

Ethos-lua Tutorial

DE

1.1

08/24

Udo Nowakowski
mit Unterstützung von

Lothar Thole



Inhaltsverzeichnis

Kapitel1: eine kurze Einführung.....	4
Allgemeines.....	4
Ziel des Tutorials.....	4
Aufbau dieser Dokumentation.....	5
typische lua Anwendungsfälle.....	6
Ethos lua Basiswissen.....	7
Lua Dateien: Namenskonvention und Ordnerstruktur.....	7
Die unterschiedlichen Lua script Typen in Ethos.....	8
Widget.....	8
System-tool.....	9
Sourcescripte.....	9
Task scripte.....	10
sonstige scripte.....	10
die Registrierung.....	11
handler.....	12
init().....	13
create().....	13
wakeup(widget).....	13
paint(widget).....	14
event(widget, category, value, x, y).....	15
configure(widget).....	15
write(widget).....	15
read(widget).....	15
menu(widget).....	15
close(widget).....	15
Ablaufdiagramm handler.....	16
Startprozesse.....	17
Die Arbeitsumgebung.....	18
Editoren.....	18
Notepad++ (Win).....	18
Visual Studio Code (Win).....	19

Nutzung des Simulators zur Entwicklung.....	20
Allgemeines.....	20
Telemetrie.....	21
Ordnerstruktur & dedizierte Arbeitsumgebungen.....	21
Ethos Suite.....	24
Debuggen der Tx Infos.....	26
Debug Infos.....	26
Windows (putty):.....	27
Weitere Informationsquellen.....	29
Ethos Lua Reference Guide.....	29
Github – Ethos community.....	29
Lua Basic Tutorial.....	29
Offizielle Lua Seite.....	30
Engel Forum: -Lua Skripts.....	30
RCG Lua thread.....	30
Lua Perfomance Tips.....	31
Glossar.....	32
Array.....	32
Compiler.....	32
Debug.....	32
Float.....	32
Frame.....	32
Github.....	33
Handler.....	33
Integer.....	33
Interpreter.....	33
Key.....	34
Klasse.....	34
Methode.....	34
MPU.....	34
Objektorientierte Programmierung (OOP).....	34
Pointer.....	34
Registrierung.....	35
UID.....	35
Widget.....	35

Kapitel1: eine kurze Einführung

Allgemeines

Ziel des Tutorials

(kleiner Hinweis: wer kein einführendes BlaBla lesen möchte bitte direkt weiter zu „Ethos-Lua Basiswissen“)

Dieses Tutorial vermittelt das Basiswissen der lua Programmierung unter Ethos.

Nach Abschluss des Tutorials sollte sich der Anwender in der Lage befinden kleinere lua Programme selbstständig zu programmieren.

Grundvoraussetzung ist dabei, dass man sich in PC Technik auskennt und grundlegende Kenntnisse in der Programmierung besitzt. Es ist dabei nicht zwingend erforderlich bereits in lua programmiert zu haben.

Selbst „Basic“ als Programmiersprache würde bereits das notwendige Hintergrundwissen übermitteln haben.

Dies ist die erste Version des Tutorials, es soll später sukzessive verfeinert und erweitert werden.

Sollte die Bedeutung von Begrifflichkeiten (z.B. handler, interpreter etc.) nicht klar sein, bitte ich daher falls diese nicht Glossar aufgeführt sind, mithilfe einer Suchmaschine den Begriff zu klären.

Ein Schlagwortverzeichnis wird sukzessive aufgebaut.

Es ist nicht Ziel dieses Tutorials, absolute Grundlagen der Programmierung komplett ohne Vorkenntnisse zu vermitteln.

Dazu verzweigen Links auf Lua Anfängerkurse und Nachschlageseiten

Aufbau dieser Dokumentation

Der Leitfaden folgt einem „TopDown“ Ansatz.

Vom Allgemeinen hin zum speziellen.

Der Bereich bis inkl. „Ethos Lua Basiswissen“ kann auch für die reinen lua Anwender interessant sein, um das Verständnis im Umgang mit lua scripten zu verbessern.

Zunächst werden allgemeine Zusammenhänge, Begrifflichkeiten, wie funktioniert lua unter Ethos, grundlegende Funktionsweisen und die Arbeitsumgebung erläutert (Kapitel 1)

Danach startet man mit beispielhaften Aufgaben, wie man sie umsetzt und jeweiligen Demo-luas.

Die Anforderungen steigern sich dabei sukzessive.

Einzelne Kapitel bündeln die Themengebiete

Es soll von einfachen Anforderungen hin zu komplexeren Aufgaben immer mehr Detail-Verständnis übermittelt werden

Am Ende sollte jeder Teilnehmer in der Lage sein, sich autodidaktisch weiter auf seine individuellen Bedürfnisse hin Fortzubilden

typische lua Anwendungsfälle

Weshalb sollte man sich überhaupt mit lua für einen Ethos Sender befassen ?

Schaut man auf die Ethos-github Seiten und die issues, existieren viele individuelle Anwendungsfälle und Erweiterungswünsche von Anwendern, die ganz bestimmte Funktionalitäten bzw Darstellungsvarianten einfordern.

Sehr häufig „passt“ eine derartige Anforderung auf einen sehr begrenzten Anwenderkreis und hätte ggf. sogar negative Seiteneffekte für andere Anwender oder die Performance allgemein.

Selbst wenn die gewünschte Funktion/Erweiterung nur optional umgesetzt werden würde, müssten dazu in irgendwelchen Ethos Menüs weitere Felder zu eventuellen Parametrisierung bzw Aktivierung vorgehalten werden.

Des weiteren „frisst“ jede erweiternde Implementierung Speicherplatz, kostet CPU Ressourcen, erhöht den Wartungsaufwand und bindet Entwicklerressourcen.

Wenn also eine Erweiterung / Idee nur einen geringen Anwenderkreis bedient, dafür aber Einschränkungen auf Seiten der Allgemeinheit bedeutet, sollte man sich fragen, ob Alternative Umsetzungen nicht die bessere Lösung darstellen.

Hier springt lua in die Bresche und ermöglicht es dem Anwender, wenn ihm seine Idee wirklich wichtig ist, weitestgehend die individuellen Bedürfnisse auf seinen Sender selbst umzusetzen.

Zudem kann er (z.B. via github) sein „Projekt“ der Allgemeinheit zur Verfügung stellen.

Ist das zu Abstrakt?

Beispiele können sein :

- mehr als einen Telemetriewert in grösseren Schriften mit farblich frei wählbaren Optionen umzusetzen
- Ansagen, deren Logik über die der per LSW's möglichen hinaus gehen
- individuelles Design der Homepage des Senders etc..



Ansonsten werden lua scripte auch gerne von Zubehöranbietern genutzt, um deren Hardware zu konfigurieren, oder wie z.B. im Funktionsmodellbau, Funktionalitäten per touch screen zu erweitern.

Ethos lua Basiswissen

Lua Dateien: Namenskonvention und Ordnerstruktur

Ich möchte mit den absoluten Grundlagen starten, die überhaupt nichts zum Wissen um die Programmierung beitragen:

Wo gehören diese scripte hin, wo finde ich sie,
wie gehe ich mit Ihnen als Datei um ?

Grundsätzlich müssen sich alle scripte in einem Unterverzeichnis namens „scripts“ befinden.

Dieses Verzeichnis befindet sich standardmässig bei Sendern mit grösserem NAND Speicher auf dem Datenvolume / „Laufwerk“ namens „NAND“ bzw „RADIO“, bei Sendern mit kleinem NAND, wie z.B. X20 / X20s , auf der SD Card.

(Bei ersteren kann der Anwender übrigens selbst entscheiden, ob die scripte auf SD card auslagern möchte)

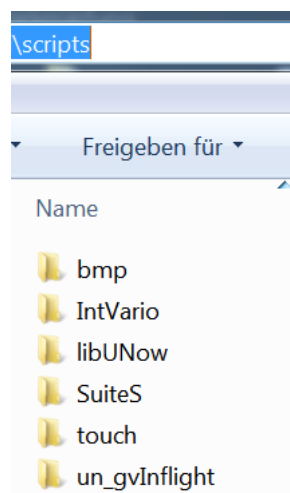
Lua scripte erkennt man am Suffix „*.lua“ (unkompiliert), bzw „*.luac“ (kompiliert)

Ein einfaches script, was ausschliesslich als eine Datei existiert kann direkt mit einem frei wählbaren Dateinamen unter dem Ordner „/scripts“ abgelegt werden.

Als Alternative kann ein Unterordner angelegt werden, dann MUSS die Datei aber in „main.lua“ bzw „main.luac“ benannt sein !

Grundsätzlich empfehle ich die Variante mit dem Unterordner, da dies der Übersichtlichkeit dient.

Dem Ordner sollte ein eindeutig identifizierbarer Name gegeben werden.



Wie immer gilt es, sich an Namenskonventionen für Ethos zu halten
(maximale Dateilänge beachten, Thema Sonderzeichen etc..)

Die unterschiedlichen Lua script Typen in Ethos

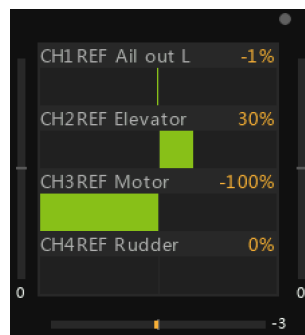
Unter Ethos existieren diverse Lua-Script Typen, die hinsichtlich Funktionsumfang auf Ihr Aufgabengebiet „optimiert“ sind. Die häufigsten sind:

Widget

Der bekannteste Typ dürfte wohl das widget sein. Diese Luas werden in den Home screens verwendet.

Home Screens, also die screens, die man nach dem Einschalten sofort sieht bzw durch die man danach blättern kann, besitzen vom Benutzer definierte unterschiedliche Layouts und damit Rahmengrößen für ein widget. (bis hin zum Vollbild). Pro Rahmen kann ein lua konfiguriert werden. Das lua „lebt“ quasi in diesem Rahmen und sieht ihn als „Gültigkeitshorizont“.

Das bedeutet z.B. dass beim Zeichnen nicht aus diesem Rahmen hinaus gezeichnet werden kann, die obere linke Ecke hat immer die Koordinaten 0,0, egal wo sich auf dem Bildschirm der Rahmen befindet.



System-tool

Das System-tool script ist ein „Einmalscript“ es wird manuell über das System Menue gestartet und nach verlassen werden keine weiteren Aktionen mehr ausgeführt.

Typischer Anwendungszweck ist das Konfigurieren von Geräten wie (Gyro-) Empfänger, ESC's, Redundanzboxen etc..



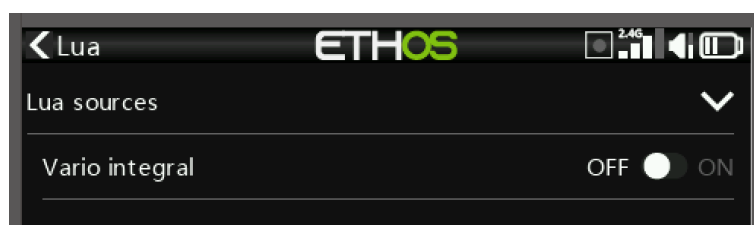
Sourcescripte

Wer mit Ethos bereits ein wenig gearbeitet hat kennt den Begriff „Quelle“ bzw „Sources“.

In vielen Menues können „Quellen“ aus Kategorien wie analoge Geber, Schalter, Telemetriewerte usw. ausgewählt werden, um Werte zu liefern.

Sobald ein Source script in ein Modell hineinkonfiguriert wurde, ist dies in der Lage ebenfalls (im script berechnete) Werte zur Verfügung zu stellen

Source scripte besitzen kein User interface und werden über die Kachel „Lua“ in der Modell Konfiguration aktiviert



Task scripte

hier können im Hintergrund z.B. event gesteuert Aufgaben abgearbeitet werden, die weder Darstellungen auf dem Display noch Ausgabe von Werten in Form der Kategorie „Quelle“ benötigen

sonstige scripte

Ausser den bereits oben genannten scripten existieren weitere Typen mit sehr speziellen Einsatzbereichen, z.B. für RF Module wie Multiprotokoll, Ghost, ELRS, Crossfire

die Registrierung

Was bedeutet "Registrierung" ?

Alle auf dem Sender gefunden scripte müssen beim Start des Senders "registriert" und daher dem Betriebssystem bekannt gemacht werden.

Dies geschieht unabhängig davon, welche Scripte im jeweiligen Modell überhaupt verwendet werden.

Der Sender besitzt nach Beenden des Bootvorgangs quasi einen "Katalog" von verfügbaren scripten der jeweiligen Typklassen und kann daraufhin während der Bedienung Auswahllisten präsentieren.

Jedes lua script unterteilt sich in gewisse „Module“, sogenannte handler, die jeweils spezielle Aufgabengebiete abdecken. (z.B. Datenverarbeitung im Hintergrund, Display ansteuern / zeichnen, widget konfigurieren, Konfiguration lesen bzw speichern, Ereignisse abfangen und auswerten....).

Auf jeden Fall muss jedes script den sogenannten init-handler definiert haben, damit beim Senderstart das jeweilige script registriert wird.

In dem Vorgang wird auch mit angegeben, welche „module“, also weiteren handler, im script benutzt werden.

Die Methode so ein script zu registrieren ist eine „Systemfunktion“ und damit aus der Klasse system

Ein widget wird beispielsweise mit der Methode „system.registerWidget()“ deklariert.

Als Code-snippet das mögliche Beispiel eines kompletten init Handlers für ein widget:

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

- **key** ist der EINDEUTIGE Schlüssel für das script, dieser darf in keinem anderen script, egal welchen Typs, nochmals vorkommen
- **name** ist der in einer Auswahlliste präsentierte Name
- **create** ist der handler, der beim ersten Aufruf des scripts ausgeführt wird und wird der Funktion gleichen Namens „create“ zugewiesen. Meistens verwendet man für die Funktion den gleichen Namen wie den des handlers, daher sieht man den Syntax „create=create“
- entsprechend wird der Hintergrund handler **wakeup** definiert
- und ebenso der **paint** handler für grafische Darstellungen

Eine detaillierte Übersicht der script-Typ spezifischen Registrierungs Methoden sowie der Parameter sind der Online Doku bzw dem Lua Reference Guide zu entnehmen, Quelle dazu wird später noch gelistet.

handler

Der Begriff „handler“ wird Programmierern ein Begriff sein. Wer sich nicht auf auf Anhieb etwas darunter vorstellen kann, eine „Einsteigererklärung“:

Ein handler im Ethos lua kann als eine Art Programm-Modul aufgefasst werden.

Prinzipiell ist ein handler eine Programmroutine, die aufgrund eines übergeordneten (System) Ereignisses gestartet wird.

Es existieren verschiedene „spezialisierte“ Module/handler für verschiedene Aufgabengebiete, bzw für verschiedene „Ereignisse“ / Trigger wie z.B:

- Senderstart (Initialisierung)
- erster Aufruf
- Touch event
- Konfigurationsanpassung

Jedes Modul besitzt einen fest definierten Namen, eine Übersicht der handler ist unten aufgeführt.

Jedes Modul, jeder handler ,führt durch den jeweils enthaltenen Programmablauf eine in sich geschlossene Routine aus, die seinem Aufgabengebiet entspricht.

Der „Datenaustausch“ zwischen den handlern, falls gewünscht, findet dabei über eine einheitliche Datenstruktur, z.B. mit Namen „widget“, statt. Diese Datenstruktur wird sinnvollerweise beim Erstaufruf (durch den create handler) generiert.

Es folgt eine knappe Erläuterung zu den einzelnen handlern.

Die detailliertere Funktionsweise wird sich aus Beispielscripten ergeben, die den einzelnen Kapiteln beigelegt sind.

Liste der typischen handler:

init()

Dieser handler wird während des Bootvorgangs des Senders aufgerufen.

Der handler wird verwendet, um dem Sender eine Liste aller Luas mit Ihren eindeutigen „Keys“ (eindeutige Schlüssel/identifizier) und ihres Typs (widget, systemTool..) bekannt zu machen. Die „Umgebung“ (welche handler existieren im script) wird dabei ebenfalls eingerichtet.

Codebsp:

```
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()

Der create handler wird beim erstmaligen Aufruf des scripts ausgeführt, zum Beispiel wenn das erste mal der screen eines widgets aktiviert wird.

Im create handler wird üblicherweise die „zentrale Datenstruktur / ein array definiert, dass an andere handler übergeben werden kann.

Diese Datenstruktur muss als Rückgabewert des handlers definiert sein.

Häufig (z.B. in einem widget), wird diese als widget benannt.

wakeup(widget)

Der wakeup handler ist der allgemeine Hintergrund handler für Aufgaben aller Art (alle Aufgaben außer denen, die in den unten aufgeführten handlern bearbeitet werden)

Dazu gehört meistens das Auslesen von Quellen wie Schalterstellungen, Analog-Inputs, Telemetrie Werte, deren Verarbeitung, Aktualisierung der zentralen Datenstruktur. (Prinzip Eingabe, Verarbeitung Ausgabe).

Er wird i.d. Regel rund dutzend mal per Sekunde aufgerufen

Ein weiteres wesentliches Element liegt darin, die Entscheidung zu treffen, ob sich Daten verändert haben die für die Anzeige (im paint handler) relevant sind.

Wenn ja triggert der wakeup handler durch die Methode lcd.invalidate() eine Aktualisierung des Displays.

Idealerweise (aus Performancegründen) wird dabei der Bereich mit angegeben, der zu aktualisieren ist.

Es dürfen mehrere Bereiche in einem wakeup-Durchlauf angegeben werden.

paint(widget)

Der paint handler übernimmt die grafischen Darstellungen in einem script.

Er ist daher nur in den scripten notwendig, die etwas auf das Display anzeigen, z.b. Widgets & Systemscripte, aber nicht Source scripte.

Nur hier haben die Methoden/Funktionen zum Zeichnen, Text und Wertedarstellung etwas verloren (lcd Klasse). Idealerweise hat der wakeup handler alle notwendigen Texte, Werte und sonstigen Variablen vorverarbeitet, so dass hier fast nur noch zur Präsentation relevante Programmblöcke vorhanden sind.

Hinweis:

In Forum tauchen immer wieder Verständnisprobleme über das Zusammenwirken des *wakeup* handlers, des *paint* handlers, und der *lcd.invalidate()* Methode .

Allgemein sollte es so aufgefasst werden:

- **wakeup** ist für die allgemeine Hintergrundverarbeitung und damit auch (soweit es möglich ist) zur Vorverarbeitung der in paint verwendeten Daten zuständig. Wakeup wird sehr häufig vom System aufgerufen.
- **paint** wird ebenfalls häufig vom System aufgerufen, „folgt“ dem wakeup handler quasi. Schwerpunkt liegt in der grafischen Präsentation. GANZ WICHTIG: **Der Aufruf einer grafischen Methode wie zum Beispiel das Zeichnen von Linien, „Drucken“ von Texten oder Werten etc., heißt NICHT zwangsläufig, dass dies auch direkt auf dem Bildschirm angezeigt wird**, es wird quasi zunächst ein sehr schneller Bild-Zwischenspeicher gefüllt
- **lcd.invalidate(x,y,w,h):** erst dieser call führt dazu, dass der Bildschirm (bzw der angegebene Bereich) am Display gelöscht, und der relativ „langsame“ Bildschirmspeicher mit neuen Daten gefüllt wird. Die Methode sollte nur von Wakeup aus aufgerufen werden. idealerweise wenn festgestellt ist, dass sich etwas an der Darstellung aktualisiert hat !

Die Tatsache, dass der Durchlauf von paint nicht zwangsläufig zu einem Bildschirmrefresh führt kann für Einsteiger verwirrend sein, hat aber deutliche Performancevorteile für das gesamte System

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

Dieser handler dient zur Verarbeitung von „äußeren Events“, in der Regel Benutzerinterventionen.

Beispielsweise Touch eines Screens, Drücken von Bedientasten (Page Up/down), Scrollrad etc...

Man sieht oben, dass außer der typischen „widget“ Struktur, seitens Ethos noch die Kategorie (z.B. touch event), der Wert (drücken / loslassen) und Koordinaten als Eingangsparameter mit übergeben werden.

configure(widget)

Wer bereits mit widgets gearbeitet hat, kennt ggf. die Möglichkeit diese über einen Menüaufruf zu konfigurieren. Dieser Aufruf triggert den config handler.

Es stehen insbesondere Möglichkeiten zur Verfügung, via forms Klasse ein umfangreiches Config Menü zu gestalten.

write(widget)

Über den write handler werden modellspezifische Parameter in die Modelldatei weggespeichert.

Typischerweise triggern Änderungen der Werte im Config handler diese Schreibaktion.

Dazu muss der handler natürlich bereits durch den init handler definiert worden sein (-;

read(widget)

Die durch den write handler geschriebenen Parameter sollen natürlich auch beim Laden des Modells (oder anderweitig) geladen werden können.

Dies geschieht durch den read handler.

menu(widget)

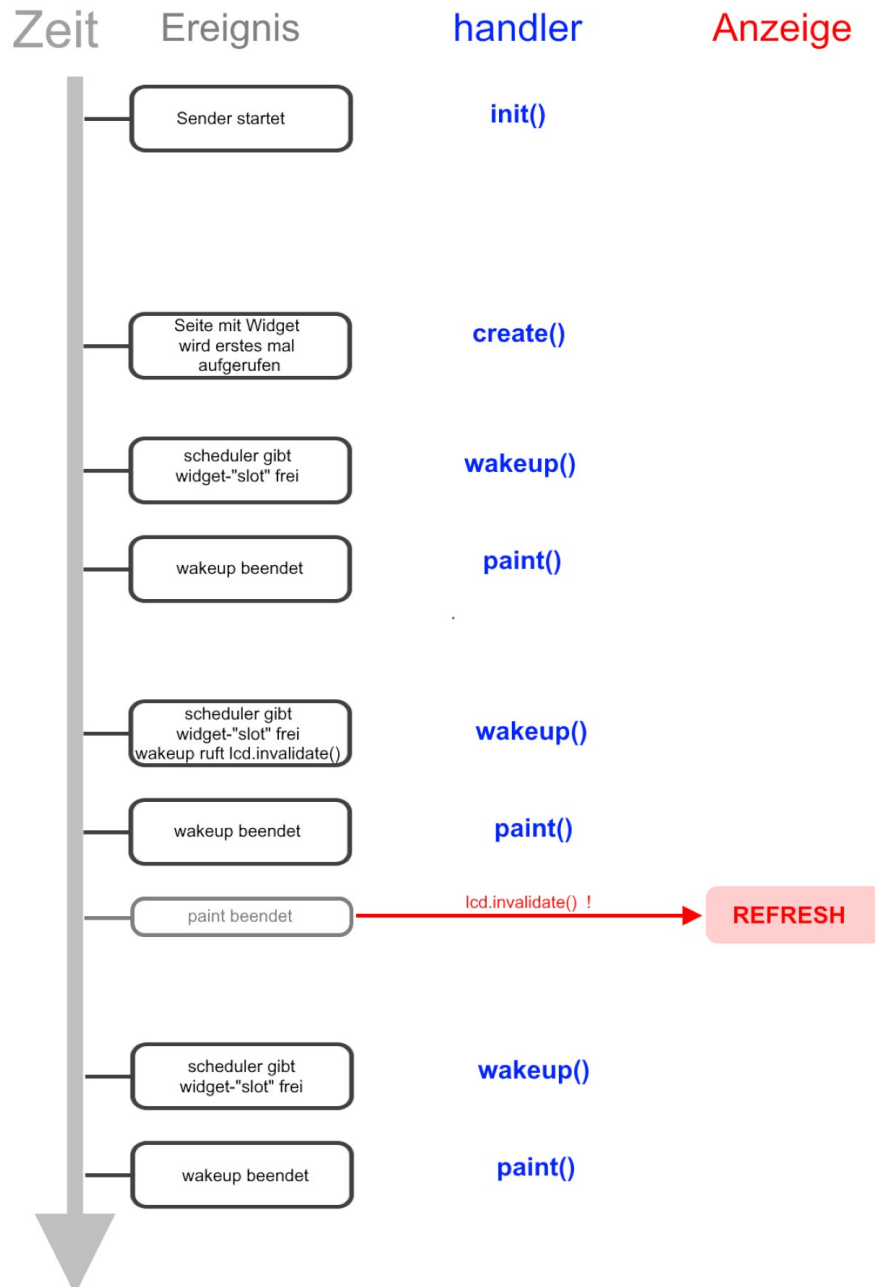
Handler, der bei der Erstellung des Kontextmenüs aufgerufen wird, um das Hinzufügen weiterer Optionen zu ermöglichen

close(widget)

Handler der aufgerufen wird, wenn die aktuelle Seite geschlossen wird

Ablaufdiagramm handler

Um den Ablauf grob zu verdeutlichen eine kleine Grafik, wann die handler aufgerufen werden.



Hier wird auch ersichtlich, dass die Bildschirmanzeige erst durch den Aufruf der `lcd.invalidate()` Methode aufgefrischt wurde.

Der wakeup handler sollte immer auch kontrollieren, ob sich auf der Anzeige etwas geändert hat (Werte etc..) und nur dann den aufwendigen refresh durch `lcd.invalidate()` triggern.

(Beim allerersten widget Aufruf frischt Ethos übrigens ebenfalls das Display selbst auf)

Startprozesse

Während des Ethos-Bootvorgangs sucht das Betriebssystem innerhalb der /scripts Ordners nach Dateien mit dem Suffix .lua (bzw .luac, falls bereits compiliert)

Diese werden eingelesen und der jeweilige init handler wird ausgeführt.

Dabei wird die UID/key, Name des scripts, Typ und die hinterlegten handler dem Sender zentral bekannt gemacht.

Danach werden alle vorhandenen Ordner direkt unterhalb von „/scripts“ geöffnet.

Dort wird jeweils nach einer Datei „main.lua“ bzw „main.luac“ gesucht, und falls vorhanden, hier der init handler ausgeführt.

Weitere Unterordner werden NICHT durchsucht.

Im Bootvorgang findet eine einfache Syntaxprüfung des scripts statt. Schlägt diese fehl, wird das script dem Sender nicht zur Verfügung gestellt.

Auf diese Weise besitzt der Sender nach Ende des Bootvorgangs einen Katalog **aller** gültigen scripte und kann sie dem User bei Bedarf vorschlagen.

In etlichen Forenbeiträgen wird davon gesprochen, dass Ethos alle scripte komplett laden würde, als ob sie alle permanent im Speicher allokiert **seien**.

Das ist definitiv nicht so.

Die Arbeitsumgebung

Ich weiß, Ihr wollt endlich loslegen.

Trotzdem noch ein paar einführende Worte, wie man sich für die lua Entwicklung eine „Arbeitsumgebung“ schafft, die die Vorteile vorhandener tools nutzt.

Insbesondere zu den Themen Editoren und dem Ethos Simulator.

Editoren

Wer sowieso bereits programmiert, wird seinen „Lieblingseditor“ weiter nutzen wollen.

Wer eher am Anfang steht und sich fragt, welchen Editor er nutzen sollte, dem möchte ich zwei (Win) Empfehlungen geben:

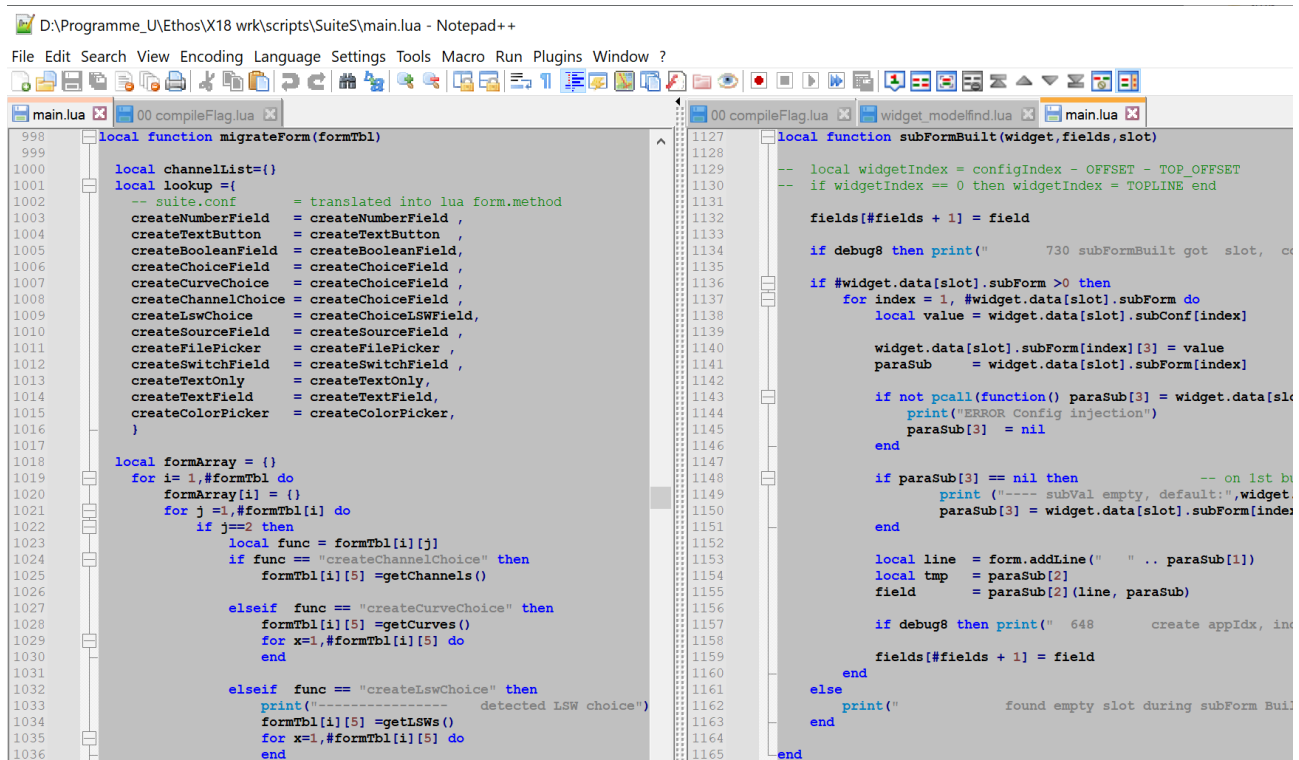
Notepad++ (Win)

Ein openSource / free Software Projekt das weit verbreitet ist.

Vorteile sind einfache Anwendung, „lua“ enabled (farbliche Markierungen / Hervorhebungen des Codings), portable, etliche Zusatz AddOns etc..

Definitiv weit verbreitet und willkommen, wenn schnelle Einarbeitung und einfache Anwendung über der Anzahl der Features steht.

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



```
998 local function migrateForm(formTbl)
999
1000     local channelList={}
1001     local lookup={
1002         -- suite.conf          = translated into lua form.method
1003         createNumberField     = createNumberField ,
1004         createTextButton      = createTextButton ,
1005         createBooleanField     = createBooleanField ,
1006         createChoiceField      = createChoiceField ,
1007         createCurveChoice      = createChoiceField ,
1008         createChannelChoice     = createChoiceField ,
1009         createLswChoice        = createChoiceLswField ,
1010         createSourceField      = createSourceField ,
1011         createFilePicker       = createFilePicker ,
1012         createSwitchField      = createSwitchField ,
1013         createTextOnly         = createTextOnly ,
1014         createTextField        = createTextField ,
1015         createColorPicker      = createColorPicker ,
1016     }
1017
1018     local formArray = {}
1019     for i=1,#formTbl do
1020         formArray[i] = {}
1021         for j=1,#formTbl[i] do
1022             if j==2 then
1023                 local func = formTbl[i][j]
1024                 if func == "createChannelChoice" then
1025                     formTbl[i][5] = getChannels()
1026
1027                 elseif func == "createCurveChoice" then
1028                     formTbl[i][5] = getCurves()
1029                     for x=1,#formTbl[i][5] do
1030
1031                     end
1032
1033                 elseif func == "createLswChoice" then
1034                     print("----- detected LSW choice")
1035                     formTbl[i][5] = getLSWs()
1036                     for x=1,#formTbl[i][5] do
1037
1038                     end
1039
1127 local function subFormBuilt(widget,fields,slot)
1128
1129     -- local widgetIndex = configIndex - OFFSET - TOP_OFFSET
1130     -- if widgetIndex == 0 then widgetIndex = TOPLINE end
1131
1132     fields[#fields + 1] = field
1133
1134     if debug8 then print("          730 subFormBuilt got slot, co
1135
1136     if #widget.data[slot].subForm > 0 then
1137         for index = 1, #widget.data[slot].subForm do
1138             local value = widget.data[slot].subConf[index]
1139
1140             widget.data[slot].subForm[index][3] = value
1141             paraSub = widget.data[slot].subForm[index]
1142
1143             if not pcall(function() paraSub[3] = widget.data[slot].subForm[index][3] then
1144                 print("ERROR Config injection")
1145                 paraSub[3] = nil
1146             end
1147
1148             if paraSub[3] == nil then
1149                 print ("---- subVal empty, default:", widget.data[slot].subForm[index][3])
1150                 paraSub[3] = widget.data[slot].subForm[index][3]
1151             end
1152
1153             local line = form.addLine(" " .. paraSub[1])
1154             local tmp = paraSub[2]
1155             field = paraSub[2] (line, paraSub)
1156
1157             if debug8 then print("          648 create appIdx, in
1158
1159             fields[#fields + 1] = field
1160         end
1161     else
1162         print("          found empty slot during subForm Built
1163     end
1164 end
1165 end
```

Visual Studio Code (Win)

Wie der Name vermuten lässt aus dem Hause Microsoft

VS Code ist ebenfalls kostenlos, funktional noch umfangreicher als Notepad++.

Wer seine Projekte in github publizieren möchte, ist bei VS Code erst recht aufgehoben, da hier addOns existieren um Dateien direkt mit git abzugleichen (push/pull/commit etc..).

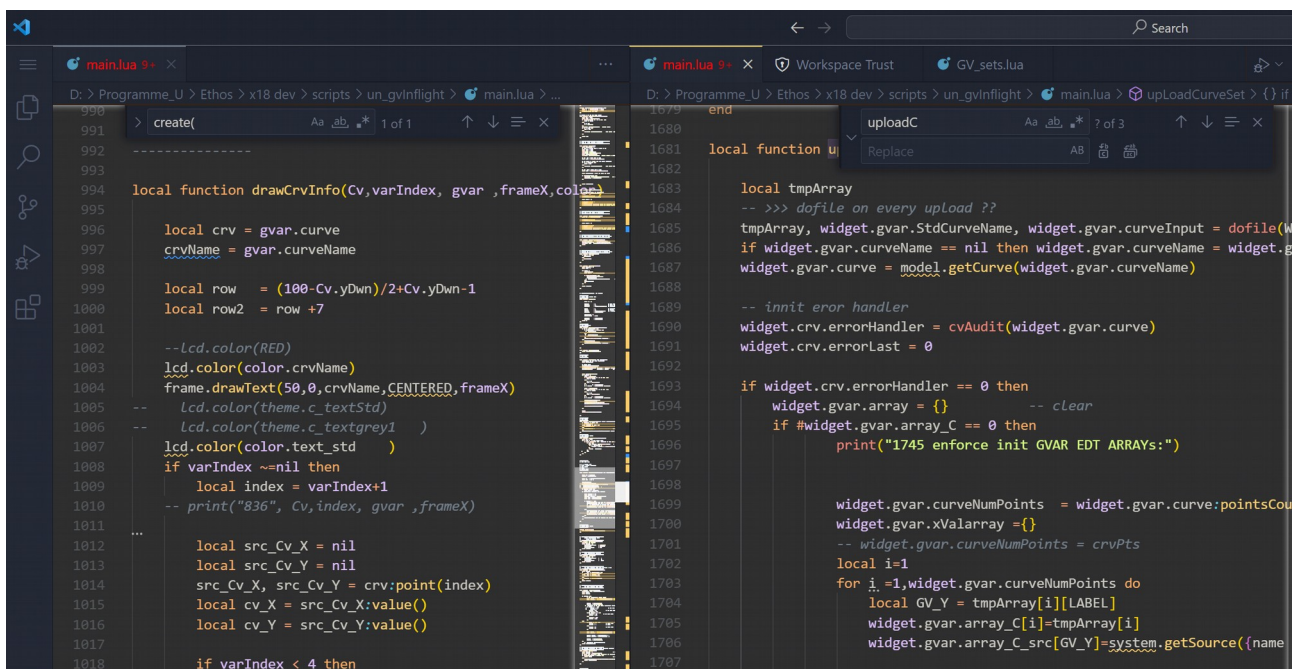
VS Code lässt sich extrem individualisieren, dazu ist aber eine entsprechende Einarbeitung in die Config Files notwendig.

Die Lua Code Unterstützung „out of the box“ ist noch etwas umfangreicher als bei notepad++, z.B. sieht man bereits farblich, ob definierte Variablen oder Funktionen im Coding überhaupt genutzt werden (farblich ausgegraut) etc..

VS Code besitzt mehr Funktionalitäten als notepad++, man benötigt allerdings auch mehr Einarbeitungszeit.

VS Code ist für diverse Betriebssysteme verfügbar (Win / Linux / IOS)

<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 eror 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
```

Nutzung des Simulators zur Entwicklung

Allgemeines

Der Ethos Simulator ist ein hervorragendes Tool, um seine scripte während der Entwicklungsphase zu testen, ohne den Sender zu benötigen.

Zum einen besitzt der PC / das Notebook / der Mac deutlich mehr Performance, insgesamt „arbeitet“ man etwas flüssiger als direkt am Sender, „Coden > Testen > Analysieren > Coden“ Zyklen sind am Sim einfach schneller.

Zudem „nutzt“ sich der Sender nicht so schnell ab.

Zum anderen hat man natürlich die Einschränkung, dass nicht restlos alle Funktionen (z.B. Trimbedienung, bestimmte Telemetrie Situationen) nachgestellt werden können (Stand Ethos 1.5.x).

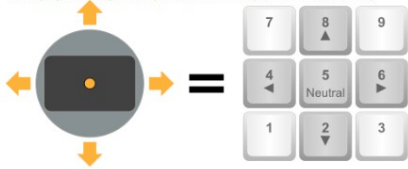
Ich persönlich nutze zu rund 90% der Entwicklungsaufwendungen den Sim.

Der aktuellste Sim ist im jeweils aktuellen Ethos Release (latest) auf Github zu finden


Bedient wird der Sim hauptsächlich via Maus

Bedienelemente wie Schalter, Sticks etc werden über Tastatur emuliert

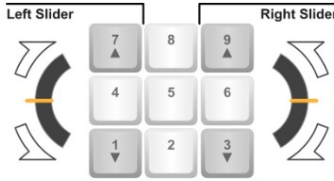
• How to move sticks
To move the left stick, use numeric keyboard as shown below.
To move the right stick, use "Control" key + numeric keyboard as shown below.
To simplify, currently the steps are set to 20% (0, 20, 40, 60, 80, 100).



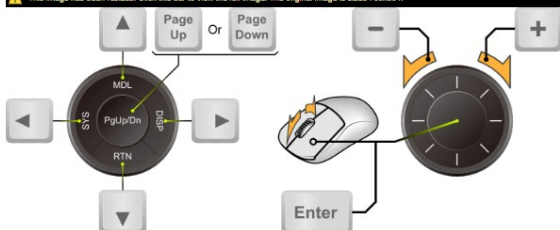
• How to change the switches position
To toggle switch A, just press A. The switch A takes successively its possible physical positions. Here is an example of how it works:
A- = [Joystick icon] = A
▶▶ Same operation for all the other switches (A, B, C, ..., I, J)



• How to move sliders and knobs
To move left slider or right slider, use numeric keyboard as shown below.
To move knob P1 or knob P2, use "Control" key + numeric keyboard as shown below.
To simplify, currently the steps are set to 20% (0, 20, 40, 60, 80, 100).
Note: No command is provided for knob P3. There no key to set slider or knob to neutral.



• Other keyboard shortcuts to navigate in the user interface
This image has been resized. Click this bar to view the full image. The original image is sized 730x504.



<https://www.rcgroups.com/forums/showpost.php?p=47748621&postcount=10729>

Telemetrie

Um am Sim die Telemetrie nachstellen zu können (nur Fixwerte), muss am Sim das RF Modul eingeschaltet sein. Sollte der Modellspeicher noch keine Telemetriesensoren besitzen, dann diese im Telemetriemenü per Suche definieren. Es werden alle typischen Sensoren gefunden.

Ordnerstruktur & dedizierte Arbeitsumgebungen

Hinweis !

Ab Version 1.5.12 wurden die Verwendung von Ordnern bzw die Pfade erheblich umgestaltet.

Zuvor wurden Modellfiles zentral in der Benutzerumgebung abgelegt, der Sim selbst lief mit allen weiteren Ordnern (audio / bitmaps / scripts....) in seinem Installationspfad.

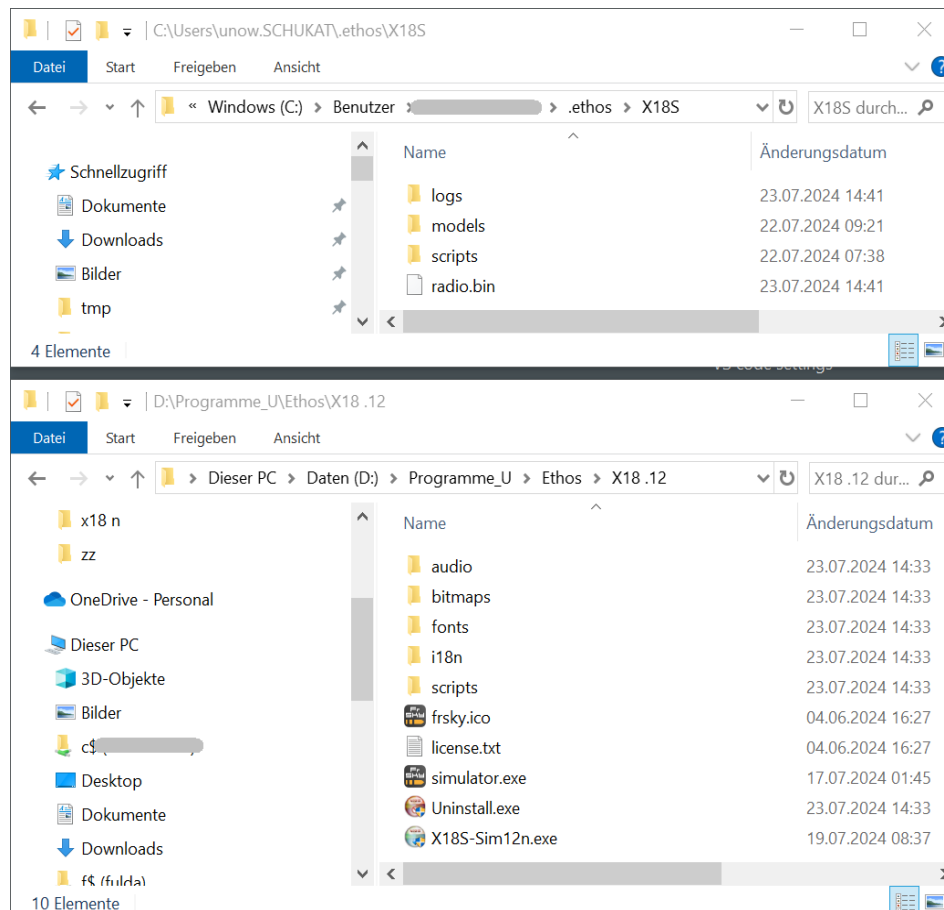
Für jede „Umgebung“, wie z.B. spezifische Sim releases , betas, Arbeits-, Entwicklungsumgebungen etc.. war eine eigene Installation notwendig, die sich ein Modellverzeichnis teilen.

Dies führte bei gleichzeitiger Verwendung von Sims unterschiedlicher Releasestände zu Problemen.

Nun (1.5.12) wird eine Hardware spezifische „Userumgebung“ unter dem Ordner „.ethos“ im Windows Benutzerpfad angelegt

Sie enthält die Ordner logs, models und scripts, sowie die radio.bin Datei inkl. Sendersettings

Alle restlichen Ordner befinden sich im Installationspfad:



Ab 1.5.12 kann beim Start eines Sims über Parameter bestimmt werden, welcher spezieller Ordner für welchen speziellen Zweck (audio, system, scripts, ...) verwendet werden soll (re-routing)

die Parameter sind:

- `--system-directory` System Verzeichnis (z.B. i18n, bitmaps)
- `--user-directory` die typischen user verzeichnisse (logs, models..)
- `--scripts-directory` Ordner für scripts
- `--radio-settings` legt dedizierte „radio.bin“ Datei fest

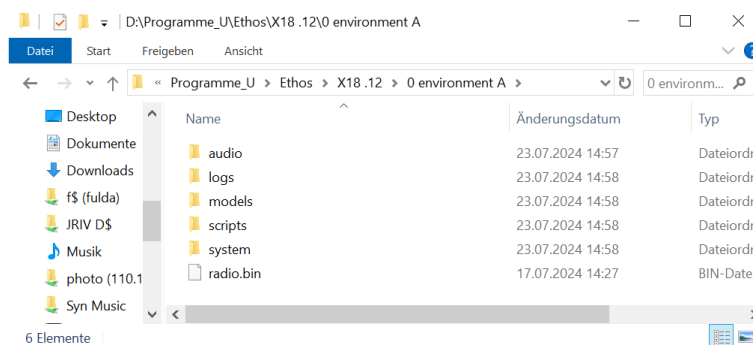
eine Batchdatei zum Start des Sims könnte daher lauten:

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"
```

Der Ordner der Arbeitsumgebung „**0 environment A**“ sollte dann so aussehen:
(„eigene“ soundfiles, modellfiles, scripts, radio.bin)



Eine weitere Arbeitsumgebung ließe sich einfach und ohne zusätzliche Sim Installation durch eine weitere, angepasste batch Datei und einen weiteren „Umgebungs-Ordner“,

z.B „0 environment B“

erstellen

- debug Fenster

SEHR WICHTIG !!!

Der Sim kann ein debug Fenster zur Verfügung stellen.

Dort werden alle Systemmeldungen inkl. Lua debug Code und Fehler präsentiert.

(Lua debug Code ist nichts anderes als ein print Befehl in bestimmten Code Zeilen um Statusinfos & Werte im Ablauf auszugeben)

Windows:

um das debug Fenster im Sim zu erhalten, muss der Sim per Command shell gestartet werden:

z.B.

- command shell öffnen (cmd.exe)
- in das Verzeichnis des Sims navigieren
- Eingabe „simulator.exe“

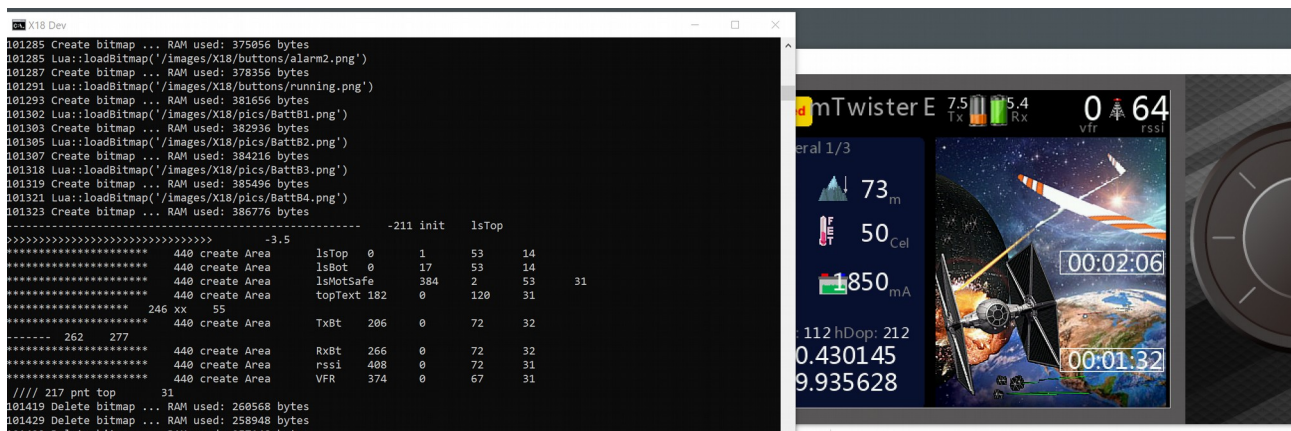
Alternativ ein batch file generieren und starten, z.B. für einen Sim, der unter „D:\Programme\Ethos\X18“ installiert wurde:

d:

cd "D:\Programme\Ethos\X18"

"D:\Programme\Ethos\X18\simulator.exe"

Links das debug Fenster zum gestarteten Sim (rechts, Teilausschnitt)



Ethos Suite

Die Ethos Suite kann ebenfalls während der lua script Entwicklung genutzt werden.

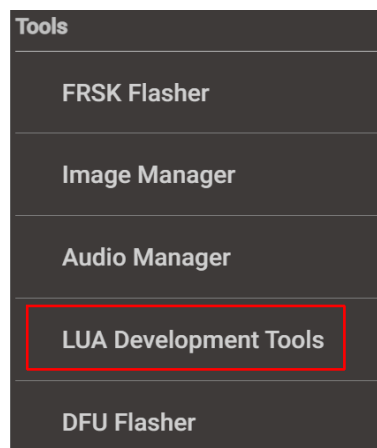
Sie wird meistens „on top“ zum Simulator angewendet.

Der Simulator stellt z.B. ausschließlich Konstante Telemetriewerte zur Verfügung, kann derzeit Trimmer Eingaben nicht entsprechend verarbeiten, und zeigt ein anderes Performanceverhalten auf.

Ab dem Moment, wo ein „real world“ Testing notwendig ist, kommt die Suite zum Einsatz.

Dabei wird der Sender (z.B. im Suite modus) mit dem Rechner verbunden und die Suite gestartet.

Durch Verwendung der Suite im „dev-tool Menue“ kann der Sender durch die Suite gesteuert zwischen Betriebsmodus und Entwicklermodus umgeschaltet werden.



Der „Entwicklermodus“ ist quasi der suite Modus mit gemounteten Laufwerken, so dass die lua Files einem Editor zur Verfügung stehen.

Ist ein lua File des Senders im Editor geöffnet, sollte der Editor in der Lage sein das File im Speicher zu halten, auch wenn die Datei im Betriebsmodus des Senders gerade nicht zur Verfügung steht.



Durch diese Art der Umschaltung lassen sich schnell Anpassungen im Code vornehmen und praktisch testen.

Es muss kein Kabel umgesteckt werden, bzw der Sender manuell in Suite / seriell Mode etc.. versetzt werden.

Der besondere und wesentliche Vorteil liegt darin, dass einem direkt ein Terminalfenster zur Verfügung steht, in dem debug Informationen des Senders in Echtzeit dargestellt werden können.

Diese debug Infos sind außerordentlich hilfreich bei der Suche nach Fehlern und deren Ursachen.

ACHTUNG, Sicherheitshinweis !

Wenn im Testmodus auch das Modell eingeschaltet werden sollte:

Um beim Umschalten Schaden durch versehentliches Anlaufen von Antriebsmotoren etc. zu verhindern, müssen Propeller demontiert werden und ggf. , je nach Modellkonfig weitere Sicherheitsmaßnahmen getroffen werden.

Debuggen der Tx Infos

Debug Infos

Gerade wurde von debug Information gesprochen.

Was versteht man darunter ?

Allgemein gesagt ist debug Information alles an (textlicher) Information des Senders / des Sims, was dem Entwickler hilft Fehler und deren Ursache aufzuspüren.

Beispiel:

der Sim zeigt im Script einen Fehler an:

```
←[31m120782 wakeup() error: /scripts/un_gvInflight/main.lua:2507: attempt to index a nil value (local 'source')  
die Zeile 2507 im main.lua konnte also nicht ausgeführt werden.
```

Schauen wir uns den Codebereich an:

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

Es sollte der Wert einer Telemetriequelle (source) der Variablen sourceVal zugewiesen werden, das funktionierte nicht.

Offensichtlich ist die Quelle nicht angelegt worden (Fehlermeldung „...nil value (local 'source')“)

Schauen wir uns an, welchen Namen der Sensor in dem Moment hatte. Dazu ergänzen wir den Code und lassen uns via print Befehl etwas „debug Infos“ ausgeben:

```
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
```

Der Sim gibt nun aus:

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

Als Name eines Telemetriesensors ist „5“ bestimmt falsch, sorgen wir im Coding also für den richtigen Sensornamen (z.B. „RSSI“), ist der Fehler behoben.

Im debug Fenster des Sims werden solche Infos einfach ermöglicht.

Wie aber erhalte ich debug Infos direkt vom Sender ?

Eine Möglichkeit wurde gerade beschrieben: via Suite und den dev-tools.

Dort existiert ein Terminalfenster, was die Informationen darstellt.

Welche Alternative gibt es?

Dazu muss man wissen, dass diese Infos als eine Art „serielle Textübertragung“ via USB gesendet werden.

Die Technik der seriellen Übertragung gibt es fast seit Anfang der Computertechnik, ein HW Standard der seriellen Übertragung war einmal die RS232 Schnittstelle, die immer noch via USB emuliert werden kann.

Zur Darstellung verwenden wir mit Sicherheit kein antikes Textterminal (es sei denn jemand besitzt z.B. noch DEC VT510 Geräte..) sondern wir überlassen das einer Software App.

Windows (putty):

Sehr geläufig und kostenlos ist das Programm „putty“

<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

Diese App visualisiert die Daten (Textnachrichten) , die aus dem Sender kommen.

Die Daten werden in einer bestimmten Geschwindigkeit gesendet und „landen“ auf einer bestimmten „virtuellen“ Schnittstelle per Emulation.

Ist das Programm putty installiert, muss zunächst diese Kommunikations-Schnittstelle eingerichtet werden.

Prinzipiell ist es möglich, dass mehrere Geräte zeitgleich seriell mit einem Rechner kommunizieren wollen.

Jedes Endgerät muss dazu eine eigene Schnittstelle „bedienen“.

Die frühere physikalische RS232 bzw „Com“ (Communication) Schnittstellen waren am Host daher mehrfach vorhanden und durchnummeriert (Com1: bis Com32: und mehr..)

Die USB Schnittstelle emuliert nun einen bestimmten Port, je nachdem auf welchem USB Port der Sender verbunden wurde, welche OS Version, welcher Treiber..

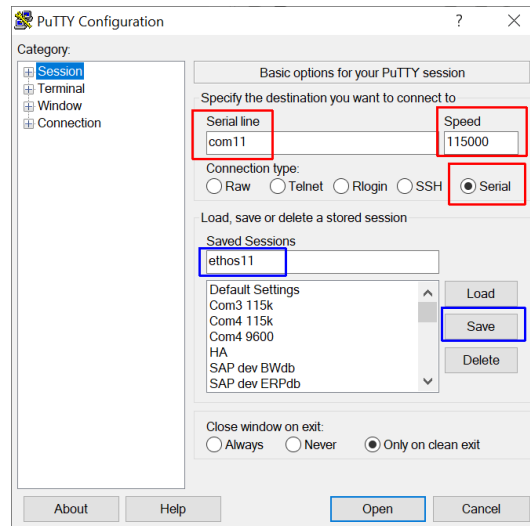
Wir müssen via Gerätemanager herausfinden, auf welchem Port der Sender zugewiesen wurde.

Dazu wird der Sender mit dem Rechner verbunden und in den seriellen Modus versetzt.

Wir öffnen deshalb den Gerätemanager und schauen nach, auf welche sogenannten COM Schnittstelle der Sender verbunden wurde.

Unter putty definieren wir nun eine Sitzung mit genau dieser Com Schnittstelle und den Parametern 115000Baud (Geschwindigkeit)

Diese Angaben sind im Bild rot gemarkert:



Wir speichern die Sitzung unter einen eindeutigen Namen (z.B. „ethos 11“) und drücken „Save/Speichern“ (blau)

Jetzt kann man per „öffnen/open“ ein Terminalenster starten, das uns Infos vom Sender ausgibt.

Weitere Informationsquellen

Ethos Lua Reference Guide

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

Der Ethos Lua Reference Guide stellt die offizielle Dokumentation seitens FrSky dar.

Es ist eine Art Nachschlagewerk bzw Übersicht aller aktuell verfügbarer Klassen und Methoden unter Ethos Lua. Des weiteren sind die Systemkonstanten aufgeführt (Kategorie „base“).

Üblicherweise steigt man via Klassenübersicht ein und schaut sich innerhalb einer Klasse die verfügbaren Methoden an um herauszufinden, was an Funktionalitäten innerhalb einer Klasse umgesetzt werden kann.

Häufig ist zu einer Methode ein Beispiel aufgeführt.

Github – Ethos community

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

Wie bekannt stellt FrSky auf Github die neuesten Software Versionen zur Verfügung, und mittels des issue Managements können Erweiterungen angefragt bzw. Fehler gemeldet werden.

Auf der Einstiegsseite des sogenannten Repos befindet sich eine git-typische Navigationsstruktur mit Dateien und Ordnern.

Darunter auch der Ordner „lua“

Hier sind etliche komplette Beispiel luas zu verschiedenen Themen wie Formulare („tool-form“), grafische Ansteuerung („widget-lcddemo“) und vieles mehr als Anschauungsobjekt abgelegt.

Lua Basic Tutorial

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

Wer noch gar keine lua / Programmierkenntnisse besitzt findet hier ein Online Tutorial für die allgemeinen Programmierkenntnisse.

Das Tutorial ist natürlich allgemein gehalten und geht nicht auf die spezifischen Ethos lua Erfordernisse ein.

Grundlagen zu Themen wie Variablen, Datentypen, Operatoren, allgemeiner Syntax etc. werden anschaulich dargestellt.

Offizielle Lua Seite

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

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

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

Die offizielle Lua Seite.

Eine gut strukturierte Informationsquelle über Lua mit Listung aller grundlegenden Funktionen.

Besonders erwähnenswert ist die Übersicht über die Bibliotheken mit ihren (teilweise versionsabhängigen) implementierten Methoden.

Diese sind zum Beispiel wichtig im Umgang mit Zeichenketten („string“), mathematischen Funktionen („math“), IO Operationen (io) etc..

Engel Forum: -Lua Skripts

<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/>

Das sehr aktive deutsche Forum besitzt auch Sektionen zur Diskussion über Lua Themen, sowie einen Bereich wo fertige lua scripte durch Autoren eingestellt werden können.

RCG Lua thread

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

Das internationale „Pendant“ zum Engel Forum.

Auch hier gibt es regelmässig Diskussionen um das Thema Ethos lua.

Auf der Einstiegsseite wurden einige scripte durch Mike verlinkt.

Lua Performance Tips

Ein hervorragendes Dokument über Performanceoptimierung unter Lua findet man hier:
<https://www.lua.org/gems/sample.pdf>

Hiermit ist das erste Kapitel (allgemeiner Einstieg) beendet.

Weiter geht es endlich mit einem kleinen, praktischen Übungsbeispiel.

Das erste selbst programmierte lua widget

Glossar

Array

Eine geordnete Sammlung von Elementen, die durch Indizes adressiert werden. In Lua werden Arrays als Tabellen implementiert, wobei die Indizes normalerweise bei 1 beginnen.

Compiler

Ein Programm, das Quellcode in eine ausführbare Datei umwandelt. Lua verwendet jedoch hauptsächlich einen Interpreter, aber es gibt Compiler wie LuaJIT, die Lua-Code kompilieren können.

Debug

Allgemein der Vorgang Fehler aus dem Coding zu entfernen. Unter Ethos wird weitestgehend notwendige Information (Werte, Ablaufinfos etc..) via print Befehl in das Shell-Fenster des Sims bzw via serielle Ausgabe zur Verfügung gestellt.

Float

Ein Datentyp für Zahlen mit Dezimalstellen. In Lua sind alle Zahlen standardmäßig Gleitkommazahlen (floats).

Frame

Ethos-Lua: die Fläche / der Rahmen, in dem ein script im HomeScreen läuft.

Für das script ist es der „Arbeitsbereich“ und „Ereignishorizont“

Das skript kennt nicht „die Welt“ außerhalb des Frames

Github

Github (<https://github.com/>) ist eine webbasierte Plattform zur Versionsverwaltung und **gemeinsamen** Entwicklung von Softwareprojekten. Es verwendet Git, ein verteiltes Versionskontrollsystem, um Änderungen am Code nachzuverfolgen.

Entwickler können Repositories erstellen, Code teilen, Fehler melden und gemeinsam an Projekten arbeiten.

Benutzer können ebenfalls Fehler oder Erweiterungswünsche mitteilen.

Es stehen Wikis zur Verfügung.

GitHub bietet zudem Funktionen wie Projektmanagement-Tools, Wikis und Continuous Integration/Continuous Deployment (CI/CD).

Handler

allgemein:

Eine Funktion oder Methode, die aufgerufen wird, um ein bestimmtes Ereignis oder eine bestimmte Bedingung zu verarbeiten.

handler werden durch ein Ereignis getriggert, z.B:

- Touch Eingabe oder Bedienung einer Steuerungstaste (event),
- Aufruf des Konfigurations Menüs (configure),
- Änderung einer Widget Konfig (read/write),
- oder wenn in der internen Ablauf die Systemloops (z.B. Mixerverarbeitung, LSW's, SF's ..) abgeschlossen wurde (wakeup)

Welche handler im script existieren, wird während der Registrierung als Argument definiert.

Integer

Ein Datentyp für ganze Zahlen ohne Dezimalstellen. In Lua können ganze Zahlen auch als Gleitkommazahlen dargestellt werden, aber es gibt Bibliotheken, die explizite Ganzzahloperationen unterstützen.

Interpreter

Ein Programm, das Quellcode direkt ausführt, anstatt ihn zu kompilieren. Lua wird standardmäßig interpretiert.

Key

Ein eindeutiger Bezeichner in einer Tabelle, der verwendet wird, um Werte zu speichern und abzurufen. In Lua können Schlüssel beliebige Werte außer nil sein, häufig sind es Zeichenketten oder Zahlen.

Klasse

Ein Konstrukt in der objektorientierten Programmierung, das eine Blaupause für Objekte definiert, einschließlich deren Eigenschaften und Methoden. Lua unterstützt objektorientierte Programmierung durch Tabellen und Metamethoden.

Methode

Eine Funktion, die innerhalb einer Klasse oder eines Objekts definiert ist und auf die Daten des Objekts zugreift oder diese manipuliert.

MPU

Abkürzung für Mikroprozessor oder Microcontroller Processing Unit. In der „embedded“ Programmierung bezieht sich dies auf den zentralen Prozessor einer „embedded application“.

Objektorientierte Programmierung (OOP)

Ein Programmierparadigma, das auf Objekten basiert, die Daten (Attribute) und Funktionen (Methoden) kapseln. Lua unterstützt OOP durch die Verwendung von Tabellen und Metamethoden.

Pointer

Ein Zeiger ist eine Variable, die die Adresse eines anderen Werts speichert. Lua hat keine expliziten Zeiger wie C/C++, aber Tabellenreferenzen können ähnlich wie Zeiger verwendet werden.

Registrierung

durch die Registrierung wird das script dem Ethos System bekannt gemacht.

Während des Bootvorgangs sucht der Sender im /scripts Verzeichnis nach allen ausführbaren scripten, überprüft die Syntax auf grobe Fehler und registriert sie, falls die Prüfung plausibel war.

Dazu wird der Initialisierungs handler ausgeführt.

Dieser gibt an, welche UID/Key, welchen Namen, von welchem Typ das script ist (widget, source, task...) und definiert die Liste der verwendeten handler.

Hier wird z.B. ein widget namens "Vartst" mit diversen handler registriert, die optionale Titeldarstellung wurde grundsätzlich abgestellt (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

Die "UID" (unique identifier) oder der "key" ist ein sogenannter eindeutiger Schlüssel.

Durch Ihn identifiziert das Ethos Betriebssystem das script

Selbst wenn der Namen des scripts verändert wird, kann Ethos das script weiter "auffinden"

Die UID darf daher kein zweites mal in einem anderen script verwendet werden.

Verwendet wird sie im init handler während der Registrierung

Widget

Oft ein grafisches Steuerelement in einer Benutzeroberfläche, wie z. B. eine Schaltfläche, ein Textfeld oder ein Menü.

In Lua sind Widgets kleine Apps, die in einem Frame laufen und von GUI-Framework organisiert werden.