

Ethos-lua Tutorial

EN

1.1

08/24

Udo Nowakowski

assisted by

Lothar Thole



Table of Contents

Chapter 1: A Brief Introduction.....	4
General Information.....	4
Goal of the Tutorial.....	4
Structure of this Documentation.....	5
Typical lua use cases.....	6
Ethos lua basic knowledge.....	7
Lua Files: Naming Conventions and Folder Structure.....	7
The Different Types of Lua Scripts in Ethos.....	8
Widget.....	8
System-tool.....	8
Sourcescript.....	9
Task script.....	9
Other scripts.....	9
Script registration.....	10
handler.....	11
init().....	12
create().....	12
wakeup(widget).....	12
paint(widget).....	13
event(widget, category, value, x, y).....	13
configure(widget).....	13
write(widget).....	14
read(widget).....	14
menu(widget).....	14
close(widget).....	14
Flowchart (handler).....	15
Boot-Process.....	16
dev-Environment.....	17
Editors.....	17
Notepad++ (Win).....	17
Visual Studio Code (Win and macOS).....	18

Using the Sim for lua development.....	19
General Information.....	19
Telemetry.....	20
Folder Structure & Dedicated Work Environments.....	20
Ethos Suite.....	23
Debugging Tx Infos.....	25

Further Information Sources.....28

Ethos Lua Reference Guide.....	28
Github – Ethos community.....	28
Lua Basic Tutorial.....	28
Official Lua Site.....	29
Engel Forum: -Lua Scripts.....	29
RCG Lua thread.....	29
Lua Perfomance Tips.....	30

Glossary.....31

Array.....	31
Compiler.....	31
Class.....	31
Debug.....	31
Float.....	31
Frame.....	32
Github.....	32
Handler.....	32
Integer.....	32
Interpreter.....	33
Key.....	33
Method.....	33
MPU.....	33
Objectorientated Programming (OOP).....	33
Pointer.....	33
Registration.....	34
UID.....	34
Widget.....	34

Chapter 1: A Brief Introduction

General Information

Goal of the Tutorial

A small note: if you don't want to read the introductory blurb, please skip directly to "Ethos-Lua Basics".

This tutorial conveys the basic knowledge of Lua programming under Ethos.

After completing the tutorial, the user should be able to independently program smaller Lua programs.

A basic requirement is familiarity with PC technology and fundamental programming knowledge. It is not necessarily required to have prior experience in programming with Lua.

Even "Basic" as a programming language would already have imparted the necessary background knowledge.

This is the first version of the tutorial, which will be refined and expanded gradually over time.

If the meaning of terms (e.g., handler, interpreter, etc.) is not clear, I ask that you clarify the term using a search engine if it is not listed in the glossary.

A Glossary will be gradually built up. It is not the goal of this tutorial to convey absolute programming basics without any prior knowledge.

For this purpose, links to Lua beginner courses and reference pages are provided.

Structure of this Documentation

The guide follows a "TopDown" approach. From the general to the specific.

The section up to and including "Ethos Lua Basics" can also be interesting for pure Lua users to improve their understanding of working with Lua scripts.

First, general connections, terminologies, how Lua works under Ethos, basic functionalities, and the working environment are explained (Chapter 1).

Then you start with illustrative tasks, how to implement them, and the respective demo Lua scripts.

The requirements gradually increase.

Individual chapters bundle the topics.

It is intended to convey an increasing level of detailed understanding, from simple requirements to more complex tasks.

In the end, every participant should be able to continue their education autodidactically according to their individual needs.

Typical lua use cases

Why should one even bother with Lua for an Ethos transmitter?

If you look at the Ethos GitHub pages and the issues, there are many individual use cases and extension requests from users who demand very specific functionalities or display variants.

Very often, such a requirement fits a very limited user base and could even have negative side effects for other users or overall performance.

Even if the desired function/extension were to be implemented only optionally, additional fields would have to be provided in some Ethos menus for possible parameterization or activation.

Furthermore, every additional implementation consumes memory space, uses CPU resources, increases maintenance effort, and binds developer resources.

So, if an extension/idea serves only a small user base but imposes limitations on the general public, one should consider whether alternative implementations might be a better solution.

This is where Lua comes into play, allowing the user, if their idea is really important to them, to implement their individual needs on their transmitter as much as possible.

Additionally, they can make their "project" available to the public (e.g., via GitHub).

Is that too abstract?

Examples can be:

- Displaying more than one telemetry value in larger fonts with freely selectable color options
- Announcements whose logic goes beyond what is possible with LSWs
- Custom design of the transmitter's homepage, etc.



Otherwise, Lua scripts are also often used by accessory providers to configure their hardware or, for example, in functional model building, to extend functionalities via touch screen.

Ethos lua basic knowledge

Lua Files: Naming Conventions and Folder Structure

I would like to start with the absolute basics that do not contribute at all to the knowledge of programming:

Where do these scripts belong, where can I find them, how do I handle them as files?

Basically, all scripts must be located in a subdirectory named "scripts".

This directory is usually located on the data volume/"drive" named "NAND" or "RADIO" for transmitters with larger NAND storage, and on the SD card for transmitters with smaller NAND, such as the X20/X20S.

(For the former, the user can decide whether to offload the scripts to the SD card.)

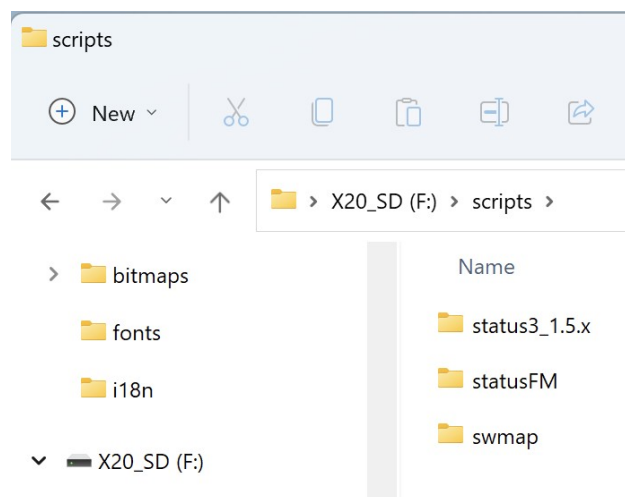
Lua scripts are identified by the suffix "*.lua" (uncompiled) or "*.luac" (compiled).

A simple script that exists solely as a single file can be placed directly in the "/scripts" folder with a freely chosen filename.

Alternatively, a subfolder can be created, but then the file **MUST** be named "main.lua" or "main.luac"!

I generally recommend the subfolder variant for the sake of clarity.

The folder should be given a clearly identifiable name.



As always, it is important to adhere to naming conventions for Ethos (pay attention to maximum file length, special characters, etc.).

The Different Types of Lua Scripts in Ethos

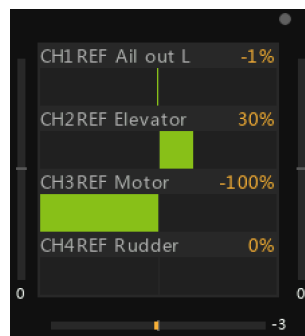
Under Ethos, there are various types of Lua scripts that are "optimized" for their specific tasks in terms of functionality. The most common are:

Widget

The most well-known type is probably the widget. These Luas are used on the home screens.

Home screens, the screens that you see immediately after turning on the transmitter or that you can scroll through afterwards, have user-defined different layouts and thus frame sizes for a widget (up to full screen). One Lua script can be configured per frame. The Lua script essentially "lives" in this frame and sees it as its "scope of validity."

This means, for example, that when drawing, it cannot draw outside of this frame; the top left corner always has the coordinates 0,0, no matter where the frame is located on the screen.



System-tool

The system tool script is a "one-time script." It is manually started via the system menu, and no further actions are executed after it is exited.

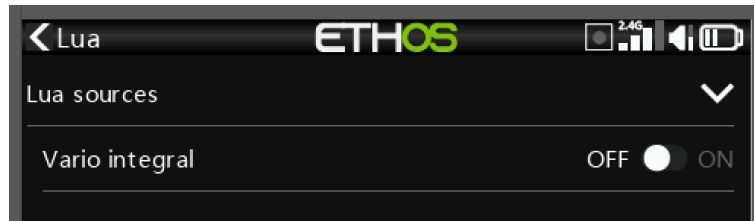


A typical application is configuring devices such as (gyro) receivers, ESCs, redundancy boxes, etc.

Source script

Anyone who has worked with Ethos for a while is familiar with the term "source" or "sources." In many menus, "sources" from categories such as analog inputs, switches, telemetry values, etc., can be selected to provide values.

Once a source script is configured into a model, it can also provide (script-calculated) values. Source scripts do not have a user interface and are activated via the "Lua" tile in the model configuration.



Task script

Here, tasks can be processed in the background.

For example, event-controlled tasks that do not require displays on the screen or the output of values in the form of the "source" category.

Other scripts

Besides the scripts already mentioned above, there are other types with very specific areas of application, such as for RF modules like Multiprotocol, Ghost, ELRS, Crossfire.

Script registration

What does "registration" mean?

All scripts found on the transmitter must be "registered" at startup and therefore made known to the operating system.

This happens regardless of which scripts are actually used in the respective model.

After completing the boot process, the transmitter essentially has a "catalog" of available scripts of the respective types and can then present selection lists during operation.

Each Lua script is divided into certain "modules", called handlers, which each cover specific tasks (e.g., background data processing, controlling/drawing the display, configuring widgets, reading or saving configurations, catching and evaluating events, etc.).

In any case, each script must define the so-called init handler so that the script is registered at transmitter startup.

During this process, it is also specified which "modules," or additional handlers, are used in the script.

The method to register such a script is a "system function" and thus from the system class.

For example, a widget is declared with the method "system.registerWidget()."

As a code snippet, here is a possible example of a complete init handler for a widget:

```
local function init()
system.registerWidget({key="Xample", name="Beispiel", create=create, wakeup=wakeup, paint=paint})
end
```

- key is the UNIQUE key for the script; it must not appear in any other script, regardless of type.
- name is the name presented in a selection list.
- create is the handler that is executed the first time the script is called and is assigned to the function of the same name "create." The handlers and their assigned functions often use the same name so you see the syntax create=create.
- accordingly, the background handler wakeup is defined.
- and similarly, the paint handler for graphical representations.
- additional handlers may be defined as required.

A detailed overview of the script-type-specific registration methods and parameters can be found in the online documentation or the Lua Reference Guide; sources will be listed later.

handler

The term "handler" will be familiar to programmers. For those who may not immediately understand what it means, here is a "beginner's explanation":

A handler in Ethos Lua can be thought of as a kind of program module.

In principle, a handler is a program routine that is started due to an overarching (system) event.

There are various "specialized" modules/handlers for different tasks, or for different "events"/triggers, such as:

- Transmitter start (initialization)
- First call
- Touch event
- Configuration adjustment...

Each module has a fixed name, an overview of the handlers is listed below.

Each module, each handler, executes a self-contained routine corresponding to its task through the program sequence it contains.

The "data exchange" between the handlers, if desired, takes place via a uniform data structure, e.g., named "widget." This data structure is appropriately generated during the first call (by the create handler).

Below is a brief explanation of the individual handlers.

More detailed functionality will be illustrated by example scripts included in the individual chapters.

List of typical handlers

init()

This handler is called during the transmitter's boot process.

The handler is used to provide the transmitter with a list of all Lua scripts with their unique "keys" (unique identifiers) and their types (widget, systemTool, etc.).

The "environment" (i.e. which handlers exist in the script) is also set up in the process.

Codeexample:

```
local function init()  
    system.registerWidget({key="GV1", name="GVinflight", create=create, paint=paint, wakeup=wakeup,  
        configure=configure, title=false, read=read, write=write} )  
end
```

create()

The create handler is executed the first time the script is called, for example, when the screen of a widget is activated for the first time.

In the create handler, the "central data structure" or an array is typically defined, which can be passed to other handlers.

This data structure must be defined as the return value of the handler.

Often (e.g., in a widget), this is named as widget.

wakeup(widget)

The wakeup handler is the general background handler for all kinds of tasks (all tasks except those handled by the handlers listed below).

This usually includes reading sources such as switch positions, analog inputs, telemetry values, their processing, and updating the central data structure (principle: input, processing, output).

It is generally called around a dozen times per second.

Another essential element is to decide whether data relevant to the display (in the paint handler) has changed.

If so, the wakeup handler triggers an update of the display using the method `lcd.invalidate()`.

Ideally (for performance reasons), the area to be updated is specified.

Multiple areas can be specified in a single wakeup run.

paint(widget)

The paint handler is responsible for graphical representations in a script.

It is therefore only necessary in scripts that display something on the screen, such as widgets and system scripts, but not source scripts.

Only here do the methods/functions for drawing, text, and value display (lcd class) have relevance. Ideally, the wakeup handler has pre-processed all necessary texts, values, and other variables, so that almost only presentation-relevant program blocks are present here.

Note:

In forums, there are often misunderstandings about the interaction of the wakeup handler, the paint handler, and the lcd.invalidate() method.

In general, it should be understood as follows:

- wakeup is responsible for general background processing and, as far as possible, for pre-processing the data used in paint. Wakeup is called very frequently by the system.
- paint is also frequently called by the system, "following" the wakeup handler. The focus is on graphical presentation. **VERY IMPORTANT: Calling a graphical method such as drawing lines, "printing" texts or values, etc., does NOT necessarily mean that this will also be directly displayed on the screen;** it initially fills a very fast image buffer.
- lcd.invalidate(x,y,w,h): Only this call causes the screen (or the specified area) to be cleared and the relatively "slow" screen memory to be filled with new data. The method should only be called from wakeup, ideally when it has been determined that something in the display has been updated!

The fact that the execution of paint does not necessarily lead to a screen refresh can be confusing for beginners but has significant performance advantages for the entire system.

event(widget, category, value, x, y)

This handler is used to process "external events," usually user interventions.

For example, touching a screen, pressing control buttons (Page Up/Down), scroll wheel, etc.

As seen above, in addition to the typical "widget" structure, Ethos also passes the category (e.g., touch event), the value (press/release), and coordinates as input parameters.

configure(widget)

Those who have already worked with widgets may be familiar with the option to configure them via a menu call. This call triggers the config handler. In particular, there are options available to design an extensive config menu via the forms class.

write(widget)

The write handler saves model-specific parameters to the model file. Typically, changes in values in the config handler trigger this write action. Of course, the handler must have already been defined by the init handler. (-;

read(widget)

The parameters written by the write handler should, of course, also be loaded when the model is loaded (or otherwise). This is done by the read handler.

menu(widget)

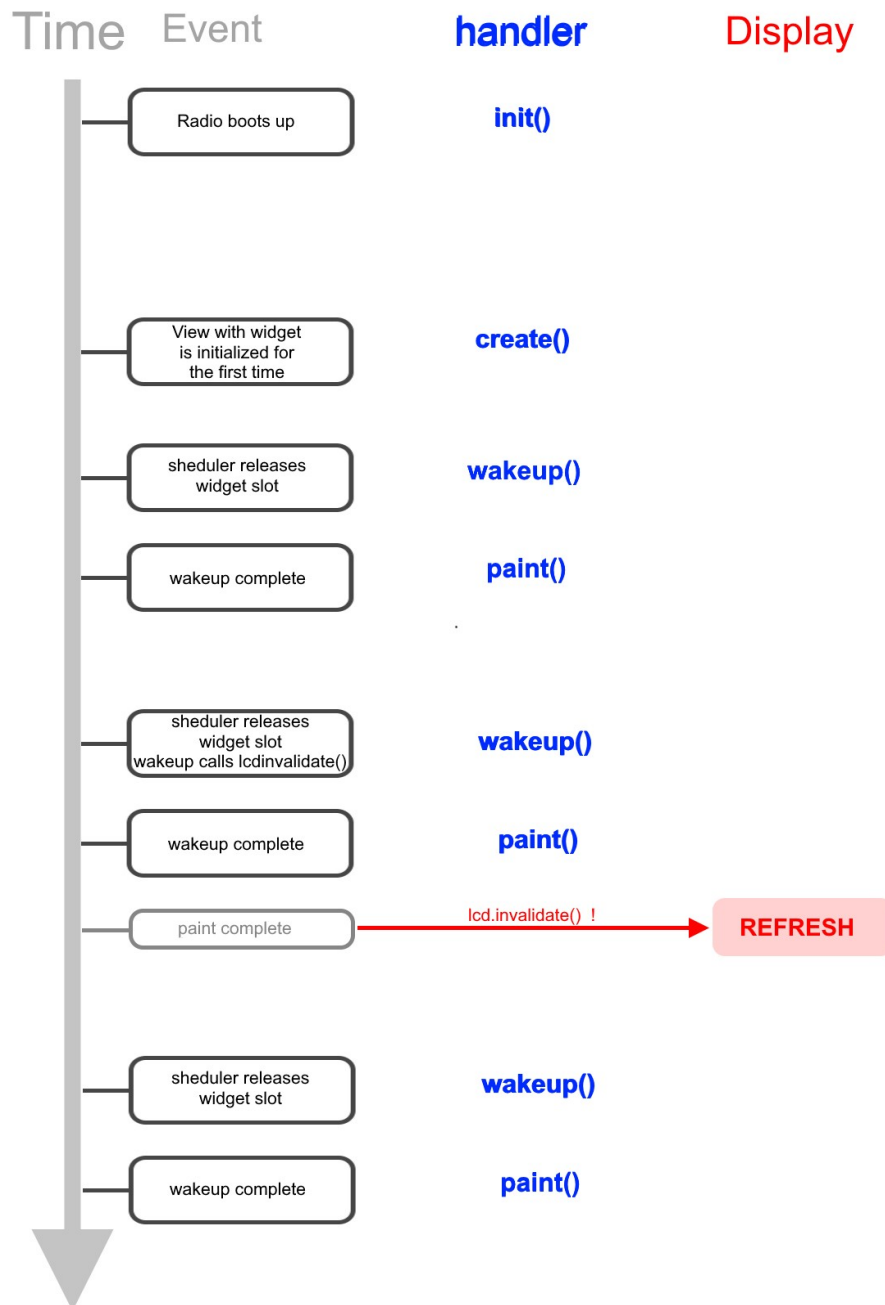
Handler that is called when creating a context menu to allow adding additional options.

close(widget)

Handler that is called when the current page is closed.

Flowchart (handler)

To roughly illustrate the flow, a small graphic showing when the handlers are called:



It also becomes apparent that the screen display is only refreshed by the call to the `lcd.invalidate()` method.

The wakeup handler should always check whether something on the display has changed (values, etc.) and only then trigger the resource-intensive refresh with `lcd.invalidate()`.

(By the way, during the very first widget call, Ethos also refreshes the display itself.)

Boot-Process

During the Ethos boot process, the operating system searches within the /scripts folder for files with the suffix .lua (or .luac if already compiled).

These are read, and the respective init handler is executed.

In the process, the UID/key, script name, type, and registered handlers are made known to the transmitter centrally.

After that, all existing folders directly under "/scripts" are opened.

There, a file "main.lua" or "main.luac" is searched for in each folder, and if present, the init handler is executed.

Further subfolders are NOT searched.

A simple syntax check of the script takes place during the boot process. If this fails, the script is not made available to the transmitter.

In this way, the transmitter has a catalog of all valid scripts at the end of the boot process and can suggest them to the user as needed.

In several forum posts, it is mentioned that Ethos would load all scripts completely, as if they were all permanently allocated in memory.

This is definitely not the case.

dev-Environment

I know you all want to get started.

However, here are a few introductory words on how to create a 'work environment' for lua development that takes advantage of existing tools, especially on the topics of editors and the Ethos simulator.

Editors

Anyone who is already programming will want to continue using their 'favorite editor.'

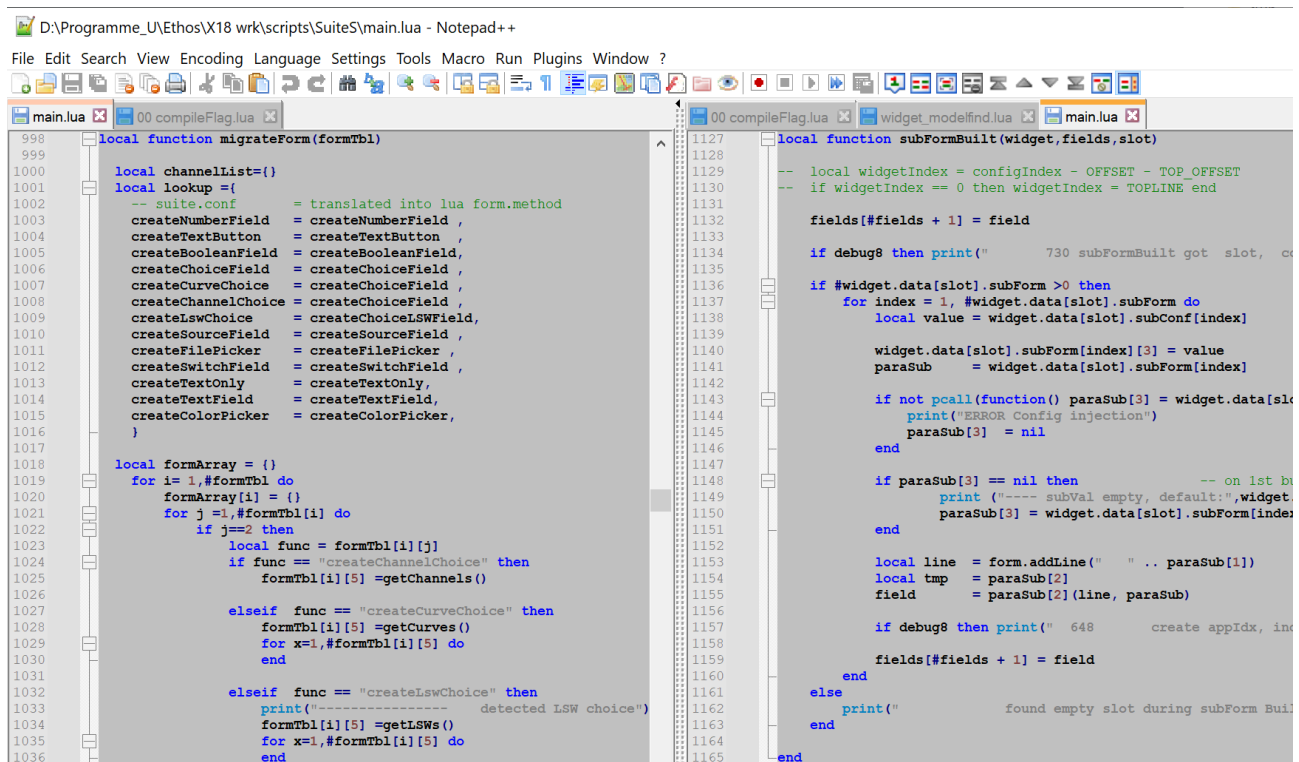
Those who are just starting and wondering which editor to use, I would like to give two recommendations:

Notepad++ (Win)

An open-source/free software project that is widely used. Advantages are easy application, 'lua' enabled (color coding/highlighting of coding), it is portable, has several additional addons, etc.

Definitely widely used and welcome if quick familiarization and easy application are more important than the number of features.

<https://notepad-plus-plus.org/>

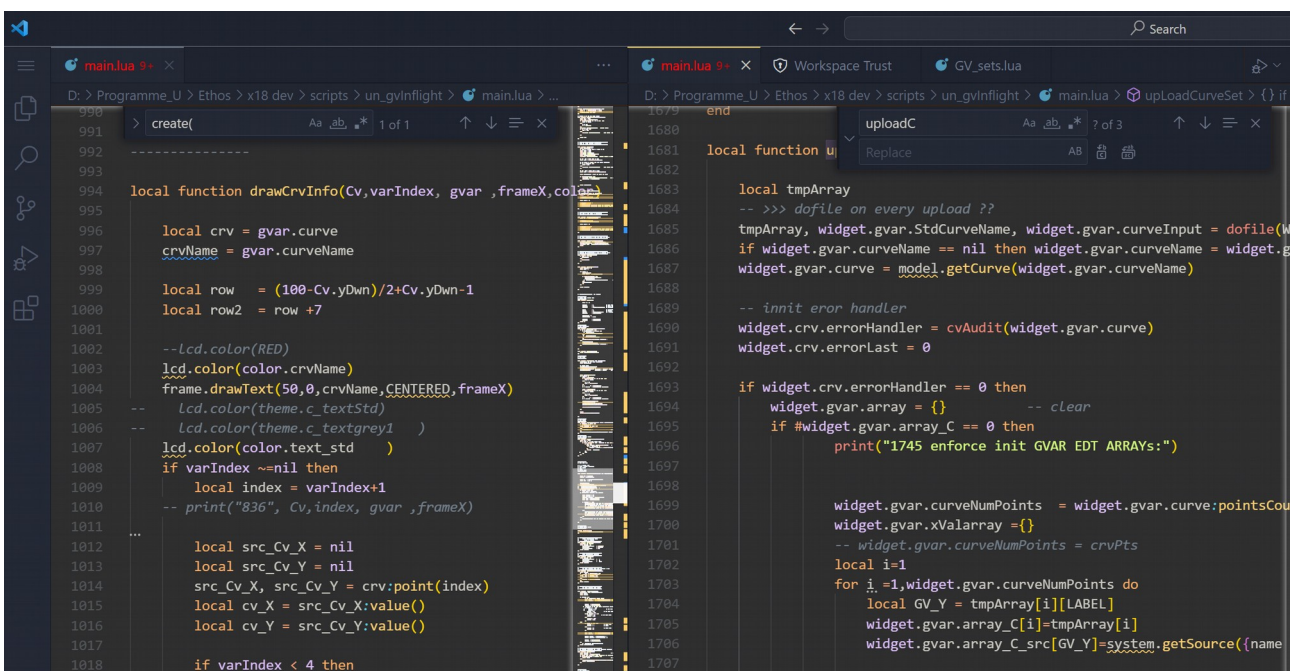


Visual Studio Code (Win and macOS)

As the name suggests, from Microsoft. VS Code is also free, functionally even more extensive than Notepad++. If you want to publish your projects on GitHub, VS Code is especially suitable because there are add-ons to synchronize files directly with git (push/pull/commit, etc.).

VS Code can be extremely customized, but it requires corresponding familiarization with the config files. The lua code support 'out of the box' is even more extensive than with Notepad++, e.g., you can already see in color whether defined variables or functions are actually used in the coding (colored gray, etc.). VS Code has more functionality than Notepad++, but it also requires more familiarization time.

<https://code.visualstudio.com/>



```
990 > create(
991
992 -----
993
994 local function drawCrvInfo(Cv,varIndex, gvar ,frameX,color)
995
996     local crv = gvar.curve
997     crvName = gvar.curveName
998
999     local row = (100-Cv.yDwn)/2+Cv.yDwn-1
1000     local row2 = row +7
1001
1002     --Lcd.color(RED)
1003     lcd.color(color.crvName)
1004     frame.drawText(50,0,crvName,CENTERED,frameX)
1005     -- lcd.color(theme.c_textStd)
1006     -- lcd.color(theme.c_textgrey1 )
1007     lcd.color(color.text_std )
1008     if varIndex ~=nil then
1009         local index = varIndex+1
1010         -- print("836", Cv,index, gvar ,frameX)
1011     ...
1012         local src_Cv_X = nil
1013         local src_Cv_Y = nil
1014         src_Cv_X, src_Cv_Y = crv:point(index)
1015         local cv_X = src_Cv_X:value()
1016         local cv_Y = src_Cv_Y:value()
1017
1018         if varIndex < 4 then
1679 end
1680
1681 local function u
1682
1683     local tmpArray
1684     -- >>> dofile on every upload ??
1685     tmpArray, widget.gvar.StdCurveName, widget.gvar.curveInput = dofile(w
1686     if widget.gvar.curveName == nil then widget.gvar.curveName = widget.g
1687     widget.gvar.curve = model.getCurve(widget.gvar.curveName)
1688
1689     -- initt error handler
1690     widget.crv.errorHandler = cvAudit(widget.gvar.curve)
1691     widget.crv.errorLast = 0
1692
1693     if widget.crv.errorHandler == 0 then
1694         widget.gvar.array = {} -- clear
1695         if #widget.gvar.array_C == 0 then
1696             print("1745 enforce init GVAR EDT ARRAYS:")
1697
1698             widget.gvar.curveNumPoints = widget.gvar.curve:pointsCou
1699             widget.gvar.xValarray ={}
1700             -- widget.gvar.curveNumPoints = crvPts
1701             local i=1
1702             for i =1,widget.gvar.curveNumPoints do
1703                 local GV_Y = tmpArray[i][LABEL]
1704                 widget.gvar.array_C[i]=tmpArray[i]
1705                 widget.gvar.array_C_src[GV_Y]=system.getSource({name
1706
1707
```


Telemetry

To replicate telemetry on the simulator (only fixed values), the RF module must be turned on in the simulator. If the model memory does not yet have telemetry sensors, then define them in the telemetry menu via the 'Discover new sensors'. All typical sensors will be found.

Folder Structure & Dedicated Work Environments

Notification!

Starting from version 1.5.12, the use of folders and paths has been significantly redesigned.

Previously, model files were centrally stored in the user environment, while the simulator itself ran with all other folders (audio, bitmaps, scripts, etc.) in its installation path.

For each "environment," such as specific simulator releases, betas, work environments, development environments, etc., a separate installation was necessary, sharing a model directory.

This led to problems when using simulators of different release versions simultaneously.

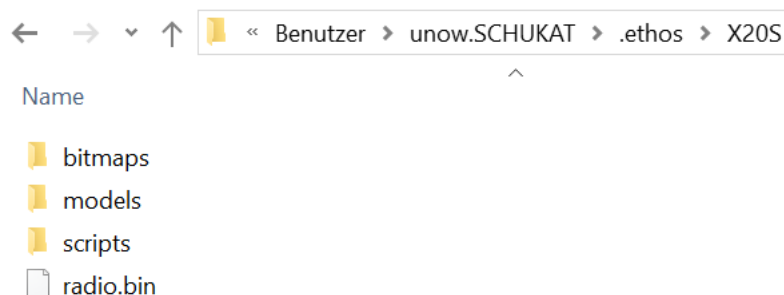
Now (version 1.5.12 & newer), the simulator and standard folders (\audio; \bitmaps\models; \bitmaps\system; \fonts; \i18n; \ scripts are located in the installation location

"[C:\Program Files\(x86\)\FrSky\Ethos\X20s](#)" (or other tx HW)



In addition, a hardware-specific "user environment" is created under the ".ethos" folder in the Windows user path C:\Users\your user name)\.ethos\X20S\models

It contains the folders logs, models, and scripts, as well as the radio.bin file including transmitter settings.



Starting from version 1.5.12, parameters can be used to specify which specific folder is used for which specific purpose (audio, system, scripts, etc.) when starting a simulator (re-routing).

These parameters are:

- `--system-directory` System directory (e.g., i18n, bitmaps)
- `--user-directory` Typical user directories (logs, models, etc.)
- `--scripts-directory` Folder for scripts
- `--radio-settings` Specifies a dedicated "radio.bin" file

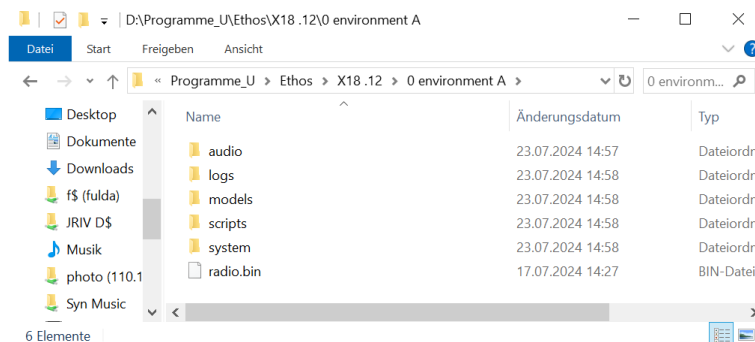
A batch file to start the simulator **could** therefore look like this:

C:

```
cd "C:\Program Files (x86)\FrSky\Ethos\X20S\"
```

```
simulator.exe --system-directory "C:\Program Files (x86)\FrSky\Ethos\X20S\0 environment A\system" --user-  
directory "C:\Program Files (x86)\FrSky\Ethos\X20S\0 environment A" --scripts-directory "C:\Program Files (x86)\  
FrSky\Ethos\X20S\0 environment A\scripts"
```

The folder of the work environment **"0 environment A"** should then look like this: ("own" sound files, model files, scripts, radio.bin)



Another work environment could be easily created without additional simulator installation by using another customized batch file and another "environment folder," e.g.:

„0 environment B“

- debug window

very important !!!

The simulator can be run with a command shell debug window.

All system messages, including lua debug code and errors, are presented there.

(Lua debug code is nothing more than a print command in certain code lines to output status info and values during the process)

Windows:

To get the debug window in the simulator, it **must** be started via command shell

- open a command shell (cmd.exe)
- Enter *"C:\Program Files (x86)\FrSky\Ethos\X20S\simulator.exe"*

Alternatively, generate and start a batch file, e.g., for a simulator installed under

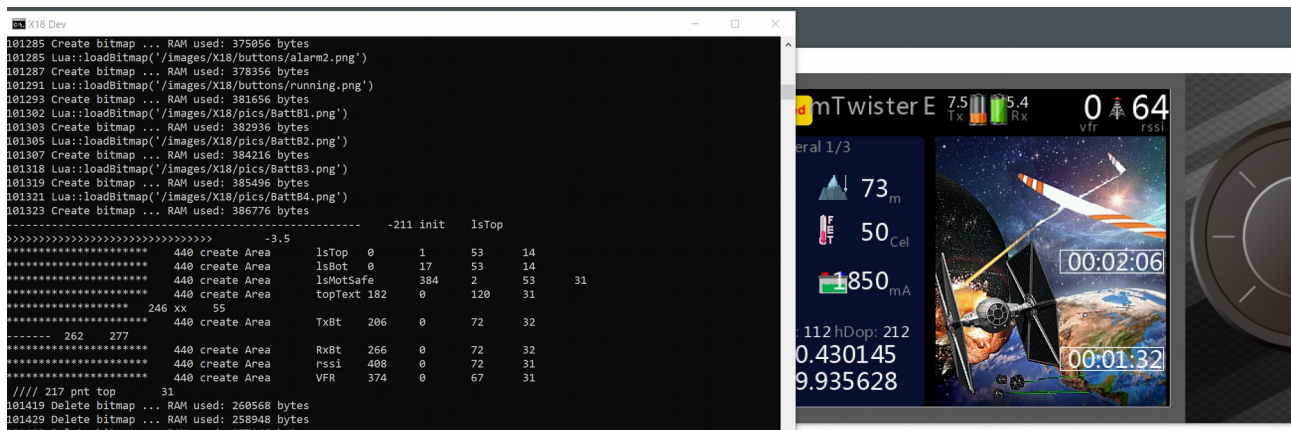
"C:\Program Files (x86)\FrSky\Ethos\X20S\"

C:

cd "C:\Program Files (x86)\FrSky\Ethos\X20S\"

"C:\Program Files (x86)\FrSky\Ethos\X20S\simulator.exe"

Left the debug window , right the started simulator



Ethos Suite

The Ethos Suite can also be used during lua script development.

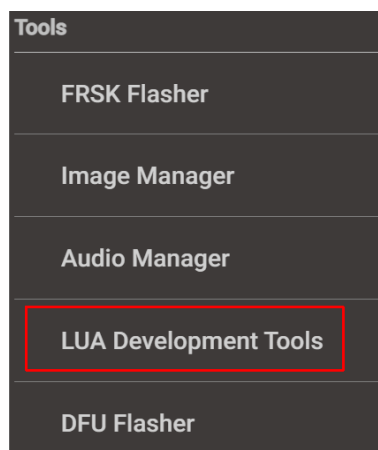
It is mostly applied 'on top' of the simulator.

The simulator provides only constant telemetry values, cannot currently process trimmer inputs accordingly, and exhibits different performance behavior.

From the moment 'real world' testing is necessary, the suite is used.

The transmitter (e.g., in suite mode) is connected to the computer and the suite is started.

By using the suite in the 'dev-tool menu,' the transmitter can be switched between operating mode and development mode.



The "developer mode" is essentially the suite mode with mounted drives, so that the Lua files are available to an editor.

If a lua file of the transmitter is open in the editor, the editor should be able to keep the file in memory, even if the file is not currently available in the operating mode of the transmitter.



This type of switching allows quick adjustments to the code and practical testing.

No cable needs to be replugged, nor does the transmitter need to be manually switched to suite/serial mode, etc.

The special and essential advantage is that a terminal window is directly available, in which debug information from the transmitter can be displayed in real-time.

These debug infos are extremely helpful in finding errors and their causes.

WARNING Safety Notice!

If the model should also be switched on in test mode:

To prevent damage from accidental starting of drive motors, etc., propellers must be removed, and depending on the model configuration, additional safety measures must be taken

Debugging Tx Infos

Debug Infos

Debug information was just mentioned. What is meant by this? Generally speaking, debug information is any (textual) information from the transmitter/simulator that helps to detect errors and their causes.

Example:

The simulator shows an error in the script: Line 2507 in main.lua could not be executed.

```
←[31m120782 wakeup() error: /scripts/un_gvInflight/main.lua:2507: attempt to index a nil value (local 'source')
```

Line 2507 in main.lua could not be executed.

Let's look at the code area:

```
2505
2506     local source      = system.getSource({name = sensor, category=CATEGORY_TELEMETRY})
2507     local sourceVal    = source:value()
2508
```

The value of a telemetry source (source) should be assigned to the variable sourceVal, but this did not work. Obviously, the source was not created (error message "..nil value (local 'source')").

Let's see what name the sensor had at that moment. For this, we add to the code and use the print command to output some "debug infos":

```
2500
2501     -- print some debug info
2502     print(">>2502: let's analyze a runtime error:")
2503     print(">>2503: variable sensor is :",sensor)
2504
2505     local source      = system.getSource({name = sensor, category=CATEGORY_TELEMETRY})
2506     local sourceVal    = source:value()
2507
```

The simulator now outputs:

```
>>2502: let's analyze a runtime error:
>>2503: variable sensor is :    5
←[31m006597 wakeup() error: /scripts/un_gvInflight/main.lua:2506: attempt to
```

The name of a telemetry sensor is "5," which is certainly wrong, so we ensure the correct sensor name (e.g., "RSSI") in the coding, and the error is fixed. In the debug window of the Sim, such information is easily provided. But how do I get debug info directly from the transmitter?

One possibility has just been described: via Suite and the dev-tools. There is a terminal window that displays the information.

What alternative is there? It is important to know that this information is sent as a kind of "serial text transmission" via USB. The technique of serial transmission has existed since the early days of computer technology. One hardware standard for serial transmission was the RS232 interface, which can still be emulated via USB. To display the information, we certainly do not use an antique text terminal (unless someone still owns, for example, DEC VT510 devices), but we leave this to a software app.

Windows (Putty):

A very common and free program is "Putty": <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

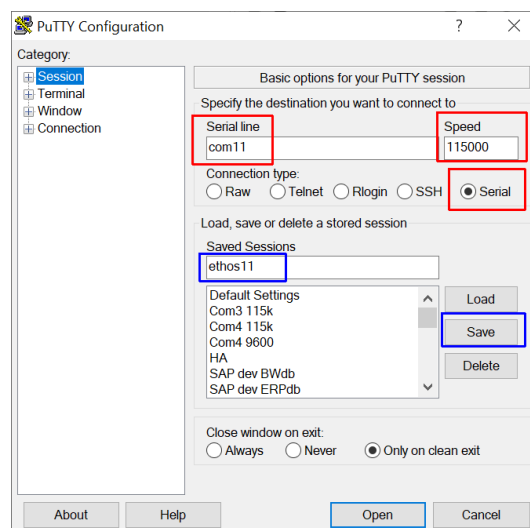
This app visualizes the text data coming from the transmitter via USB. The data is sent at a certain speed and "lands" on a specific "virtual" interface through emulation. Once Putty is installed, this communication interface needs to be set up first.

It is generally possible for multiple devices to communicate serially with a computer at the same time. Each device must use its own interface. The former physical RS232 or "Com" (Communication) interfaces were therefore present on the host multiple times and numbered (Com1: to Com32: and more).

The USB interface now emulates a specific port, depending on which USB port the transmitter is connected to, which OS version, which driver, etc. We need to find out which port the transmitter is assigned to via the Device Manager.

To do this, the transmitter is connected to the computer and put into serial mode. We then open the Device Manager and check which so-called COM interface the transmitter is connected to.

In Putty, we define a session with exactly this COM interface and the parameters 115kB (speed).



We save the session under a unique name (e.g., "ethos11") and can now start a terminal window by clicking "Open," which will display information from the transmitter.

Further Information Sources

Ethos Lua Reference Guide

<https://www.frsky-rc.com/wp-content/uploads/Downloads/EthosSuite/LuaDoc/index.html>

The Ethos Lua Reference Guide represents the official documentation from FrSky.

It is a kind of reference book or overview of all currently available classes and methods under Ethos Lua. Additionally, the system constants are listed (category "base").

Usually, you start via the class overview and look at the available methods within a class to find out what functionalities can be implemented within a class.

Often, an example is provided for a method.

Github – Ethos community

<https://github.com/FrSkyRC/ETHOS-Feedback-Community>

As is well known, FrSky provides the latest software versions on GitHub, and using issue management, extensions can be requested or bugs can be reported.

On the entry page of the so-called repo, there is a typical git navigation structure with files and folders.

Among them is the folder "lua."

Here, several complete example Lua scripts on various topics such as forms ("tool-form"), graphical control ("widget-lcddemo"), and much more are available as examples.

Lua Basic Tutorial

<https://www.tutorialspoint.com/lua/index.htm>

Anyone who does not have any Lua/programming knowledge can find an online tutorial here for general programming knowledge.

The tutorial is, of course, general and does not address the specific requirements of Ethos Lua.

Fundamentals on topics such as variables, data types, operators, general syntax, etc., are clearly presented.

Official Lua Site

<https://www.lua.org/>

<https://www.lua.org/manual/5.4/>

<https://www.lua.org/pil/contents.html#P3>

The official Lua site.

A well-structured information source about Lua with a listing of all basic functions.

Especially noteworthy is the overview of the libraries with their (partially version-dependent) implemented methods.

These are, for example, important in handling strings ("string"), mathematical functions ("math"), IO operations (io), etc.

Engel Forum: -Lua Scripts

<https://www.frsky-forum.de/forum/index.php?board/208-lua-scripte-f%C3%BCr-ethos/>

<https://www.frsky-forum.de/forum/index.php?board/241-download-bereich-f%C3%BCr-fertige-scripte/>

The very active German forum also has sections for discussing Lua topics, as well as an area where finished Lua scripts can be posted by authors.

RCG Lua thread

<https://www.rcgroups.com/forums/showthread.php?4018791-FrSky-ETHOS-Lua-Script-Programming>

The international "counterpart" to the Engel forum.

Here, too, there are regular discussions about the topic of Ethos Lua.

On the entry page, some scripts have been linked by Mike.

Lua Performance Tips

A very good document about optimizing lua performance can be found here:
<https://www.lua.org/gems/sample.pdf>

**This concludes the first chapter (general introduction).
Next, we finally continue with a small practical exercise.
The first self-programmed Lua widget.**

Glossary

Array

An ordered collection of elements addressed by indices.

In Lua, arrays are implemented as tables, with indices typically starting at 1.

Arrays can be multidimensional.

Compiler

A program that converts source code into an executable file.

However, Lua primarily uses an interpreter, but there are compilers like LuaJIT that can compile Lua code.

Class

A construct in object-oriented programming that defines a blueprint for objects, including their properties and methods.

Lua supports object-oriented programming through tables and metatables.

Debug

Generally, the process of removing errors from the code.

Under Ethos, the necessary information (values, process info, etc.) is largely provided via the print command in the simulator's shell window or via serial output.

Float

A data type for numbers with decimal places.

In Lua, all numbers are floats by default

Frame

Ethos-Lua: the area/frame in which a script runs on the home screen.

For the script, it is the "workspace" and "event horizon."

The script does not know "the world" outside the frame.

Github

GitHub (<https://github.com/>) is a web-based platform for version control and collaborative development of software projects.

It uses Git, a distributed version control system, to track changes in the code.

Developers can create repositories, share code, report bugs, and collaborate on projects.

Users can also report bugs or request enhancements. Wikis are available.

GitHub also offers features such as project management tools, wikis, and Continuous Integration/Continuous Deployment (CI/CD).

Handler

General:

A function or method that is called to process a specific event or condition.

Handlers are triggered by an event, such as:

- Touch input or operation of a control button (event)
- Calling the configuration menu (configure)
- Changing a widget configuration (read/write)
- Or when the system loops (e.g., mixer processing, LSW's, SF's, etc.) are completed in the internal process (wakeup)
-

Which handlers exist in the script is defined as an argument during registration.

Integer

A data type for whole numbers without decimal places.

In Lua, integers can also be represented as floating-point numbers, but there are libraries that support explicit integer operations.

Interpreter

A program that executes source code directly instead of compiling it. Lua is interpreted by default.

Key

A unique identifier in a table used to store and retrieve values. In Lua, keys can be any values except nil, but they are often strings or numbers.

Method

A function defined within a class or object that accesses or manipulates the object's data.

MPU

Abbreviation for Microprocessor or Microcontroller Processing Unit.

In embedded programming, this refers to the central processor of an "embedded application."

Objectorientated Programming (OOP)

A programming paradigm based on objects that encapsulate data (attributes) and functions (methods).

Lua supports OOP through the use of tables and metatables.

https://en.wikipedia.org/wiki/Object-oriented_programming

Pointer

A pointer is a variable that stores the address of another value.

Lua does not have explicit pointers like C/C++, but table references can be used similarly to pointers !

Registration

By registration, the script is made known to the Ethos system.

During the boot process, the transmitter searches the /scripts directory for all executable scripts, checks the syntax for major errors, and registers them if the check is plausible.

For this, the initialization handler is executed. This specifies the UID/Key, name, and type of the script (widget, source, task, etc.) and defines the list of used handlers.

For example, a widget named "Vartst" with various handlers is registered here, and the optional title display is generally turned off (title=false).

Code



```
local function init()
    system.registerWidget({key="test10", name="VARTst", create=create, paint=paint, wakeup=wakeup, configure=configure, read = read, write =
write, title=false} )
end
```

UID

The "UID" (unique identifier) or "key" is a so-called unique key.

It identifies the script to the Ethos operating system. Even if the script's name is changed, Ethos can still "find" the script.

Therefore, the UID must not be used in another script.

It is used in the init handler during registration.

Widget

Often a graphical control element in a user interface, such as a button, textbox, or menu.

In Lua, widgets are little Apps running in a frame, organized by GUI framework.