G940LEDControl Documentation

Release

Mark van Renswoude

Aug 13, 2017

Contents:

1 Introduction **1**
- 1.1 Source code . . . 1

2 Scripting **3**
- 2.1 Script locations . . . 3
- 2.2 Anatomy of a button function . . . 3

3 Scripting reference **7**
- 3.1 Global functions . . . 7
- 3.2 Global variables . . . 12

CHAPTER 1

Introduction

G940LEDControl allows you to bind functions to the throttle's buttons which control the LEDs. Each function may contains several states, each of which can be configured for a certain colour and/of flashing pattern.

Since version 2.0 all functions are implemented using Lua scripts. This means it is fairly easy to create customized versions of the standard functions, or even add a completely new function. For more information, see the page on *Scripting* or dive straight into the *Scripting reference*.

Source code

Since version 2.0, G940LEDControl is released as open-source under the GNU General Public License v3.0. The main Git repository is located at https://git.x2software.net/delphi/g940ledcontrol with a clone being kept up to date at https://github.com/PsychoMark/G940LEDControl.

G940LEDControl is compiled using Delphi XE2. The following additional libraries are required:

- OmniThreadLibrary (1.0.4)
- VirtualTreeView (5.3.0)
- X2Log
- X2Utils

Newer versions Delphi and/or of the libraries might work as well, though have not been tested yet.

A copy of DelphiLua is included in the G940LEDControl repository.

CHAPTER 2

Scripting

All functionality introduced by G940LEDControl is described in the *Scripting reference*. For more information about Lua in general, please refer to the Lua 5.2 Reference Manual.

This guide will walk through how Lua scripting works in G940LEDControl. To avoid confusion when talking about functions in this context they will be referred to as 'button functions' and 'Lua functions'.

Script locations

The default scripts included with G940LEDControl can be found in the folder selected during installation, for example "C:\Program Files (x86)\G940 LED Control". In there you will find a Scripts\FSX folder containing the Lua files.

In addition scripts are loaded from your user data path. This folder is automatically created by G940LEDControl. To open the folder, type or paste "%APPDATA%\G940LEDControl\Scripts\FSX" into a Windows Explorer address bar and press Enter. Scripts in this folder will not be overwritten when installing a new version.

Anatomy of a button function

Let's take the Autopilot airspeed button function as an example, which at the time of writing looks like this:

```
local strings = require './lib/strings'

RegisterFunction(
  {
    uid = 'autoPilotAirspeed',
    category = strings.Category.FSX.AutoPilot,
    displayName = 'Autopilot airspeed',
    states = {
      on = { displayName = 'On', default = LEDColor.Green },
      off = { displayName = 'Off', default =  LEDColor.Red },
      notAvailable = { displayName = 'Not available', default = LEDColor.Off }
    }
```

```
  },
  function(context)
    SetState(context, 'notAvailable')

    OnSimConnect(context,
      {
        autoPilotAvailable = { variable = 'AUTOPILOT AVAILABLE', type =␣
↪SimConnectDataType.Bool },
        autoPilotAirspeed = { variable = 'AUTOPILOT AIRSPEED HOLD', type =␣
↪SimConnectDataType.Bool }
      },
      function(context, data)
        if data.autoPilotAvailable then
          if data.autoPilotAirspeed then
            SetState(context, 'on')
          else
            SetState(context, 'off')
          end
        else
          SetState(context, 'notAvailable')
        end
      end)
  end
)
```

Using anonymous functions like this to implement the various callbacks results in compact code, which can arguably be more difficult to read. For clarity let's expand it first. The following example works exactly the same:

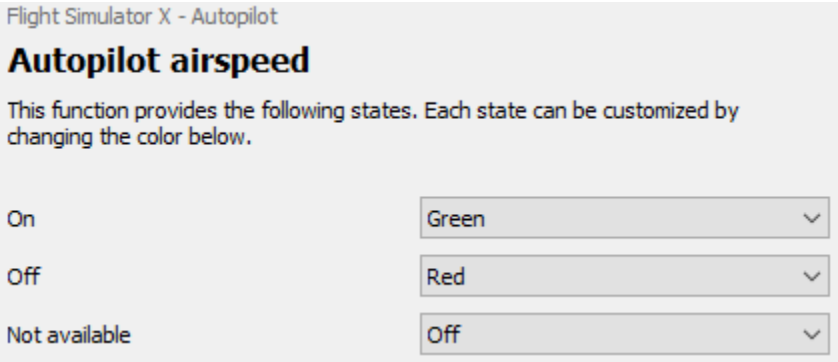
```
local strings = require './lib/strings'

local function variablesChanged(context, data)
  if data.autoPilotAvailable then
    if data.autoPilotAirspeed then
      SetState(context, 'on')
    else
      SetState(context, 'off')
    end
  else
    SetState(context, 'notAvailable')
  end
end

local function setup(context)
  SetState(context, 'notAvailable')

  OnSimConnect(context,
    {
      autoPilotAvailable = { variable = 'AUTOPILOT AVAILABLE', type =␣
↪SimConnectDataType.Bool },
      autoPilotAirspeed = { variable = 'AUTOPILOT AIRSPEED HOLD', type =␣
↪SimConnectDataType.Bool }
    },
    variablesChanged)
end


RegisterFunction(
  {
```

```
    uid = 'autoPilotAirspeed',
    category = strings.Category.FSX.AutoPilot,
    displayName = 'Autopilot airspeed',
    states = {
      on = { displayName = 'On', default = LEDColor.Green },
      off = { displayName = 'Off', default =  LEDColor.Red },
      notAvailable = { displayName = 'Not available', default = LEDColor.Off }
    }
  },
  setup)
```

So what's happening? When the script is loaded it is automatically run. At this time you should call *RegisterFunction* for each button function you want to be visible in G940LEDControl. *RegisterFunction* accepts two parameters: a table which describes the button function you want to add, and a Lua function to be called when the button function is assigned to a button.

Every button function must have a unique 'uid'. It is used to save and load profiles and should therefore not be changed once in use. The category and displayName are only used in the main and button function selection screens and can be freely changed.

A button function must also have one or more states. Each state has a key which, like the 'uid', is used to save and load profiles and should also not be changed once in use. A default *LED color* can also be set, which sets the initial value in the selection screen when assigning it to a button:



Setup function

As soon as the button function is attached to one of the buttons the setup function passed to *RegisterFunction* is called. It receives a 'context' parameter, the contents of which are not useable by the script directly, but which you need to pass along to for example *SetState* later on so that it knows which button function's state needs to be changed.

In the above example the first thing we do is to set the default state using *SetState*. The second parameter is a string containing the key of one of the states as defined in the *RegisterFunction* call.

After that you will normally call one of the built-in Lua functions to be notified of certain events. At the time of writing you can either call *OnTimer* to perform checks on a regular interval or, more common in the case of FSX, *OnSimConnect* to be notified when one or more of the simulation variables change.

CHAPTER 3

Scripting reference

G940LEDControl uses Lua 5.2. Please refer to the Lua 5.2 Reference Manual for more information.

- *Global functions*
  - *Log functions*
  - *RegisterFunction*
  - *SetState*
  - *OnSimConnect*
  - *OnTimer*
  - *WindowVisible*
  - *FSXWindowVisible*
- *Global variables*
  - *LEDColor*
  - *SimConnectDataType*

Global functions

Log functions

```
Log.Verbose(msg, ...)
Log.Info(msg, ...)
Log.Warning(msg, ...)
Log.Error(msg, ...)
```

Writes a message to the application log. If you pass more than one parameter, they will be concatenated in the log message separated by spaces. The parameters do not need to be strings, other simple types will be converted and tables will be output as '{ key = value, ... }'.

The application log can be opened on the Configuration tab of the main screen.

RegisterFunction

```
RegisterFunction(info, setupCallback)
```

Registers a button function.

Parameters

info: table
A Lua table describing the function. The following keys are recognized:

> **uid**: string
> *Required.* A unique ID for this function. Used to save and load profiles.
>
> **category**: string
> The category under which this function is grouped in the button function selection screen.
>
> **displayName**: string
> The name of the function which is shown in the main screen and button function selection screen.
>
> **states**: table
> A table of states supported by this function.
> Each state has it's own unique key and a table describing the state. The following keys are recognized for the table:
>
> > **displayName**: string
> > The name of the state which is shown in the button function selection screen.
> >
> > **default**: string
> > The default color and/or animation assigned to the state when it is first selected. See *LEDColor* for a list of valid values.
> >
> > **order**: number (optional)
> > Specifies the order in which the state is shown in the button function selection screen. If not specified, defaults to 0. States with an equal order are sorted alphabetically.

setupCallback: function
A Lua function which is called when the button function is configured. Please note that if a button function is linked to multiple G940 throttle buttons, setupCallback is called multiple times, so be careful with variables which are outside of the setupCallback's scope (global or script-local)!

setupCallback is passed a single parameter 'context'.

Example

```
RegisterFunction(
  {
    uid = 'autoPilotAirspeed',
    category = strings.Category.FSX.AutoPilot,
    displayName = 'Autopilot airspeed',
    states = {
      on = { displayName = 'On', default = LEDColor.Green },
      off = { displayName = 'Off', default =  LEDColor.Red },
      notAvailable = { displayName = 'Not available', default = LEDColor.Off }
    }
  },
  function(context)
    -- implementation of setupCallback
  end)
```

SetState

```
SetState(context, newState)
```

Sets the current state of a button function.

Parameters

context
The context parameter as passed to setupCallback which determines the button function to be updated.

newState: string
The new state. Must be the name of a state key as passed to *RegisterFunction*.

Example

```
SetState(context, 'on')
```

OnSimConnect

```
OnSimConnect(context, variables, variablesChangedCallback)
```

Registers a Lua function to be called when the specified SimConnect variable(s) change. For a list of variables please refer to Simulation variables.

Parameters

context
The context parameter as passed to setupCallback.

variables: table

A table containing information about the simulation variables you want to monitor. Each key will be reflected in the 'data' table passed to the variablesChangedCallback. Each value is a Lua table describing the variable.

variable: string
The name of the variable as described in Simulation variables.

type: string
One of the *SimConnectDataType* values.

units: string
If relevant to the variable, one of the Units of Measurement supported by SimConnect. For example, 'percent'. If type is SimConnectDataType.Bool, this will be automatically set to 'bool'.

variablesChangedCallback: function
A Lua function which is called when the variable's value changes. It receives 2 parameters: 'context' and 'data'. The data parameter is a Lua table where each key corresponds to a variable defined in the 'variables' parameter and it's value is the current value of the simulation variable.

Example

```
OnSimConnect(context,
  {
    autoPilotAvailable = { variable = 'AUTOPILOT AVAILABLE', type =␣
↪SimConnectDataType.Bool },
    autoPilotAirspeed = { variable = 'AUTOPILOT AIRSPEED HOLD', type =␣
↪SimConnectDataType.Bool }
  },
  function(context, data)
    if data.autoPilotAvailable then
      if data.autoPilotAirspeed then
        SetState(context, 'on')
      else
        SetState(context, 'off')
      end
    else
      SetState(context, 'notAvailable')
    end
  end)
```

OnTimer

```
OnTimer(context, interval, timerCallback)
```

Registers a Lua function to be called when the specified interval elapses.

Parameters

context
The context parameter as passed to setupCallback.

interval
The interval between calls to timerCallback in milliseconds. At the time of writing the minimum value is 100 milliseconds.

timerCallback
A Lua function which is called when the interval elapses. It is passed a single parameter 'context'.

Example

```
OnTimer(context, 1000,
  function(context)
    if FSXWindowVisible('ATC Menu') then
      SetState(context, 'visible')
    else
      SetState(context, 'hidden')
    end
  end)
```

WindowVisible

Checks if a window is currently visible. This is a thin wrapper around the FindWindow/FindWindowEx/IsWindowVisible Windows API. In the context of FSX panels you are probably looking for *FSXWindowVisible*.

All parameters are optional, but at least one parameter is required. To skip a parameter simply pass nil instead.

To get a window's class name, use a tool like Greatis Windowse.

Parameters

className
The window class name of the window

title
The title / caption / text of the window

parentClassName
The parent window's class name. If specified, the first two parameters are considered to be a child window of this parent.

parentTitle
The parent window's title / caption / text. If specified, the first two parameters are considered to be a child window of this parent.

FSXWindowVisible

Checks if an FSX window is currently visible. Uses WindowVisible as a workaround because SimConnect does not expose this information directly.

Parameters

title
The title of the panel.

Checks for both docked and undocked windows. Equal to:

```
WindowVisible('FS98CHILD', title, 'FS98MAIN') or WindowVisible('FS98FLOAT', title)
```

Global variables

LEDColor

Keys

- Off
- Green
- Amber
- Red
- FlashingGreenFast
- FlashingGreenNormal
- FlashingAmberFast
- FlashingAmberNormal
- FlashingRedFast
- FlashingRedNormal

The 'Fast' flashing versions stay on and off for half a second, the 'Normal' version for 1 second.

Example

```
{ default = LEDColor.Green }
```

SimConnectDataType

Keys

- Float64
- Float32
- Int64

- Int32
- String
- Bool
- XYZ
- LatLonAlt
- Waypoint

The XYZ, LatLonAlt and Waypoint data types will return a table in the 'data' parameter for the OnSimConnect callback with the following keys:

XYZ

- X
- Y
- Z

LatLonAlt

- Latitude
- Longitude
- Altitude

Waypoint

- Latitude
- Longitude
- Altitude
- KtsSpeed
- PercentThrottle
- Flags

The Flags value is a table containing the following keys, where each is a boolean:

- SpeedRequested
- ThrottleRequested
- ComputeVerticalSpeed
- IsAGL
- OnGround
- Reverse
- WrapToFirst