO to reset a camera, sending commands over the serial port or i2c bus, or other functions that are required to successfully configure and use the camera.

**Process summary**:

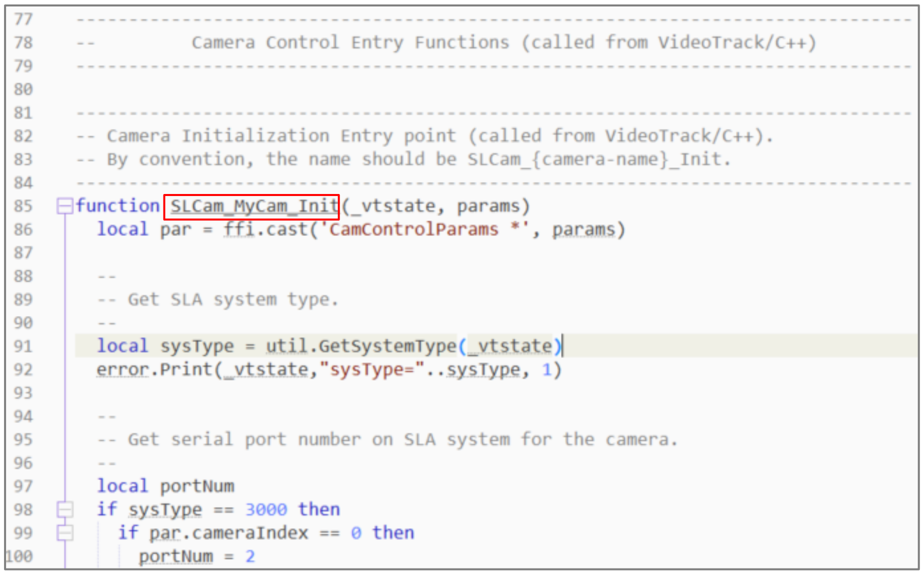
The following process is explained in the subsequent sections below.

- Create a Lua script that contains a custom initialization routine.
- Upload Lua scripts to the target OEM hardware using the SightLine upgrade utility.
- Configure the acquisition parameters in Panel Plus.
- Enable the Lua script with Panel Plus.

## 9.1 Create a Lua Script

The Lua script will contain a special function that will get called during the camera initialization process. The function name will be specified as a string in the options argument of the acquisition parameters. The function name should follow the rules below. The camera initialization function can exist as part of an existing Lua script or be standalone.

*For an example see cameraControl_Sample.lua script.*



```
----------------------------------------------------------------------------
--         Camera Control Entry Functions (called from VideoTrack/C++)
----------------------------------------------------------------------------

----------------------------------------------------------------------------
-- Camera Initialization Entry point (called from VideoTrack/C++).
-- By convention, the name should be SLCam_{camera-name}_Init.
----------------------------------------------------------------------------
function SLCam_MyCam_Init(_vtstate, params)
  local par = ffi.cast('CamControlParams *', params)

  --
  -- Get SLA system type.
  --
  local sysType = util.GetSystemType(_vtstate)
  error.Print(_vtstate,"sysType="..sysType, 1)

  --
  -- Get serial port number on SLA system for the camera.
  --
  local portNum
  if sysType == 3000 then
    if par.cameraIndex == 0 then
      portNum = 2
```

Figure 10: Camera Initialization - cameraControl_Sample.lua script

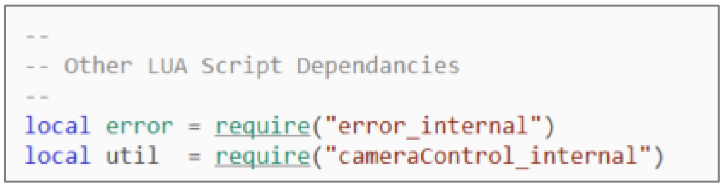
The name of the camera initialization function should use the following pattern:

`SLCam_<<CameraName>>_Init`

*<<CameraName>> can be the name or model of the camera or any other string identifier that is relevant.*

*The <<CameraName>> part of the function name will be used in the options argument to the acquisition parameters.*

Common functionality such as querying for information, serial port communication and GPIO, can be found in the *cameraControl_internal.lua* script. This script will need to be uploaded to the target hardware along with the camera specific Lua script.



```
--
-- Other LUA Script Dependancies
--
local error = require("error_internal")
local util  = require("cameraControl_internal")
```

Figure 11: Including Common Functionality from Another Lua Script

## 9.2 Load Script to Target Hardware

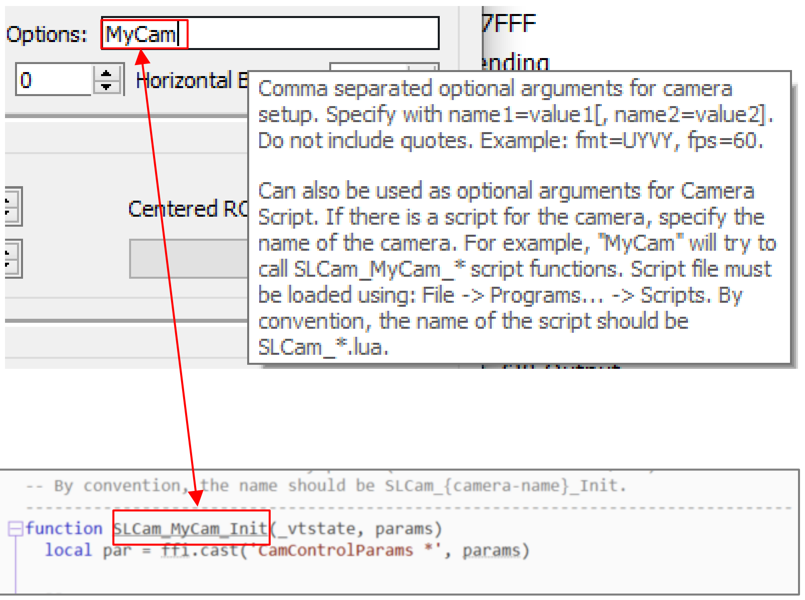
Follow the existing procedures in the Uploading Scripts section to move the script to the target hardware using the SightLine firmware upgrade utility.

## 9.3 Configure Acquisition Parameters

The name of the camera initialization function in the previous section can now be set as an option argument in the *Options* field of the *Acquisition Settings* dialog in Panel Plus.

1. Form the main menu in Panel Plus » *Configuration* » *Acquisition Settings*.
2. In the *Options* field, enter the *<<CameraName>>* used in the Lua script function. This can be combined with other options separated by a comma.

*The acquisition options string matches function in Lua script as shown in Figure 12.*



Comma separated optional arguments for camera setup. Specify with name1=value1[, name2=value2]. Do not include quotes. Example: fmt=UYVY, fps=60.

Can also be used as optional arguments for Camera Script. If there is a script for the camera, specify the name of the camera. For example, "MyCam" will try to call SLCam_MyCam_* script functions. Script file must be loaded using: File -> Programs... -> Scripts. By convention, the name of the script should be SLCam_*.lua.

```
  -- By convention, the name should be SLCam_{camera-name}_Init.
  ------------------------------------------------------------------
function SLCam_MyCam_Init(_vtstate, params)
  local par = ffi.cast('CamControlParams *', params)
```

**Figure 12: Options Field - Panel Plus Acquisition Settings Dialog**

## 9.4 Enable Lua Script

Follow the existing procedures in the Enabling / Disabling Scripts to move the script to the target hardware using the *External Programs* dialog window in Panel Plus.

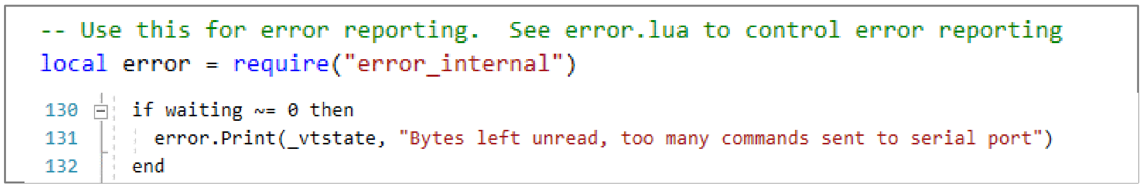
When the system reboots the Lua script will be loaded and the initialization function will be called during the camera acquisition setup procedure.

## 10 Troubleshooting

| Issue | Recommendation |
|---|---|
| Script does not run. | From the Panel Plus main menu, go to *View* » *User Warnings* and enable the user warnings. Start the script and monitor the User Warning dialog window to determine if there are any User Warning preventing the script from running. |
| Cannot debug script. | Print statements are helpful but require access to the Linux console to see the output. This requires a serial connection as well as modifying the silent argument in u-boot on the 1500-OEM and 3000-OEM. Contact support if you have further questions on how to do this. Additionally, debug statements can be drawn to the screen in the same manner using the *helloworld.lua* example. |
| System is sluggish after loading script. | From the Panel Plus main menu, go to *View* » *Performance Graphs*. Select *Enable System Status* and *CPU Timing*. After a few seconds *ARM* should appear in the list of timings. Expand this and look at the time for *Post AnalyzeScript*. The times presented are (*Average, Minimum, Maximum*) over 100 samples in microseconds. If these numbers are large, e.g., several thousand microseconds, reevaluate the level of complexity of the script. |
| 64-bit integer data types. | Add *ULL* to the end of 64-bit integer constants, e.g., *setTime.utcTime = 0x54deab2bd7500ULL* *The Lua interpreter can crash without adding ULL.* |

### 10.1 Additional Script Debugging

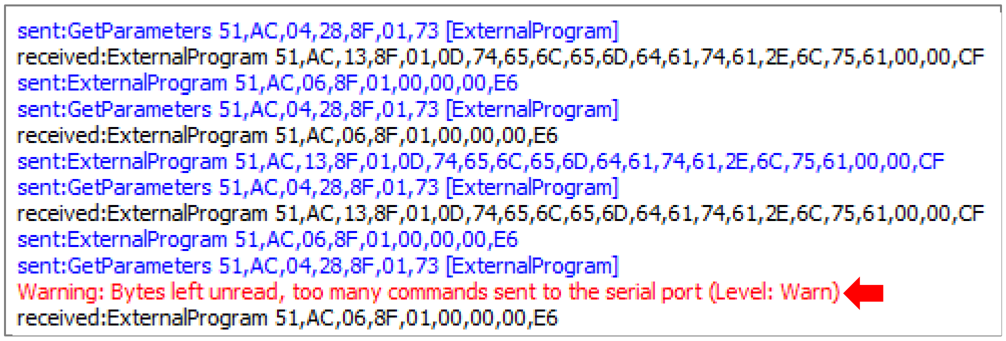
To display diagnostics messages to the Panel Plus window, use the SightLine warning messages. Refer to the *error_internal.lua* script. An example of using error reporting can be found in *hitachi_internal.lua* shown in Figure 13.



```
-- Use this for error reporting.  See error.lua to control error reporting
local error = require("error_internal")

if waiting ~= 0 then
  error.Print(_vtstate, "Bytes left unread, too many commands sent to serial port")
end
```

Figure 13: User Error Reporting Script Example

The output is then displayed in Panel Plus. Right-click in the command area and choose *Clear Text* to clear the error of extraneous errors while testing.



```
sent:GetParameters 51,AC,04,28,8F,01,73 [ExternalProgram]
received:ExternalProgram 51,AC,13,8F,01,0D,74,65,6C,65,6D,64,61,74,61,2E,6C,75,61,00,00,CF
sent:ExternalProgram 51,AC,06,8F,01,00,00,00,E6
sent:GetParameters 51,AC,04,28,8F,01,73 [ExternalProgram]
received:ExternalProgram 51,AC,06,8F,01,00,00,00,E6
sent:ExternalProgram 51,AC,13,8F,01,0D,74,65,6C,65,6D,64,61,74,61,2E,6C,75,61,00,00,CF
sent:GetParameters 51,AC,04,28,8F,01,73 [ExternalProgram]
received:ExternalProgram 51,AC,13,8F,01,0D,74,65,6C,65,6D,64,61,74,61,2E,6C,75,61,00,00,CF
sent:ExternalProgram 51,AC,06,8F,01,00,00,00,E6
sent:GetParameters 51,AC,04,28,8F,01,73 [ExternalProgram]
Warning: Bytes left unread, too many commands sent to the serial port (Level: Warn)
received:ExternalProgram 51,AC,06,8F,01,00,00,00,E6
```

Figure 14: User Error Reporting Example in Panel Plus

### 10.2 Questions and Additional Support

For questions and additional support, please contact Support. Additional support documentation and Engineering Application Notes (EANs) can be found on the Documentation page of the SightLine Applications website.

## Appendix A - Lens Control Script

New in software version 2.24.xx.

### A1 Overview

The lens control example scripts show how to implement a basic auto focus algorithm in Lua. They also show how to receive SightLine lens commands sent from Panel Plus or another user interface and translate those commands into vendor specific commands such as zoom or focus.

These examples also show how to interface with a serial port from Lua, how to do bitwise operations in Lua, plus many other helpful functions. This script was not designed to provide an out-of-the box auto focus algorithm that is plug-and-play for any system.

**IMPORTANT:** Use Sony or Hitachi cameras with a built-in lens.

To switch from the Sony to the Hitachi camera, edit the *lensctrl.lua* script to require the *hitachictrl_internal.lua* script as shown in Figure A1.

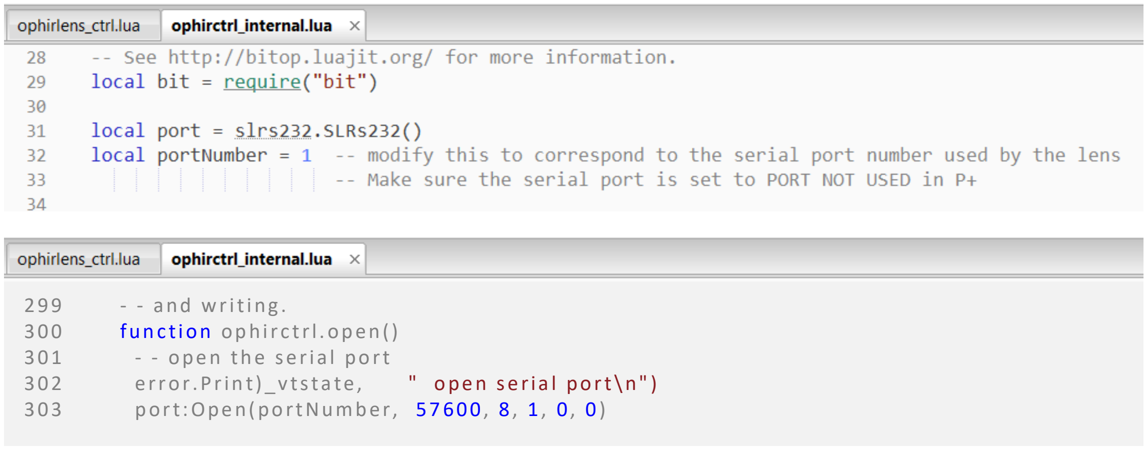
```
-- Use the correct "Lens/Camera" interface here. Currently the options are
-- "hitachictrl" and “sonyctrl". At this point this should be the only thing you
-- need to change to go between the Sony and Hitachi cameras.
local lensctrl = require(“hitachictrl_internal”)
```

**Figure A1: Edit lensctrl.lua Script**

### A2 Testing on SightLine Hardware

1. Modify existing scripts as needed on the PC to match the serial port settings used by the lens.

*Other settings such as encoder positions can also be changed as needed to match the properties of the lens.*



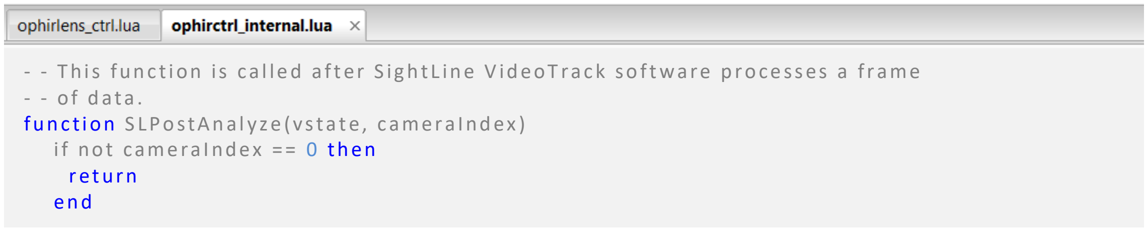
```
-- See http://bitop.luajit.org/ for more information.
local bit = require("bit")

local port = slrs232.SLRs232()
local portNumber = 1  -- modify this to correspond to the serial port number used by the lens
                      -- Make sure the serial port is set to PORT NOT USED in P+

```

```
-- and writing.
function ophirctrl.open()
  -- open the serial port
  error.Print)_vtstate,    " open serial port\n")
  port:Open(portNumber, 57600, 8, 1, 0, 0)
```

**Figure A2: Modify Scripts to Match Lens Serial Port Settings**

2. Modify the script to match the camera index that the lens control is attached to.

   Add or modify code to exit early from the *SLPostAnalyze* function for the non-lens-controlled camera. For example, if *Cam0* is being used then exclude everything that is not *cameraIndex zero (0).*



```
ophirlens_ctrl.lua    ophirctrl_internal.lua

-- This function is called after SightLine VideoTrack software processes a frame
-- of data.
function SLPostAnalyze(vstate, cameraIndex)
    if not cameraIndex == 0 then
      return
    end
```

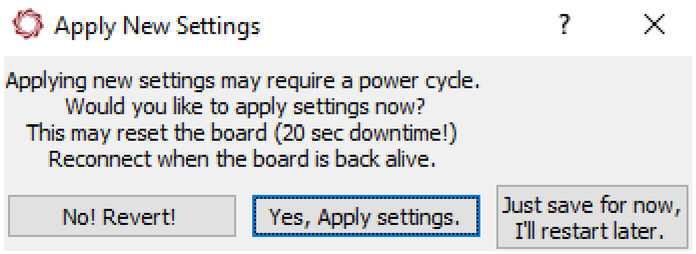
**Figure A3: Exit from SLPostAnalyze Function - Multiple Cameras**

3. Upload the scripts to the target hardware.
4. Using Panel Plus, configure the system to stream video from the digital camera.
5. Enable the lens script.
6. Configure the serial port as *Port Not Used*. For example, from the main menu go to *Configure* » *Serial Ports* and verify that *Serial Port 2* protocol is configured as *Port Not Used*.

*See the Serial Ports section when using VIN1 on the 3000-OEM or Cam 1 on the 4000-OEM or Cam1 on the 1750-OEM. See the EAN-Ethernet-and-Serial-Communication document for more information on configuring serial ports.*

7. To save the configuration to the parameter file, from the Panel Plus main menu » *Parameters* » *Save to board*.

*In 3.01.xx and earlier software versions, saving the Serial Port settings will prompt an additional dialog window. Some setting changes require the board to be restarted for the settings to take effect. In the Apply New Settings dialog window, select an option to save the port configuration.*

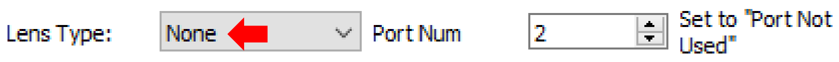


Apply New Settings

Applying new settings may require a power cycle.
Would you like to apply settings now?
This may reset the board (20 sec downtime!)
Reconnect when the board is back alive.

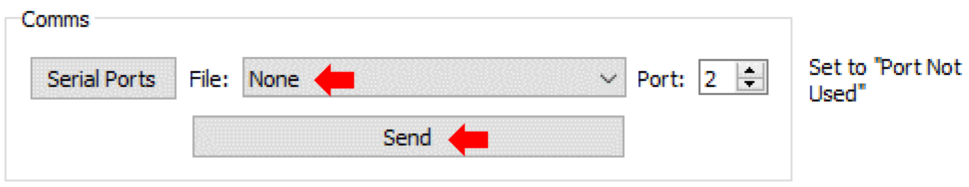
**Figure A4: Apply New Settings Dialog - 3.01.xx and Earlier**

8. When the system has rebooted, open the *Lens* tab in Panel Plus:
   a. In 3.4.x and earlier software versions verify that the *Lens Type* is set to *None*. If not, make the change, save parameters, and then reset the board.

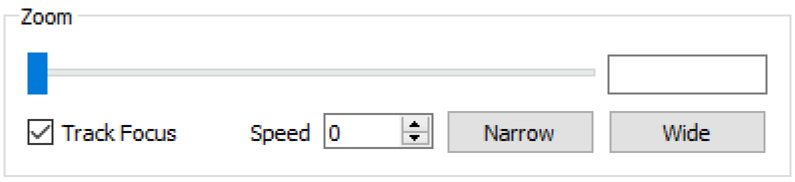


Lens Type: None    Port Num 2    Set to "Port Not Used"

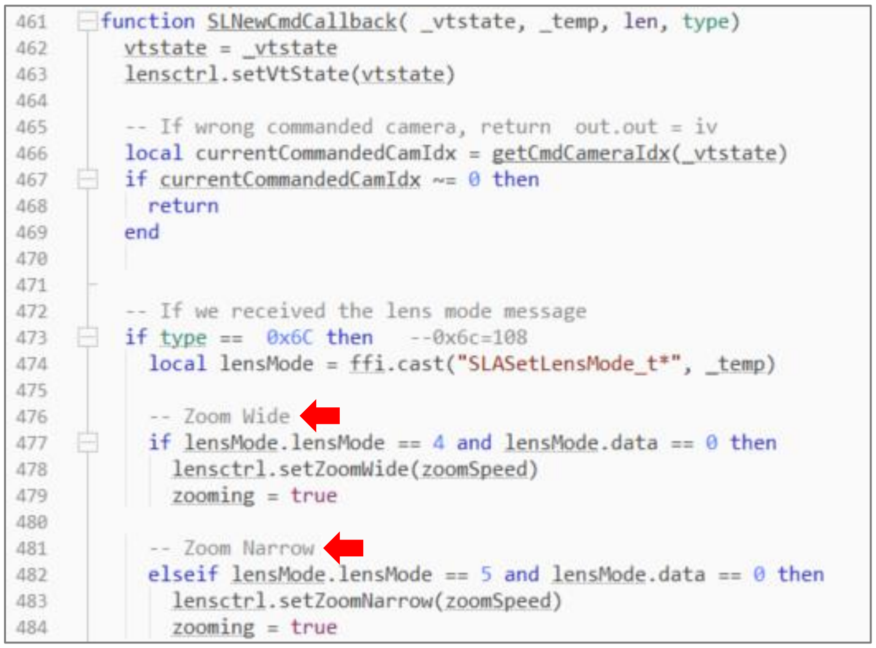
b. In 3.5.x software and above, verify that *File* in the *Comms* section is set to *None*. If not, make the change, click *Send*, save parameters, and then reset the board.



9. Verify functionality by using the *Narrow* or *Wide* zoom controls.



*Not all lens mode controls are implemented in every Lua example. To see the controls that are enabled in the Lua script, review the SLNewCmdCallback function implementation.* For example, *SLNewCmdCallback implements specific lens mode operations such as Zoom shown in Figure A5.*



```
function SLNewCmdCallback( _vtstate, _temp, len, type)
  vtstate = _vtstate
  lensctrl.setVtState(vtstate)

  -- If wrong commanded camera, return  out.out = iv
  local currentCommandedCamIdx = getCmdCameraIdx(_vtstate)
  if currentCommandedCamIdx ~= 0 then
    return
  end


  -- If we received the lens mode message
  if type ==  0x6C then    --0x6c=108
    local lensMode = ffi.cast("SLASetLensMode_t*", _temp)

    -- Zoom Wide
    if lensMode.lensMode == 4 and lensMode.data == 0 then
      lensctrl.setZoomWide(zoomSpeed)
      zooming = true

    -- Zoom Narrow
    elseif lensMode.lensMode == 5 and lensMode.data == 0 then
      lensctrl.setZoomNarrow(zoomSpeed)
      zooming = true
```

**Figure A5: SLNewCmdCallback Implementation - Zoom**

## A3 Basic Troubleshooting

Monitor the user warnings in Panel Plus. If there are no user warnings and the camera is not responding, try different baud rates. Common baud rates for the Hitachi are 4800 and 9600. Common baud rates for the Sony are 9600 and 19200.

If the camera is still not responding, contact Support.

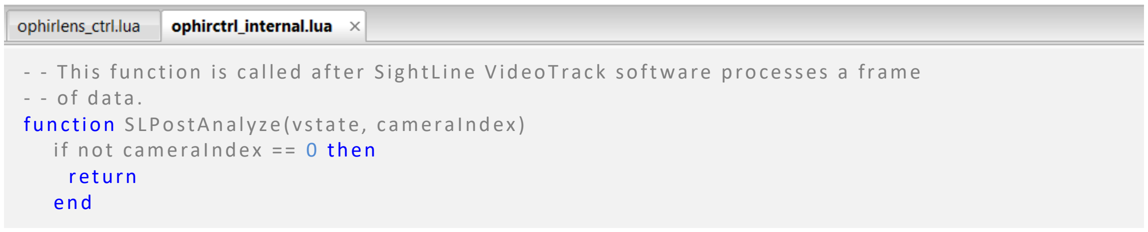
## A4 Serial Ports

On the 3000-OEM if the camera is connected on VIN0, no changes are needed. If the camera is connected on VIN1, the scripts that open the serial ports should be updated to *Serial Port 3* instead of *Serial Port 2*. See *sonyctrl_internal.lua* or *hitachictrl_internal.lua* in the Script Summary table.

On the 4000-OEM or 1750-OEM, if using *Cam 1* instead of *Cam 0,* update the serial ports to *Serial Port 6* instead of *Serial Port 2*.

## A5 Camera Index

If multiple cameras are being processed on the 3000-OEM or 4000-OEM or 1750-OEM, add code to exit early from the *SLPostAnalyze* function for the non-lens-controlled camera. For example, if *Cam0* is being used then exclude everything that is not *cameraIndex zero (0).*


ophirlens_ctrl.lua | ophirctrl_internal.lua

```
- - This function is called after SightLine VideoTrack software processes a frame
- - of data.
function SLPostAnalyze(vstate, cameraIndex)
   if not cameraIndex == 0 then
     return
   end
```

Figure A6: Exit from SLPostAnalyze Function - Multiple Cameras

The example script does not check the commanded camera. To create auto focus scripts for two different cameras, additional code should be added to check the camera in the *SLNewCmdCallback* function.

# Appendix B - Reticle Selection Script

## B1 Overview

Reticle selection scripts can be used to draw overlay reticles on top of the stream and can be customized for each camera. Two script files are available:

- *ReticlesConfig_internal.lua*: Configuration parameters to customize the display of reticles.
- *Reticles.lua*: Adds reticles to the display image. This is the primary script file that should be loaded to the target to display the reticles on the screen.

## B2 Configuring Reticle Scripts

ⓘ **IMPORTANT:** Make sure to edit *reticlesConfig_internal.lua* with a text editor before uploading *reticles.lua* to the hardware.

**Table B1: User Configuration Parameters**

| Parameter | Description |
|---|---|
| CAM<X>_RETICLE_INDEX | Reticle to draw for camera with index X. See Reticle index section for available options. |
| CAM<X>_FIELD_OF_VIEW | Field of view of camera with index X (degrees). |
| CAM<X>_CIRCLE_DEGREE | Diameter of circle (degree) of camera with index X. |
| CAM<X>_CENTER_CIRCLE_RADIUS | Radius of the center point (circle) of camera with index X, default is 2. |
| CAM<X>_COLOR_FOREGROUND | Foreground color (Color 1) to draw for camera with index X. Check Reticle Color mapping section for options. |
| CAM<X>_COLOR_BACKGROUND | Background color (Color 2) to draw for camera with index X. Check Reticle color mapping section for options. |

**Example setting for Camera 1:**

*CAM1_RETICLE_INDEX = 2*

– set camera 1 to use reticle with index 2

*CAM1_COLOR_FOREGROUND = 13*

– set camera 1 foreground color as yellow

*CAM1_COLOR_BACKGROUND = 12*

– set camera 1 background color to orange

*CAM1_FIELD_OF_VIEW = 50*

– set camera 1 field of view to 50 degrees

*CAM1_CIRCLE_DEGREE = 10*

– set camera 1 field of view to 10 degrees

*CAM1_CENTER_CIRCLE_RADIUS = 2*

– set camera 1 center point radius to 2

Field of View**:** Field of view of camera (degrees) is used to calculate the size of lines used to draw the overlays.

Circle Degree**:** Diameter of Circle (degree) relative to the field of view.

Center Circle Radius**:** Center point used in some reticles. Diameter of the circle in degrees.

## B3 Reticle Color Mapping

Reticle color mapping to be used for background and foreground color settings in the configuration file.

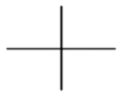
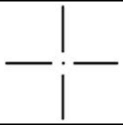
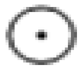
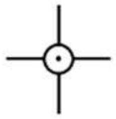
**Table B2: Reticle Color Mapping**

| Color | Value | Color | Value |
|---|---|---|---|
| WHITE | 0 | DARK BLUE | 7 |
| BLACK | 1 | LIGHT GREEN | 8 |
| LIGHT GRAY | 2 | GREEN | 9 |
| GRAY | 3 | DARK GREEN | 10 |
| DARK GRAY | 4 | RED | 11 |
| LIGHT BLUE | 5 | ORANGE | 12 |
| BLUE | 6 | YELLOW | 13 |

## B4 Reticle Index

There are four distinct types of reticles available now with index ranging from 0 to 3. Each camera index is assigned a single reticle. To choose a different reticle for a certain camera change corresponding camera reticle index. For example, to change camera 2 to use reticle with index 0 change *CAM2_RETICLE_INDEX = 0*. All the reticles will be drawn with the same color combination based on *COLOR_TO_DRAW* value.

**Table B3: Reticle Indexes**

| Reticle Index | Picture | Reticle Index | Picture |
|---|---|---|---|
| 0 |  | 2 |  |
| 1 |  | 3 |  |