OpenTX 2.2 Lua Reference Guide

Published with GitBook

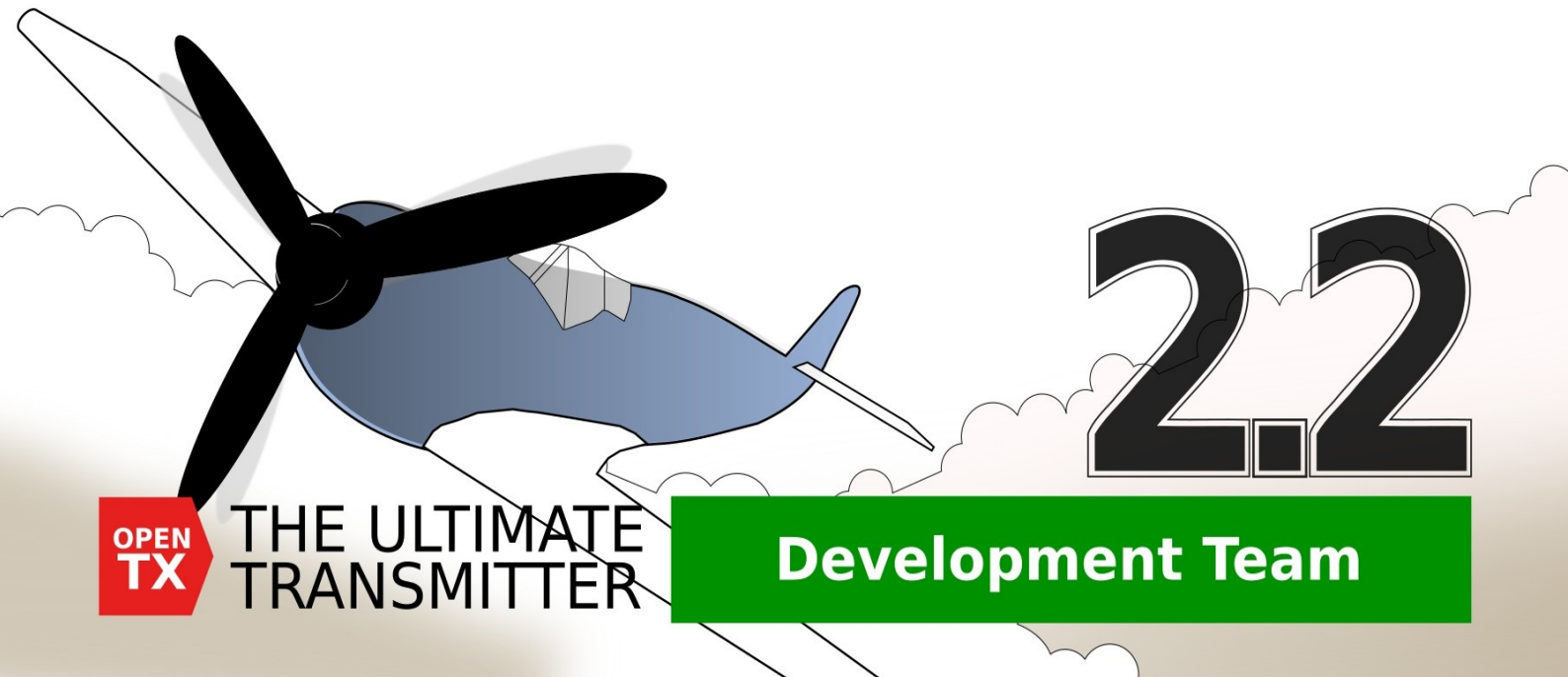


Table of Contents

OpenTX 2.2 Lua Reference Guide

Join the chat at https://opentx.rocket.chat

Go to https://opentx.gitbooks.io/opentx-2-2-lua-reference-guide/content/ for the latest published version of this guide.

This guide covers the development of user-written scripts for R/C transmitters running the OpenTX 2.2 operating system with Lua support. Readers should be familiar with OpenTX, the OpenTX Companion, and know how to transfer files the SD card in the transmitter.

Part I of the guide shows how to enable Lua support for Taranis and includes basic examples of each types of script.

Part II is a programming guide that introduces the types of OpenTX Lua scripts and how to use them.

Part III is the OpenTX Lua API Reference

Part IV addresses common issues in converting Lua scripts that were originally written for OpenTX 2.0

Part V addresses common issues in converting Lua scripts that were originally written for OpenTX 2.1

Part VI covers advanced topics with examples

last updated on 2017/08/27 14:51:27 UTC

Introduction

This section includes Acknowledgments and Getting Started.

Acknowledgments

The OpenTX team has no intention of making a profit from their work. OpenTX is free and open source and will remain free and open source. But OpenTX is more expensive to maintain than most open source projects. The reason is that there is a never ending flood of hardware to integrate and maintain code for. Hardware that costs.

Another reason is that OpenTX maintains a build server that serves firmware compiled on demand. This is where OpenTX Companion orders your customized firmware. The server is not for free and the bandwidth is ever increasing with tens of thousands of firmware downloads each month.

The OpenTX team is grateful to those who have donated to the project. You have helped making OpenTX and OpenTX Companion great.

The Github Donor List is updated at each OpenTX release.

If you would like to contribute to OpenTX, donations are welcome and appreciated:

Getting Started

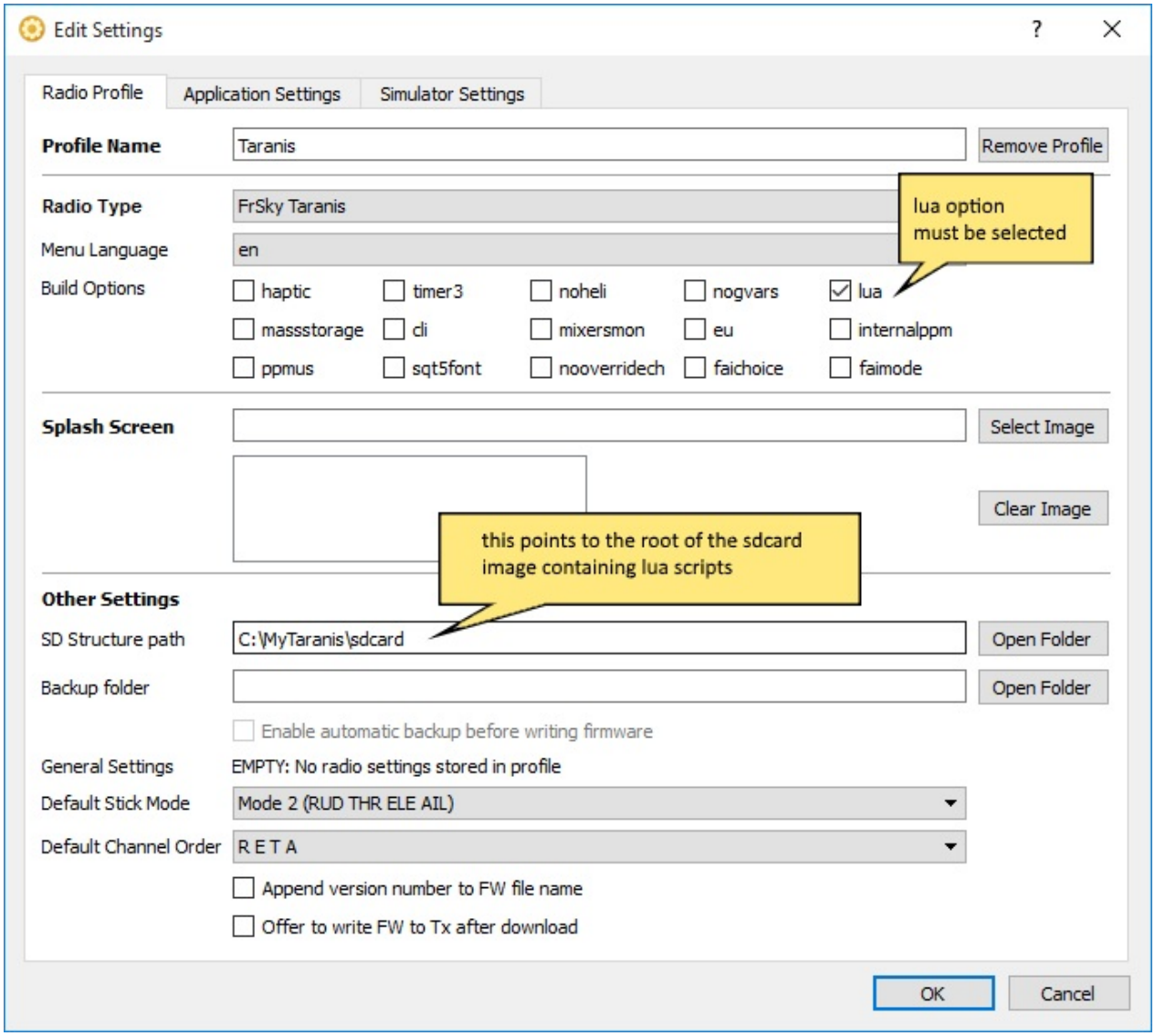
Downloading OpenTX Companion

OpenTX Companion 2.2 is available for download at http://www.open-tx.org/downloads.html

Updating firmware with Lua option selected

If you intend to use mixer scripts, when updating the firmware on your transmitter you need to make sure the lua option is checked in the settings for your radio profile (Main menu -> Settings ->Settings...) as shown below. This is not required if you only intend to run telemetry, one-time and function scripts, support for those is included by default.

Also note that the SD Structure path should contain a valid path to a copy of your transmitter's SD card contents, although that's not specific to Lua.



Edit Settings dialog from OpenTX Companion

Part I - Script Type Overview

This section introduces the types of Lua scripts supported by OpenTX and how they may be used.

Mix Scripts

***WARNING* - Do not use Lua mix scripts for controlling any aspect of your model that could cause a crash if script stops executing.**

Description

Each model can have several mix scripts associated with it. These scripts are run periodically for entire time that model is selected. These scripts behave similar to standard OpenTX mixers but at the same time provide much more flexible and powerful tool.

Mix scripts take one or more values as inputs, do some calculation or logic processing based on them and output one or more values. Each run of a script should be as short as possible. Exceeding the script execution runtime limit will result in the script being forcefully stopped and disabled.

Typical uses

- replacement for complex mixes that are not critical to model function
- complex processing of inputs and reaction to their current state and/or their history
- filtering of telemetry values

Limitations

- cannot update LCD screen or perform user input.
- should not exceed allowed run-time/ number of instructions.
- mix scripts are run with less priority than built-in mixes. Their execution period is around 30ms and is not guaranteed!
- can be disabled/killed anytime due to logic errors in script, not enough free memory, etc...)

Location

Place them on SD card in folder /SCRIPTS/MIXES/. File name length (without extension) **must be 6 characters or less** (this limit was 8 characters in OpenTX 2.1).

Lifetime

- script is loaded from SD card when model is selected
- script *init* function is called
- script *run* function is periodically called (inside GUI thread, period cca 30ms)
- script is killed (stopped and disabled) if it misbehaves (too long runtime, error in code, low memory)
- all mix scripts are stopped while one-time script is running (see Lua One-time scripts)

Disabled script

If as script output is used as a `mixer source` and the script is killed for what ever reason, then `the whole mixer line is disabled` ! This can be used for example to provide a fall-back in case Lua mixer script gets killed.

Example where Lua mix script is used to drive ailerons in some clever way, but control falls back to the standard aileron mix if script is killed. Second mixer line replaces the first one when the script is alive:

```
CH1  [I4]Ail Weight(+100%)
  := LUA4b Weight(+100%)
```

Script interface definition

Every script must include a *return* statement at the end, that defines its interface to the rest of OpenTX code. This statement defines:

- script *input* table (optional, see Input Table Syntax)
- script *output* table (optional, see Output Table Syntax)
- script *init* function (optional, see Init Function Syntax)
- script *run* function (see Run Function Syntax)

Example (interface only):

```
local input {}

local output {}

local function init_func()
end

local function run_func()
end

return { input=input, output=output, run=run_func, init=init_func }
```

Notes:

- inputs table defines input parameters (name and source) to run function (see Input Table Syntax)
- outputs table defines names for values returned by run function (see Output Table Syntax)
- init_func() function is called once when script is loaded.
- run_func() function is called periodically

Telemetry Scripts

General description

Telemetry scripts are used for building customized screens. Each model can have up to three active scripts as configured on the model's telemetry configuration page. The same script can be assigned to multiple models.

File Location

Scripts are located on the SD card in the folder /SCRIPTS/TELEMETRY/<*name*>.lua. File name length (without extension) **must be 6 characters or less** (this limit was 8 characters in OpenTX 2.1).

Lifetime of telemetry script

Telemetry scripts are started when the model is loaded.

- script init function is called
- script background function is periodically called when custom telemetry screen is **not visible**. *Notice*:
  - In OpenTX 2.0 this function is **not called** when the custom telemetry screen is visible.
  - Starting from OpenTX 2.1 this function is **always called** no matter if the custom screen is visible or not.
- script run function is periodically called when custom telemetry screen is **visible**
- script is stopped and disabled if it misbehaves (too long runtime, error in code, low memory)
- all telemetry scripts are stopped while one-time script is running (see Lua One-time scripts)

Script interface definition

Every script must include a return statement at the end, that defines its interface to the rest of OpenTX code. This statement defines:

- script **init** function *(optional)*
- script **background** function
- script **run** function

Example (interface only):

```
local function init_func()
  -- init_func is called once when model is loaded
end

local function bg_func()
  -- bg_func is called periodically (always, the screen visibility does not matter)
end

local function run_func(event)
  -- run_func is called periodically only when screen is visible
end

return { run=run_func, background=bg_func, init=init_func  }
```

Notes:

- `init_func()` function is called once when script is loaded and begins execution.
- `bg_func()` is called periodically, the screen visibility does not matter.
- `run_func(event)` function is called periodically when custom telemetry screen is visible. The `event` parameter indicates which transmitter button has been pressed (see Key Events). This is the time when the script has full control of the LCD screen and keys and should draw something on the screen.

One-Time Scripts

Overview

One-Time scripts start when called upon by a specific radio function or when the user selects them from a contextual menu. They do their task and are then terminated and unloaded. Please note that all persistent scripts are halted during the execution of one time scripts. They are automatically restarted once the one time script is finished. This is done to provide enough system resources to execute the one time script.

WARNING! -

- Running a One-Time script will suspend execution of all other currently loaded Lua scripts (Mix, Telemetry, and Functions)

File Location

Place them anywhere on SD card, the folder /SCRIPTS/ is recommended. The only exception is official model wizard script, that should be put into /SCRIPTS/WIZARD/ folder that way it will start automatically when new model is created.

Lifetime of One-Time scripts

Script is executed when user selects Execute on a script file from SD card browser screen.

Script executes until:

- it returns value different from 0
- is forcefully closed by user by long press of EXIT key
- is forcefully closed by system if if it misbehaves (too long runtime, error in code, low memory)

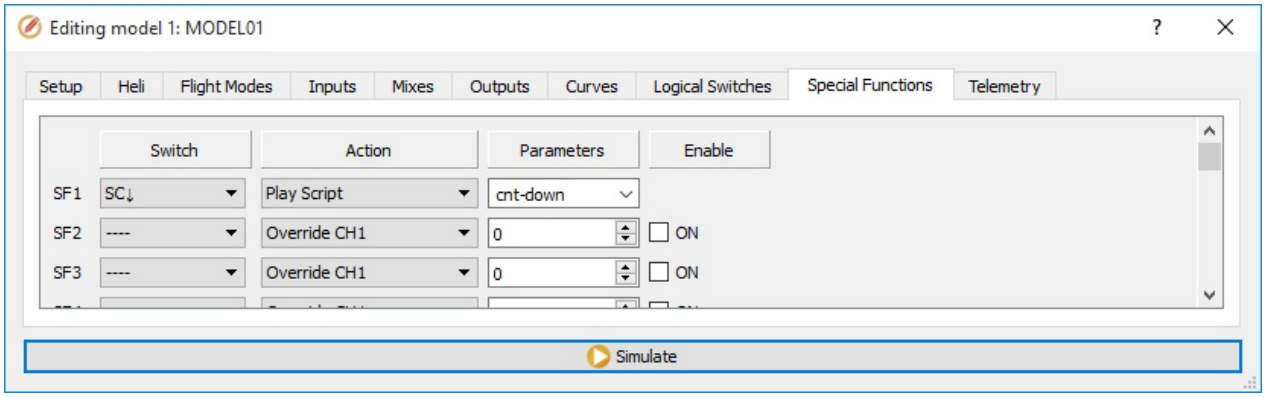
Wizard

TODO: Need to determine status of wizard in 2.2

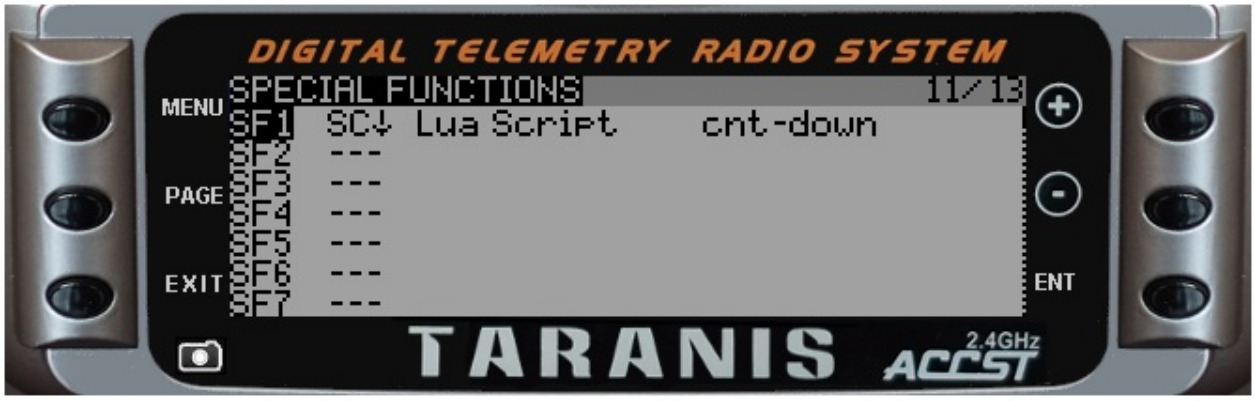
Function Scripts

Overview

Function scripts are invoked via the **'Lua Script'** option of Special Functions configuration page.



Companion Special Functions Window



Taranis Special Functions Display

Typical uses

- specialized handling in response to switch position changes
- customized announcements

Limitations

- should not exceed allowed run-time/ number of instructions.
- all function scripts are stopped while one-time script is running (see Lua One-time

scripts)
- Function scripts **DO NOT HAVE ACCESS TO LCD DISPLAY**

Location

Place them on SD card in folder /SCRIPTS/FUNCTIONS/

Lifetime

- script *init* function is called once when model is loaded
- script *run* function is periodically called as long as switch condition is true
- script is stopped and disabled if it misbehaves (too long runtime, error in code, low memory)

Script interface definition

Every script must include a *return* statement at the end, that defines its interface to the rest of OpenTX code. This statement defines:

- script *init* function (optional, see Init Function Syntax)
- script *run* function (see Run Function Syntax)

Example (interface only):

```
local function init_func()
end

local function run_func()
end

return { run=run_func, init=init_func }
```

Notes:

- local variables retain their values for as long as the model is loaded regardless of switch condition value

Advanced example (save as /SCRIPTS/FUNCTIONS/cntdwn.lua)

The script below is an example of customized countdown announcements. Note that the init function determines which version of OpenTX is running and sets the unit parameter for playNumber() accordingly.

```


local lstannounce
```

```
local target

local running = false

local duration = 120 -- two minute countdown
local announcements = { 120, 105, 90, 75, 60, 55, 50, 45, 40, 35, 30, 29, 28, 27, 26,
25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2
, 1, 0}
local annIndex

local minUnit

local function init()
  local version = getVersion()
  if version < "2.1" then
    minUnit = 16  -- unit for minutes in OpenTX 2.0
  elseif version < "2.2" then
    minUnit = 23  -- unit for minutes in OpenTX 2.1
  else
    minUnit = 25  -- unit for minutes in OpenTX 2.2
  end
end

local function run()

  local timenow = getTime() -- 10ms tick count
  local remaining
  local minutes
  local seconds

  if not running then
    running = true
    target = timenow + (duration * 100)
    annIndex = 1
  end

  remaining = math.floor(((target - timenow) / 100) + .7) --  +.7 adjust for announcem
ent lag

  if remaining < 0 then
    running = false -- we were 'paused' and missed zero
    return
  end

  while remaining < announcements[annIndex] do
    annIndex = annIndex + 1 -- catch up in case we were paused
  end

  if remaining == announcements[annIndex] then
    minutes = math.floor(remaining / 60)
    seconds = remaining % 60
    if minutes > 0 then
      playNumber(minutes, minUnit, 0)
```

```
    end
    if seconds > 0 then
      playNumber(seconds, 0, 0)
    end
    annIndex = annIndex + 1
  end

  if remaining <= 0 then
    playNumber(0,0,0)
    running = false
  end

end

return { init=init, run=run }
```

Widgets (HORUS ONLY) Scripts

General description

Widgets are small scripts that show some info in a 'zone' in one of the model specific user defined (telemetry) screens. You can define those screens within the telemetry menu on the HORUS.

Each model can have up to five custom screens, with up to 8 widgets per screen, depending on their size and layout. Each instance of a widget has his own custom settings.

File Location

Widgets are located on the SD card, each in their specific folder /WIDGETS/<*name*>/main.lua (*name* must be in 8 characters or less).

Lifetime of widgets

Widgets need to be registered through the telemetry setup menu.

- widget create function is called
- widget update function is called upon registration and at change of settings in the telemetry setup menu.
- widget background function is periodically called when custom telemetry screen is **not visible**. *Notice*:
  - This is different from the way telemetry scripts are handled
- widget refresh function is periodically called when custom telemetry screen is **visible**
- widget is stopped and disabled if it misbehaves (too long runtime, error in code, low memory)
- all widgets are stopped while one-time script is running (see Lua One-time scripts)

Once registered, widgets are started when the model is loaded.

Script interface definition

Every widget must include a return statement at the end, that defines its interface to the rest of OpenTX code. This statement defines:

- widget **name** (*name* must be a string of 10 characters or less)
- widget **options** array (maximum five options are allowed, 10 character names max, no

spaces!)
- widget **create** function
- widget **update** function
- script **background** function
- script **refresh** function

Example (draws a moving counter that counts only when not visible):

```
local defaultOptions = {
  { "ControlX", SOURCE, 1 },
  { "ScrollZ", BOOL, 1 }, -- BOOL is actually not a boolean, but toggles between 0 and
 1
  { "StepZ", VALUE, 1, 0, 10},
  { "COLOR", COLOR, RED },
}

local function createWidget(zone, options)
  lcd.setColor( CUSTOM_COLOR, options.COLOR )
  --  the CUSTOM_COLOR is foreseen to have one color that is not radio template relate
d, but it can be used by other widgets as well!
  local someVariable = 0
  local anotherVariable = {xWidget=0, yWidget = 0}
  return { zone=zone, options=options , someVariable = someVariable, anotherVariable=a
notherVariable }
end

local function updateWidget(widgetToUpdate, newOptions)
  widgetToUpdate.options = newOptions
  lcd.setColor( CUSTOM_COLOR, widgetToUpdate.options.COLOR )
  --  the CUSTOM_COLOR is foreseen to have one color that is not radio template relate
d, but it can be used by other widgets as well!
end

local function backgroundProcessWidget(widgetToProcessInBackground)
  local function process(...)
          return ... + 1
        end
  widgetToProcessInBackground.someVariable = process (widgetToProcessInBackground.some
Variable)
end

local function refreshWidget(widgetToRefresh)
  local counterLength = 50
  local counterHeight = 30

  --backgroundProcessWidget(widgetToRefresh)
  --background is not called automatically in display mode, so do it here if you need
it.

  local function anotherProcess(parameter,step,maxParameter)
          return (parameter + step) % maxParameter
```

```
        end

  widgetToRefresh.anotherVariable.xWidget
    = anotherProcess ( widgetToRefresh.anotherVariable.xWidget
      ,getValue(widgetToRefresh.options.ControlX)/10.24/20
      ,widgetToRefresh.zone.w-counterLength)

  widgetToRefresh.anotherVariable.yWidget
    = anotherProcess ( widgetToRefresh.anotherVariable.yWidget
      ,(widgetToRefresh.options.ScrollZ==1) and widgetToRefresh.options.StepZ or 0
      ,widgetToRefresh.zone.h-counterHeight)

  lcd.drawNumber(widgetToRefresh.anotherVariable.xWidget + widgetToRefresh.zone.x
    , widgetToRefresh.anotherVariable.yWidget + widgetToRefresh.zone.y
    , widgetToRefresh.someVariable
    , LEFT + DBLSIZE + CUSTOM_COLOR);
end

return { name="MovingCntr", options=defaultOptions, create=createWidget, update=update
Widget
  , refresh=refreshWidget, background=backgroundProcessWidget }
```

Notes:

- *options* are only passed through to OpenTX to be used on widget creation. Don't change them during operation, this has no effect.
- *create()* function is called once when widget is loaded and begins execution.
- *update()* function is called once when widget is loaded and begins execution.
- *background()* is called periodically when custom telemetry screen containing widget is not visible.
- *refresh()* function is called periodically when custom telemetry screen containing wodget is visible.

- in the example given, you can see that no global variables or functions are needed to operate the widget.

- variables that are used throughout the widget, can best be declared *inside* the create function as local variables
- those local variablkes can then be passed through to the other functions as an element of the widget array that is returned

Theme Scripts

Part II - OpenTX Lua API Programming Guide

This section provides more specifics on the OpenTX Lua implementation. Here you will find syntax rules for interface tables and functions. Also included is a table showing which of the available Lua libraries are accessible to OpenTx scripts.

Input Table Syntax

Overview

The input table defines what values are available as input(s) to mix scripts. There are two forms of input table entries.

- **SOURCE syntax**

  ```
  { "<name>", SOURCE }
  ```

  SOURCE inputs provide the current value of a selected OpenTX variable. The source must set by the user when the mix script is configured. Source can be any value OpenTX knows about (inputs, channels, telemetry values, switches, custom functions,...).
  Note: typical range is -1024 thru +1024. Simply divide the input value by 10.24 to interpret as a percentage from -100% to +100%.

- **VALUE syntax**

  ```
  { "<name>", VALUE, <min>, <max>, <default> }
  ```

  VALUE inputs provide a constant value that is set by the user when the mix script is configured.

  - *name* - maximum length of 8 characters
  - *min* - minimum value of -128
  - *max* - maximum value of 127
  - *default* - must be within the valid range specified

- **Maximum of 6 inputs per script (warning : was 8 in 2.1)**

Example using a SOURCE and a VALUE

```
local input =
    {
        { "Strength", SOURCE},                -- user selects source (typically slider
 or knob)
        { "Interval", VALUE, 0, 100, 0 }    -- interval value, default = 0.
    }

local function run(strength, interval)
    -- variable strength will contain the current slider value
    -- variable interval is set by the user and constant through script lifetime

    -- this script has no return value but may use playFile() to alert user

    return
end

return {input=input, run=run}
```

Output Table Syntax

Overview

Outputs are only used in mix scripts. The output table defines only name(s), the actual values are determined by the script's run function.

```
{ "<name1>", "<name2>" }
```

Example:

```
local output { "Val1", "Val2" }

local function run()
    return 0, -1024 -- these values will be available in OpenTX as Val1 and Val2
end

return {output=output, run=run}
```

Notes:

- Output name is limited to four characters.
- A maximum of 6 outputs are supported
- Number Format Outputs are 16 bit signed integers when they leave Lua script and are then divided by 10.24 to produce output value in percent:

Script Return Value	Mix Value in OpenTX
0	0%
996	97.2%
1024	100%
-1024	-100%

Init Function Syntax

If defined, *init* function is called right after the script is loaded from SD card and begins execution. Init is called only once before the run function is called for the first time.

```
local function <init_function_name>()
  -- code here runs only once when the model is loaded
end
```

- **Input Parameters:**

  none

- **Return values:**

  none

Run Function Syntax

The run function is the function that is periodically called for the lifetime of script execution. Syntax of the run function is different between mix scripts and telemetry scripts.

Run Function for Mix Scripts

```
local function <run_function_name>([first input, [second input], …])

   -- if mix has no return values
   return

   -- if mix has two return values
   return value1, value2

end
```

- **Input parameters:**

  zero or more input values, their names are arbitrary, their meaning and order is defined by the input table. (see Input Table Syntax)

- **Return values:**

  - none - if output table is empty (i.e. script has no output) values
    - or -
  - comma separated list of output values, their order and meaning is defined by the output table. (see Output Table Syntax)

Run Function for Telemetry Scripts

```
local function <run_function_name>(key-event)
  return 0 -- values other than zero will halt the script
end
```

- **Input parameters:**

  The *key-event* parameter indicates which transmitter button has been pressed (see Key Events)

- **Return values:**

  A non-zero return value will halt the script

Return Statement Syntax

The return statment is the last statement in an OpenTX Lua script. It defines the input/output table values and functions used to run the script.

Parameters *init*, *input* and *output* are optional. If a script doesn't use them, they can be omitted from return statement.

Example without *init* and *output*:

```
local inputs = { { "Aileron", SOURCE }, { "Ail. ratio", VALUE, -100, 100, 0 } }

local function run_func(ail, ratio)
    -- do some stuff
    if (ail > 50) and ( ratio < 40) then
        playFile("foo.wav")
    end
end

-- script that only uses input and run
return { run=run_func, input=inputs }
```

The following Lua libraries are available in OpenTx

Lua Standard Libraries	Included
package	-
coroutine	-
table	-
io	since OpenTX 2.1.0 (with limitations)
os	-
string	since OpenTX 2.1.7
bit32	since OpenTX 2.1.0
math	since OpenTX 2.0.0
debug	-

io library

The **io** library has been simplified and only a subset of functions and their functionality is available. What follows is a complete reference of io functions that are available to OpenTX scripts

Available functions:

- io.open()
- io.close()
- io.read()
- io.write()
- io.seek()

Examples

Read the whole file

```
-- this is an OpenTX stand-alone script

local function run(event)
  print("lua io.read test")         -- print() statements are visible in Debug output
window
  local f = io.open("foo.bar", "r")
  while true do
    local data = io.read(f, 10)     -- read up to 10 characters (newline char also cou
nts!)
    if #data == 0 then break end    -- we get zero length string back when we reach en
d of the file
    print("data: "..data)
    end
  io.close(f)
  return 1
end

return {  run=run }
```

Append data to file

```
-- this is an OpenTX stand-alone script

local function run(event)
  print("lua io.write test")
  local f = io.open("foo.bar", "a")        -- open file in append mode
  io.write(f, "first line\r\nsecond line\r\n")
  io.write(f, 4, "\r\n", 35.6778, "\r\n")  -- one can write multiple items at the same
 time
  local foo = -4.45
  io.write(f, foo, "\r\n")
  io.close(f)
  return 1    -- this will end the script execution
end

return { run=run }
```

io.open(<filename> [, <mode>])

The io.open() function is used to open the file on SD card for subsequent reading or writing. After the script is done with the file manipulation io.close() function should be used.

Notice: this functions is fully functional from OpenTX 2.1.5.

Parameters

- `filename` full path to the file starting from the SD card root directory. This function can't create a new file in non-existing directory.
- `mode` supported mode strings are:
  - `"r"` read access. File must exist beforehand. The read pointer is located at the beginning of file. This is the default mode if is omitted.
  - `"w"` write access. File is opened or created (if it didn't exist) and truncated (all existing file contents are lost).
  - "a" write access. File is opened or created (if it didn't exist) and write pointer is located at the end of the file. The existing file contents are preserved.

Return value

- `<file object>` if file was successfully opened.
- `nil` if file could not be opened.

io.close(<file object>)

The io.close() function is used to close open file.

Parameters

- `file object` a file object that was returned by the io.open() function.

Return value

- none

io.read(<file object> , <length>)

The io.read() function is used to read data from the file on SD card.

Notice: other read *commands* (like "*all", etc..) are *not supported*.

Parameters

- `file object` a file object that was returned by the io.open() function. The file must be opened in read mode.
- `length` number of characters/bytes to read. The number of actual read/returned characters can be less if the file end is reached.

Return value

- `<string>` a string with a length equal or less than
- `""` a zero length string if the end of file was reached

io.write(<file object> , <data>[, <data>, ...])

The io.write() function is used to write data to the file on SD card.

Parameters

- `file object` a file object that was returned by the io.open() function. The file must be opened in write or append mode.
- `data` any Lua type that can be converted into string. If more than one data parameter is used their contents are written to the file by one in the same order as they are specified.

Return value

- `<file object>` if data was successfully opened.
- `nil, <error string>, <error number>` if the data can't be written.

io.seek(<file object> , <offset>)

The io.seek() function is used to move the current read/write position.

Notice: other read standard seek bases (like "cur", "end") are **not supported**.

Parameters

- `file object` a file object that was returned by the io.open() function.
- `offset` position the read/write file pointer at the specified offset from the beginning of the file. If specified offset is bigger than the file size, then the pointer is moved to the end of the file.

Return value

- `0` success
- `<number>` any other value means failure.

Part III - OpenTX Lua API Reference

Constants

Key Events

This page describes the value that is passed to scripts in the `event` parameter. It is used in Telemetry and One-Time scripts.

The key event mechanism

Each time a key is pressed, held and released a number of events get generated. Here is a typical flow:

- when a key is pressed a `FIRST` event is generated
- if the key continues to be pressed, then after a while a `LONG` event is generated
- if the key continues to be pressed, then a `REPEAT` events are being generated
- when the key is released a `BREAK` event is generated

Couple of examples:

- a short press on key would generate: `FIRST`, `BREAK`
- a longer pres on key would generate: `FIRST`, `LONG`, `BREAK`
- even longer press: `FIRST`, `LONG`, `REPEAT`, REPEAT, ..., `BREAK`

This normal key event sequence can be altered with the killEvents(key) function. Any time this function is called (after the `FIRST` event) all further key events for this key will be suppressed until the next key press of this key. Examples:

- kill immediately after the key press would generate: `FIRST`
- kill after the long key press would generate: `FIRST`, `LONG`

Constants

The `event` parameter in the Telemetry and One-Time scripts run function actually carries two pieces of information:

- key number
- type of event

The two fields are combined into one single number. Some of these combinations are defined as constants and are available to Lua scripts:

Key Event Name	Comments
EVT_MENU_BREAK	**MENU** key release
EVT_PAGE_BREAK	**PAGE** key release
EVT_PAGE_LONG	**MENU** key long press
EVT_ENTER_BREAK	**ENT** key release
EVT_ENTER_LONG	**ENT** key long press
EVT_EXIT_BREAK	**EXIT** key release
EVT_PLUS_BREAK	**+** key release
EVT_MINUS_BREAK	**-** key release
EVT_PLUS_FIRST	**+** key press
EVT_MINUS_FIRST	**-** key press
EVT_PLUS_REPT	**+** key repeat
EVT_MINUS_REPT	**-** key repeat

Radios with rotary encoder (X7 and Horus) have also:

Key Event Name	Comments
EVT_ROT_BREAK	rotary encoder release
EVT_ROT_LONG	rotary encoder long press
EVT_ROT_LEFT	rotary encoder rotated left
EVT_ROT_RIGHT	rotary encoder rotated right

General Functions

GREY()

Returns gray value which can be used in LCD functions

@status current Introduced in 2.0.13

Parameters

none

Return value

- `(number)` a value that represents amount of *greyness* (from 0 to 15)

Notice

Only available on Taranis X9 series (212x64 displays)

crossfireTelemetryPop()

Pops a received Crossfire Telemetry packet from the queue.

@status current Introduced in 2.2.0

Parameters

none

Return value

- `nil` queue does not contain any (or enough) bytes to form a whole packet
- `multiple` returns 2 values:
  - command (number)
  - packet (table) data bytes

crossfireTelemetryPush()

This functions allows for sending telemetry data toward the TBS Crossfire link.

When called without parameters, it will only return the status of the output buffer without sending anything.

@status current Introduced in 2.2.0

Parameters

- `command` command
- `data` table of data bytes

Return value

- `boolean` data queued in output buffer or not.

defaultChannel(stick)

Get channel assigned to stick. See Default Channel Order in General Settings

@status current Introduced in 2.0.0

Parameters

- `stick` (number) stick number (from 0 to 3)

Return value

- `number` channel assigned to this stick (from 0 to 3)
- `nil` stick not found

defaultStick(channel)

Get stick that is assigned to a channel. See Default Channel Order in General Settings.

@status current Introduced in 2.0.0

Parameters

- `channel` (number) channel number (0 means CH1)

Return value

- `number` Stick assigned to this channel (from 0 to 3)

getDateTime()

Return current system date and time that is kept by the RTC unit

Parameters

none

Return value

- `table` current date and time, table elements:
  - `year` (number) year
  - `mon` (number) month
  - `day` (number) day of month
  - `hour` (number) hours
  - `min` (number) minutes
  - `sec` (number) seconds

Examples

general/getDateTime-example

```
local function run(e)
  local datenow = getDateTime()
  lcd.clear()
  lcd.drawText(1,1,"getDateTime() example",0)
  lcd.drawText(1,11,"year, mon, day: ", 0)
  lcd.drawText(lcd.getLastPos()+2,11,datenow.year..", "..datenow.mon..", "..datenow.da
y,0)
  lcd.drawText(1,21,"hour, min, sec: ", 0)
  lcd.drawText(lcd.getLastPos()+2,21,datenow.hour..", "..datenow.min..", "..datenow.se
c,0)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
getDateTime() example
year, mon, day: 2015, 10, 8
hour, min, sec: 19, 5, 21
ENT
TARANIS
2.4GHz
ACCST

getFieldInfo(name)

Return detailed information about field (source)

The list of valid sources is available:

OpenTX Version	Radio
2.0	all
2.1	X9D and X9D+, X9E
2.2	X9D and X9D+, X9E, Horus

@status current Introduced in 2.0.8, 'unit' field added in 2.2.0

Parameters

- `name` (string) name of the field

Return value

- `table` information about requested field, table elements:
  - `id` (number) field identifier
  - `name` (string) field name
  - `desc` (string) field description
  - 'unit' (number) unit identifier Full list
- `nil` the requested field was not found

Examples

general/getFieldInfo-example

```
local function run(e)
  local fieldinfo = getFieldInfo('rs')
  lcd.clear()
  lcd.drawText(1,1,"getFieldInfo() example",0)
  if fieldinfo then
    lcd.drawText(1,11,"id: ", 0)
    lcd.drawText(lcd.getLastPos()+2,11,fieldinfo['id'],0)
    lcd.drawText(1,21,"name: ", 0)
    lcd.drawText(lcd.getLastPos()+2,21,fieldinfo['name'],0)
    lcd.drawText(1,31,"desc: ", 0)
    lcd.drawText(lcd.getLastPos()+2,31,fieldinfo['desc'],0)
  else
    lcd.drawText(1,11,"Requested field not available!", 0)
  end
end

return{run=run}
```

getFlightMode(mode)

Return flight mode data.

@status current Introduced in 2.1.7

Parameters

- `mode` (number) flight mode number to return (0 - 8). If mode parameter is not specified (or contains invalid value), then the current flight mode data is returned.

Return value

- `multiple` returns 2 values:
  - (number) (current) flight mode number (0 - 8)
  - (string) (current) flight mode name

getGeneralSettings()

Returns (some of) the general radio settings

@status current Introduced in 2.0.6, `imperial` added in TODO, `language` and `voice` added in 2.2.0.

Parameters

none

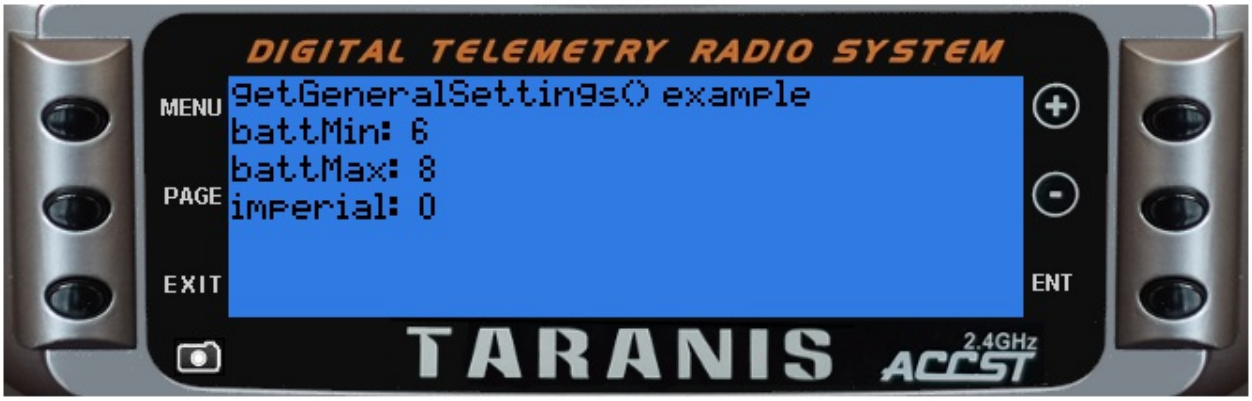
Return value

- `table` with elements:
  - `battMin` (number) radio battery range - minimum value
  - `battMax` (number) radio battery range - maximum value
  - `imperial` (number) set to a value different from 0 if the radio is set to the IMPERIAL units
  - `language` (string) radio language (used for menus)
  - `voice` (string) voice language (used for speech)

Examples

general/getGeneralSettings-example

```
local function run(e)
  local settings = getGeneralSettings()
  lcd.clear()
  lcd.drawText(1,1,"getGeneralSettings() example",0)
  lcd.drawText(1,11,"battMin: ", 0)
  lcd.drawText(lcd.getLastPos()+2,11,settings['battMin'],0)
  lcd.drawText(1,21,"battMax: ", 0)
  lcd.drawText(lcd.getLastPos()+2,21,settings['battMax'],0)
  lcd.drawText(1,31,"imperial: ", 0)
  lcd.drawText(lcd.getLastPos()+2,31,settings['imperial'],0)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
getGeneralSettings() example
battMin: 6
battMax: 8
imperial: 0
ENT
TARANIS
2.4GHz
ACCST

getRAS()

Return the RAS value or nil if no valid hardware found

@status current Introduced in 2.2.0

Parameters

none

Return value

- `number` representing RAS value. Value bellow 0x33 (51 decimal) are all ok, value above 0x33 indicate a hardware antenna issue. This is just a hardware pass/fail measure and does not represent the quality of the radio link

Notice

RAS was called SWR in the past

getRSSI()

Get RSSI value as well as low and critical RSSI alarm levels (in dB)

@status current Introduced in 2.2.0

Parameters

none

Return value

- `rssi` RSSI value (0 if no link)
- `alarm_low` Configured low RSSI alarm level
- `alarm_crit` Configured critical RSSI alarm level

getTime()

Return the time since the radio was started in multiple of 10ms

The timer internally uses a 32-bit counter which is enough for 30 years so overflows will not happen.

@status current Introduced in 2.0.0

Parameters

none

Return value

- `number` Number of 10ms ticks since the radio was started Example: run time: 12.54 seconds, return value: 1254

getValue(source)

Returns the value of a source.

The list of fixed sources:

OpenTX Version	Radio
2.0	all
2.1	X9D and X9D+, X9E
2.2	X9D and X9D+, X9E, Horus

In OpenTX 2.1.x the telemetry sources no longer have a predefined name. To get a telemetry value simply use it's sensor name. For example:

- Altitude sensor has a name "Alt"
- to get the current altitude use the source "Alt"
- to get the minimum altitude use the source "Alt-", to get the maximum use "Alt+"

@status current Introduced in 2.0.0, changed in 2.1.0, `Cels+` and `Cels-` added in 2.1.9

Parameters

- `source` can be an identifier (number) (which was obtained by the getFieldInfo()) or a name (string) of the source.

Return value

- `value` current source value (number). Zero is returned for:
  - non-existing sources
  - for all telemetry source when the telemetry stream is not received
- `table` GPS position is returned in a table:
  - `lat` (number) latitude, positive is North
  - `lon` (number) longitude, positive is East
  - `pilot-lat` (number) pilot latitude, positive is North
  - `pilot-lon` (number) pilot longitude, positive is East
- `table` GPS date/time, see getDateTime()
- `table` Cells are returned in a table (except where no cells were detected in which case the returned value is 0):

- table has one item for each detected cell:
- key (number) cell number (1 to number of cells)
- value (number) current cell voltage

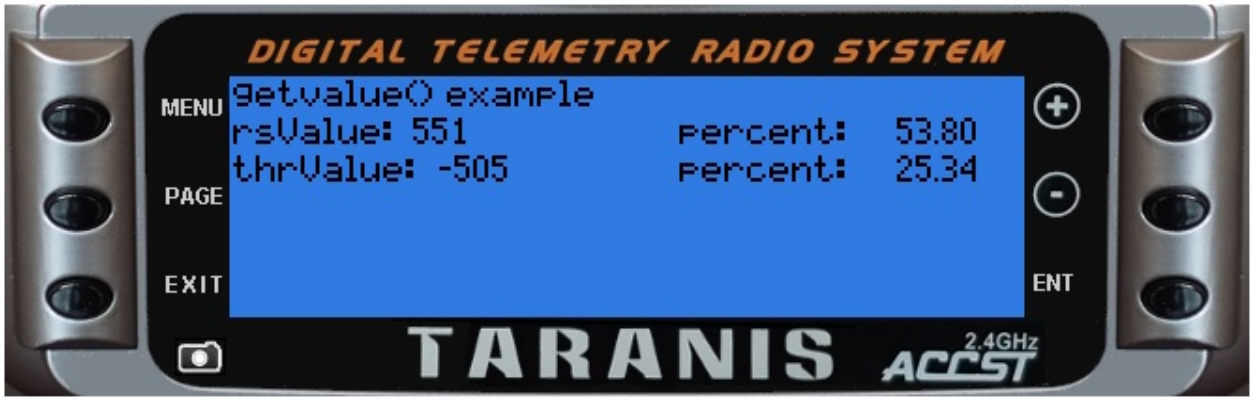
Notice

Getting a value by its numerical identifier is faster then by its name. While `Cels` sensor returns current values of all cells in a table, a `Cels+` or `Cels-` will return a single value - the maximum or minimum Cels value.

Examples

general/getValue-example

```
local function run(e)
  --
  -- NOTE: analog values (e.g. sticks and sliders) typically range from -1024 to +1024
  --       divide by 10.24 to scale into -100% thru +100%
  --       or add 1024 and divide by 20.48 to scale into 0% thru 100%
  --
  local rsValue = getValue('rs')
  local thrValue = getValue('thr')
  lcd.clear()
  lcd.drawText(1, 1, "getvalue() example",0)
  lcd.drawText(1, 11, "rsValue: ", 0)
  lcd.drawText(lcd.getLastPos() + 2, 11, rsValue, 0)
  lcd.drawText(120, 11, "percent: ", 0)
  lcd.drawNumber(lcd.getLastPos() + 32, 11, rsValue / 10.24, PREC2)
  lcd.drawText(1, 21, "thrValue: ", 0)
  lcd.drawText(lcd.getLastPos() + 2, 21, thrValue, 0)
  lcd.drawText(120, 21, "percent: ", 0)
  lcd.drawNumber(lcd.getLastPos() + 32, 21, (thrValue + 1024) / 20.48, PREC2)
end

return{run=run}
```



getVersion()

Return OpenTX version

@status current Introduced in 2.0.0, expanded in 2.1.7

Example

This example also runs in OpenTX versions where the function returned only one value:

```
local function run(event)
  local ver, radio, maj, minor, rev = getVersion()
  print("version: "..ver)
  if radio then print ("radio: "..radio) end
  if maj then print ("maj: "..maj) end
  if minor then print ("minor: "..minor) end
  if rev then print ("rev: "..rev) end
  return 1
end

return {  run=run }
```

Output of the above script in simulator:

```
version: 2.1.7
radio: taranis-simu
maj: 2
minor: 1
rev: 7
```

Parameters

none

Return value

- `string` OpenTX version (ie "2.1.5")
- `multiple` (available since 2.1.7) returns 5 values:
  - (string) OpenTX version (ie "2.1.5")
  - (string) radio version: `x9e` , `x9d+` or `x9d` . If running in simulator the "-simu" is added

- (number) major version (ie 2 if version 2.1.5)
- (number) minor version (ie 1 if version 2.1.5)
- (number) revision number (ie 5 if version 2.1.5)

killEvents(key)

Stops key state machine. See Key Events for the detailed description.

@status current Introduced in 2.0.0

Parameters

- `key` (number) key to be killed, can also include event type (only the key part is used)

Return value

none

loadScript(file [, mode], [,env])

Load a Lua script file. This is similar to Lua's own loadfile() API method, but it uses OpenTx's optional pre-compilation feature to save memory and time during load.

Return values are same as from Lua API loadfile() method: If the script was loaded w/out errors then the loaded script (or "chunk") is returned as a function. Otherwise, returns nil plus the error message.

@status current Introduced in 2.2.0

Example

```
fun, err = loadScript("/SCRIPTS/FUNCTIONS/print.lua")
if (fun ~= nil) then
   fun("Hello from loadScript()")
else
   print(err)
end
```

Parameters

- `file` (string) Full path and file name of script. The file extension is optional and ignored (see `mode` param to control which extension will be used). However, if an extension is specified, it should be ".lua" (or ".luac"), otherwise it is treated as part of the file name and the .lua/.luac will be appended to that.
- `mode` (string) (optional) Controls whether to force loading the text (.lua) or pre-compiled binary (.luac) version of the script. By default OTx will load the newest version and compile a new binary if necessary (overwriting any existing .luac version of the same script, and stripping some debug info like line numbers). You can use `mode` to control the loading behavior more specifically. Possible values are:
  - `b` only binary.
  - `t` only text.
  - `T` (default on simulator) prefer text but load binary if that is the only version available.
  - `bt` (default on radio) either binary or text, whichever is newer (binary preferred when timestamps are equal).
  - Add `x` to avoid automatic compilation of source file to .luac version. Eg: "tx", "bx", or "btx".

    - Add `c` to force compilation of source file to .luac version (even if existing version is newer than source file). Eg: "tc" or "btc" (forces "t", overrides "x").
    - Add `d` to keep extra debug info in the compiled binary. Eg: "td", "btd", or "tcd" (no effect with just "b" or with "x").
- `env` (integer) See documentation for Lua function loadfile().

Return value

- `function` The loaded script, or `nil` if there was an error (e.g. file not found or syntax error).
- `string` Error message(s), if any. Blank if no error occurred.

Notice

Note that you will get an error if you specify `mode` as "b" or "t" and that specific version of the file does not exist (eg. no .luac file when "b" is used). Also note that `mode` is NOT passed on to Lua's loader function, so unlike with loadfile() the actual file content is not checked (as if no mode or "bt" were passed to loadfile()).

playDuration(duration [, hourFormat])

Play a time value (text to speech)

@status current Introduced in 2.1.0

Parameters

- `duration` (number) number of seconds to play. Only integral part is used.
- `hourFormat` (number):
  - `0 or not present` play format: minutes and seconds.
  - `!= 0` play format: hours, minutes and seconds.

Return value

none

Examples

The one time script below will announce "zero hours 1 minute and 1 second"

```
local function run()
  playDuration(61, 1) -- announce "zero hours 1 minute and 1 second
  return 1
end

return { run=run }
```

playFile(name)

Play a file from the SD card

@status current Introduced in 2.0.0, changed in 2.1.0

Parameters

- `path` (string) full path to wav file (i.e. “/SOUNDS/en/system/tada.wav”) Introduced in 2.1.0: If you use a relative path, the current language is appended to the path (example: for English language: `/SOUNDS/en` is appended)

Return value

none

Examples

Example telemetry script

```
local eleid

local function init()
  local fieldinfo = getFieldInfo('ele')
  if fieldinfo then
    eleid = fieldinfo.id
  else
    eleid = -1
  end
end

local function run(e)
  lcd.clear()
  lcd.drawText(1,1,"playFile() example",0)
  local eleVal = getValue(eleid)
  if eleVal > 900 then
    lcd.drawText(1,11,"Whoa - easy there cowboy", 0)
    playFile("horn.wav")
  else
    lcd.drawText(1,11,"eleVal: " .. eleVal, 0)
  end
end

return {init=init, run=run}
```

playHaptic(duration, pause [, flags])

Generate haptic feedback

@status current Introduced in 2.2.0

Parameters

- `duration` (number) length of the haptic feedback in milliseconds
- `pause` (number) length of the silence after haptic feedback in milliseconds
- `flags` (number):
  - `0 or not present` play with normal priority
  - `PLAY_NOW` play immediately

Return value

none

playNumber(value, unit [, attributes])

Play a numerical value (text to speech)

@status current Introduced in 2.0.0

Parameters

- `value` (number) number to play. Value is interpreted as integer.
- `unit` (number) unit identifier Full list
- `attributes` (unsigned number) possible values:
  - `0 or not present` plays integral part of the number (for a number 123 it plays 123)
  - `PREC1` plays a number with one decimal place (for a number 123 it plays 12.3)
  - `PREC2` plays a number with two decimal places (for a number 123 it plays 1.23)

Return value

none

Examples

Example mix script

```
local nbr = 0
local unit = 0
local prec = 0
local lastnbr = 0
local lastunit = 0
local lastprec = 0
local lasttime = 0

local input =
    {
        { "innbr", SOURCE},
        { "inprec", SOURCE},
        { "toggle", SOURCE}
    }

local output = {"nbr", "prec", "unit"}

local function run(innbr, inprec, toggle)
```

```
  local change = false
  local advance = false
  local timenow = getTime()

  nbr = innbr -- will range from - 1024 thru + 1024
  prec = math.floor((inprec + 1024) * (2 / 2014)) -- force range to 0 thru 2

  if (toggle > 0) then
    change = true
    advance = true
  end

  if math.abs(lastnbr - nbr) > 10 then
    change = true
  end

  if not (lastprec == prec) then
    change = true
  end

  if change and ((timenow - lasttime) > 200) then
    lasttime = timenow
    lastnbr = nbr
    if advance then
      lastunit = (lastunit + 1) % 31
    end
    lastprec = prec
    if (lastprec == 0) then
      playNumber(lastnbr, lastunit, 0)
    elseif (lastprec == 1) then
      playNumber(lastnbr, lastunit, PREC1)
    else
      playNumber(lastnbr, lastunit, PREC2)
    end
  end
  return lastnbr * 10.24, lastprec * 10.24, lastunit * 10.24
end

return {run=run, input=input, output=output}
```

playTone(frequency, duration, pause [, flags [, freqIncr]])

Play a tone

@status current Introduced in 2.1.0

Parameters

- `frequency` (number) tone frequency in Hz (from 150 to 15000)
- `duration` (number) length of the tone in milliseconds
- `pause` (number) length of the silence after the tone in milliseconds
- `flags` (number):
  - `0 or not present` play with normal priority.
  - `PLAY_BACKGROUND` play in background (built in vario function uses this context)
  - `PLAY_NOW` play immediately
- `freqIncr` (number) positive number increases the tone pitch (frequency with time), negative number decreases it. The frequency changes every 10 milliseconds, the change is `freqIncr * 10Hz`. The valid range is from -127 to 127.

Return value

none

popupConfirmation(title, event)

Raises a pop-up on screen that asks for confirmation

@status current Introduced in 2.2.0

Parameters

- `title` (string) text to display
- `event` (number) the event variable that is passed in from the Run function (key pressed)

Return value

- `"CANCEL"` user pushed EXIT key

Notice

Use only from stand-alone and telemetry scripts.

popupInput(title, event, input, min, max)

Raises a pop-up on screen that allows uses input

@status current Introduced in 2.0.0

Parameters

- `title` (string) text to display
- `event` (number) the event variable that is passed in from the Run function (key pressed)
- `input` (number) value that can be adjusted by the +/- keys
- `min` (number) min value that input can reach (by pressing the - key)
- `max` (number) max value that input can reach

Return value

- `number` result of the input adjustment
- `"OK"` user pushed ENT key
- `"CANCEL"` user pushed EXIT key

Notice

Use only from stand-alone and telemetry scripts.

popupWarning(title, event)

Raises a pop-up on screen that shows a warning

@status current Introduced in 2.2.0

Parameters

- `title` (string) text to display
- `event` (number) the event variable that is passed in from the Run function (key pressed)

Return value

- `"CANCEL"` user pushed EXIT key

Notice

Use only from stand-alone and telemetry scripts.

setTelemetryValue(id, subID, instance, value [, unit [, precision [, name]]])

@status current Introduced in 2.2.0

Parameters

- `id` Id of the sensor, valid range is from 0 to 0xFFFF
- `subID` subID of the sensor, usually 0, valid range is from 0 to 7
- `instance` instance of the sensor (SensorID), valid range is from 0 to 0xFF
- `value` fed to the sensor
- `unit` unit of the sensor Full list
- `precision` the precision of the sensor
  - `0 or not present` no decimal precision.
  - `!= 0` value is divided by 10^precision, e.g. value=1000, prec=2 => 10.00.
- `name` (string) Name of the sensor if it does not yet exist (4 chars).
  - `not present` Name defaults to the Id.
  - `present` Sensor takes name of the argument. Argument must have name surrounded by quotes: e.g., "Name"

Return value

- `true,` if the sensor was just added. In this case the value is ignored (subsequent call will set the value)

Notice

All three parameters `id`, `subID` and `instance` can't be zero at the same time. At least one of them must be different from zero.

sportTelemetryPop()

Pops a received SPORT packet from the queue. Please note that only packets using a data ID within 0x5000 to 0x52FF (frame ID == 0x10), as well as packets with a frame ID equal 0x32 (regardless of the data ID) will be passed to the LUA telemetry receive queue.

@status current Introduced in 2.2.0

Parameters

none

Return value

- `nil` queue does not contain any (or enough) bytes to form a whole packet
- `multiple` returns 4 values:
  - sensor ID (number)
  - frame ID (number)
  - data ID (number)
  - value (number)

sportTelemetryPush()

This functions allows for sending SPORT telemetry data toward the receiver, and more generally, to anything connected SPORT bus on the receiver or transmitter.

When called without parameters, it will only return the status of the output buffer without sending anything.

@status current Introduced in 2.2.0

Parameters

- `sensorId` physical sensor ID
- `frameId` frame ID
- `dataId` data ID
- `value` value

Return value

- `boolean` data queued in output buffer or not.

Model Functions

model.defaultInputs()

Set all inputs to defaults

@status current Introduced in 2.0.0

Parameters

none

Return value

none

model.deleteInput(input, line)

Delete line from specified input

@status current Introduced in 2.0.0

Parameters

- `input` (unsigned number) input number (use 0 for Input1)
- `line` (unsigned number) input line (use 0 for first line)

Return value

none

model.deleteInputs()

Delete all Inputs

@status current Introduced in 2.0.0

Parameters

none

Return value

none

model.deleteMix(channel, line)

Delete mixer line from specified Channel

@status current Introduced in 2.0.0

Parameters

- `channel` (unsigned number) channel number (use 0 for CH1)
- `line` (unsigned number) mix number (use 0 for first line(mix))

Return value

none

model.deleteMixes()

Remove all mixers

@status current Introduced in 2.0.0

Parameters

none

Return value

none

model.getCurve(curve)

Get Curve parameters

Note that functions returns the tables starting with index 0 contrary to LUA's usual index starting with 1

@status current Introduced in 2.0.12

Parameters

- `curve` (unsigned number) curve number (use 0 for Curve1)

Return value

- `nil` requested curve does not exist
- `table` curve data:
  - `name` (string) name
  - `type` (number) type
  - `smooth` (boolean) smooth
  - `points` (number) number of points
  - `y` (table) table of Y values:
    - `key` is point number (zero based)
    - `value` is y value
  - `x` (table) **only included for custom curve type**:
    - `key` is point number (zero based)
    - `value` is x value

model.getCustomFunction(function)

Get Custom Function parameters

@status current Introduced in 2.0.0, TODO rename function

Parameters

- `function` (unsigned number) custom function number (use 0 for CF1)

Return value

- `nil` requested custom function does not exist
- `table` custom function data:
  - `switch` (number) switch index
  - `func` (number) function index
  - `name` (string) Name of track to play (only returned only returned if action is play track, sound or script)
  - `value` (number) value (only returned only returned if action is **not** play track, sound or script)
  - `mode` (number) mode (only returned only returned if action is **not** play track, sound or script)
  - `param` (number) parameter (only returned only returned if action is **not** play track, sound or script)
  - `active` (number) 0 = disabled, 1 = enabled

model.getGlobalVariable(index [, flight_mode])

Return current global variable value

Example:

```
-- get GV3 (index = 2) from Flight mode 0 (FM0)
val = model.getGlobalVariable(2, 0)
```

Parameters

- `index` zero based global variable index, use 0 for GV1, 8 for GV9
- `flight_mode` Flight mode number (0 = FM0, 8 = FM8)

Return value

- `nil` requested global variable does not exist
- `number` current value of global variable

Notice

a simple warning or notice

model.getInfo()

Get current Model information

@status current Introduced in 2.0.6, changed in 2.2.0

Parameters

none

Return value

- `table` model information:
  - `name` (string) model name
  - `bitmap` (string) bitmap name (not present on X7)

model.getInput(input, line)

Return input data for given input and line number

@status current Introduced in 2.0.0, `switch` added in TODO

Parameters

- `input` (unsigned number) input number (use 0 for Input1)
- `line` (unsigned number) input line (use 0 for first line)

Return value

- `nil` requested input or line does not exist
- `table` input data:
  - `name` (string) input line name
  - `source` (number) input source index
  - `weight` (number) input weight
  - `offset` (number) input offset
  - `switch` (number) input switch index

model.getInputsCount(input)

Return number of lines for given input

@status current Introduced in 2.0.0

Parameters

- `input` (unsigned number) input number (use 0 for Input1)

Return value

- `number` number of configured lines for given input

model.getLogicalSwitch(switch)

Get Logical Switch parameters

@status current Introduced in 2.0.0

Parameters

- `switch` (unsigned number) logical switch number (use 0 for LS1)

Return value

- `nil` requested logical switch does not exist
- `table` logical switch data:
  - `func` (number) function index
  - `v1` (number) V1 value (index)
  - `v2` (number) V2 value (index or value)
  - `v3` (number) V3 value (index or value)
  - `and` (number) AND switch index
  - `delay` (number) delay (time in 1/10 s)
  - `duration` (number) duration (time in 1/10 s)

model.getMix(channel, line)

Get configuration for specified Mix

@status current Introduced in 2.0.0, parameters below `multiplex` added in 2.0.13

Parameters

- `channel` (unsigned number) channel number (use 0 for CH1)
- `line` (unsigned number) mix number (use 0 for first line(mix))

Return value

- `nil` requested channel or line does not exist
- `table` mix data:
  - `name` (string) mix line name
  - `source` (number) source index
  - `weight` (number) weight (1024 == 100%) value or GVAR1..9 = 4096..4011, -GVAR1..9 = 4095..4087
  - `offset` (number) offset value or GVAR1..9 = 4096..4011, -GVAR1..9 = 4095..4087
  - `switch` (number) switch index
  - `multiplex` (number) multiplex (0 = ADD, 1 = MULTIPLY, 2 = REPLACE)
  - `curveType` (number) curve type (function, expo, custom curve)
  - `curveValue` (number) curve index
  - `flightModes` (number) bit-mask of active flight modes
  - `carryTrim` (boolean) carry trim
  - `mixWarn` (number) warning (0 = off, 1 = 1 beep, .. 3 = 3 beeps)
  - `delayUp` (number) delay up (time in 1/10 s)
  - `delayDown` (number) delay down
  - `speedUp` (number) speed up
  - `speedDown` (number) speed down

model.getMixesCount(channel)

Get the number of Mixer lines that the specified Channel has

@status current Introduced in 2.0.0

Parameters

- `channel` (unsigned number) channel number (use 0 for CH1)

Return value

- `number` number of mixes for requested channel

model.getModule(index)

Get RF module parameters

`rfProtocol` values:

- -1 OFF
- 0 D16
- 1 D8
- 2 LR12

@status current Introduced in TODO

Parameters

- `index` (number) module index (0 for internal, 1 for external)

Return value

- `nil` requested module does not exist
- `table` module parameters:
  - `rfProtocol` (number) protocol index
  - `modelId` (number) receiver number
  - `firstChannel` (number) start channel (0 is CH1)
  - `channelsCount` (number) number of channels sent to module

model.getOutput(index)

Get servo parameters

@status current Introduced in 2.0.0

Parameters

- `index` (unsigned number) output number (use 0 for CH1)

Return value

- `nil` requested output does not exist
- `table` output parameters:
  - `name` (string) name
  - `min` (number) Minimum % * 10
  - `max` (number) Maximum % * 10
  - `offset` (number) Subtrim * 10
  - `ppmCenter` (number) offset from PPM Center. 0 = 1500
  - `symetrical` (number) linear Subtrim 0 = Off, 1 = On
  - `revert` (number) irection 0 = ---, 1 = INV
  - `curve`
    - (number) Curve number (0 for Curve1)
    - or `nil` if no curve set

model.getTimer(timer)

Get model timer parameters

@status current Introduced in 2.0.0

Parameters

- `timer` (number) timer index (0 for Timer 1)

Return value

- `nil` requested timer does not exist
- `table` timer parameters:
  - `mode` (number) timer trigger source: off, abs, stk, stk%, sw/!sw, !m_sw/!m_sw
  - `start` (number) start value [seconds], 0 for up timer, 0> down timer
  - `value` (number) current value [seconds]
  - `countdownBeep` (number) countdown beep (0 = silent, 1 = beeps, 2 = voice)
  - `minuteBeep` (boolean) minute beep
  - `persistent` (number) persistent timer

model.insertInput(input, line, value)

Insert an Input at specified line

@status current Introduced in 2.0.0, `switch` added in TODO

Parameters

- `input` (unsigned number) input number (use 0 for Input1)
- `line` (unsigned number) input line (use 0 for first line)
- `value` (table) input data, see model.getInput()

Return value

none

model.insertMix(channel, line, value)

Insert a mixer line into Channel

@status current Introduced in 2.0.0, parameters below `multiplex` added in 2.0.13

Parameters

- `channel` (unsigned number) channel number (use 0 for CH1)
- `line` (unsigned number) mix number (use 0 for first line(mix))
- `value` (table) see model.getMix() for table format

Return value

none

model.resetTimer(timer)

Reset model timer to a startup value

@status current Introduced in TODO

Parameters

- `timer` (number) timer index (0 for Timer 1)

Return value

none

model.setCurve(curve, params)

Set Curve parameters

The first and last x value must 0 and 100 and x values must be monotonically increasing

@status current Introduced in 2.2.0

Example setting a 4-point custom curve:

```
params = {}
params["x"] =  {0, 34, 77, 100}
params["y"] = {-70, 20, -89, -100}
params["smooth"] = 1
params["type"] = 1
val =  model.setCurve(2, params)
```

setting a 6-point standard smoothed curve

```
val = model.setCurve(3, {smooth=1, y={-100, -50, 0, 50, 100, 80}})
```

Parameters

- `curve` (unsigned number) curve number (use 0 for Curve1)
- `params` see model.getCurve return format for table format. setCurve uses standard lua array indexing and array start at index 1

Return value

- `` 0 - Everything okay

```
1 - Wrong number of points
2 - Invalid Curve number
3 - Cuve does not fit anymore
4 - point of out of index
5 - x value not monotonically increasing
6 - y value not in range [-100;100]
7 - extra values for y are set
8 - extra values for x are set
```

model.setCustomFunction(function, value)

Set Custom Function parameters

@status current Introduced in 2.0.0, TODO rename function

Parameters

- `function` (unsigned number) custom function number (use 0 for CF1)
- `value` (table) custom function parameters, see model.getCustomFunction() for table format

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

model.setGlobalVariable(index, flight_mode, value)

Sets current global variable value. See also model.getGlobalVariable()

Parameters

- `index` zero based global variable index, use 0 for GV1, 8 for GV9
- `flight_mode` Flight mode number (0 = FM0, 8 = FM8)
- `value` new value for global variable. Permitted range is from -1024 to 1024.

Return value

none

Notice

Global variable can only store integer values, any floating point value is converted into integer value by truncating everything behind a floating point.

Examples

Example

this is a sample example

model/setGlobalVariable-example

```
function foo(bar)
  local x = bar * 2
end
```

model.setInfo(value)

Set the current Model information

@status current Introduced in 2.0.6, changed in TODO

Parameters

- `value` model information data, see model.getInfo()

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

model.setLogicalSwitch(switch, value)

Set Logical Switch parameters

@status current Introduced in 2.0.0

Parameters

- `switch` (unsigned number) logical switch number (use 0 for LS1)
- `value` (table) see model.getLogicalSwitch() for table format

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

To set the `and` member (which is Lua keyword) use the following syntax:
`model.setLogicalSwitch(30, {func=4,v1=1,v2=-99, ["and"]=24})`

model.setModule(index, value)

Set RF module parameters

@status current Introduced in TODO

Parameters

- `index` (number) module index (0 for internal, 1 for external)
- `value` module parameters, see model.getModule()

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

model.setOutput(index, value)

Set servo parameters

@status current Introduced in 2.0.0

Parameters

- `index` (unsigned number) channel number (use 0 for CH1)
- `value` (table) servo parameters, see model.getOutput() for table format

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

model.setTimer(timer, value)

Set model timer parameters

@status current Introduced in 2.0.0

Parameters

- `timer` (number) timer index (0 for Timer 1)
- `value` timer parameters, see model.getTimer()

Return value

none

Notice

If a parameter is missing from the value, then that parameter remains unchanged.

Lcd Functions

Lcd Functions Overview

Description

Lcd functions allow scripts to interact with the transmitter display. This access is limited to the 'run' functions of One-Time and Telemetry scripts. Widget scripts on the Horus (X10 and X12S) can make use of the lcd functions as well.

Notes:

The run function is periodically called when the screen is visible. In OpenTX 2.0 each invocation starts with a blank screen (unless lcd.lock() is used). Under OpenTX 2.1 screen state is always persisted across calls to the run function. **Many scripts originally written for OpenTX 2.0 require a call to lcd.clear() at the beginning of their run function in order to display properly under 2.1 and 2.2.**

Many of the lcd functions accept parameters named *flags* and *patterns*. The names and their meanings are described below.

Flags Constants

Name	Description	Version	Notes
0	normal font, default precision for numeric		
DBLSIZE	double size font		
MIDSIZE	mid sized font		
SMLSIZE	small font		
INVERS	inverted display		
BLINK	blinking text		
XXLSIZE	jumbo font	2.0.6	
LEFT	left justify	2.0.6	Default for most functions not related to bitmaps
RIGHT	right justify		
PREC1	single decimal place		
PREC2	two decimal places		
GREY_DEFAULT	grey fill		
TIMEHOUR	dislay hours		Only for drawTimer

Patterns Constants

Name	Description
FORCE	pixels will be black
ERASE	pixels will be white
DOTTED	lines will appear dotted

Screen Constants

Name	Description
LCD_W	width in pixels
LCD_H	height in pixels

Screen Information

Radio	LCD_W	LCD_H	Colours
X7	128	64	1 bit
X9D	212	64	4 bit
X9D+	212	64	4 bit
X9E	212	64	4 bit
X10	480	272	RGB565
X12S	480	272	RGB565

lcd.RGB(r, g, b)

Returns a 5/6/5 rgb color code, that can be used with lcd.setColor

@status current Introduced in 2.2.0

Parameters

- `r` (integer) a number between 0x00 and 0xff that expresses te amount of red in the color
- `g` (integer) a number between 0x00 and 0xff that expresses te amount of green in the color
- `b` (integer) a number between 0x00 and 0xff that expresses te amount of blue in the color

Return value

- `number` (integer) rgb color expressed in 5/6/5 format

Notice

Only available on Horus

lcd.clear([color])

Clear the LCD screen

@status current Introduced in 2.0.0, `color` parameter introduced in 2.2.0 RC12

Parameters

- `color` (optional, only on color screens)

Return value

none

Notice

This function only works in stand-alone and telemetry scripts.

Examples

lcd/clear-example

```
--
--
-- This example demonstrates the lcd.clear() function
--
-- NOTE: Compare the output of the images below
--       lcd.clear() is NOT CALLED automatically in OpenTX 2.1
--

local startTicks
local nowTicks

local function init()
  startTicks = getTime() / 100.0
end

local function background()
  nowTicks = getTime() / 100.0
end

local function run(e)
  background()
  local interval = 10 - math.floor(nowTicks % 10)
  lcd.drawText(1, 1, "clear() example",0)
  lcd.drawText((10 * interval) + 1, 10 , interval, 0)
  if interval == 10 then
    lcd.clear()
  end
end

return{run=run, background=background}
```

clear-example.lua running under OpenTX 2.1

clear-example.lua running under OpenTX 2.0


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
clear() example
1
ENT
TARANIS
2.4GHz
ACCST

lcd.drawBitmap(bitmap, x, y [, scale])

Displays a bitmap at (x,y)

@status current Introduced in 2.2.0

Parameters

- `bitmap` (pointer) point to a bitmap previously opened with Bitmap.open()
- `x,y` (positive numbers) starting coordinates
- `scale` (positive numbers) scale in %, 50 divides size by two, 100 is unchanged, 200 doubles size. Omitting scale draws image in 1:1 scale and is faster than specifying 100 for scale.

Return value

none

Notice

Only available on Horus

lcd.drawChannel(x, y, source, flags)

Display a telemetry value at (x,y)

@status current Introduced in 2.0.6, changed in 2.1.0 (only telemetry sources are valid)

Parameters

- `x,y` (positive numbers) starting coordinate
- `source` can be a source identifier (number) or a source name (string). See getValue()
- `flags` (unsigned number) drawing flags

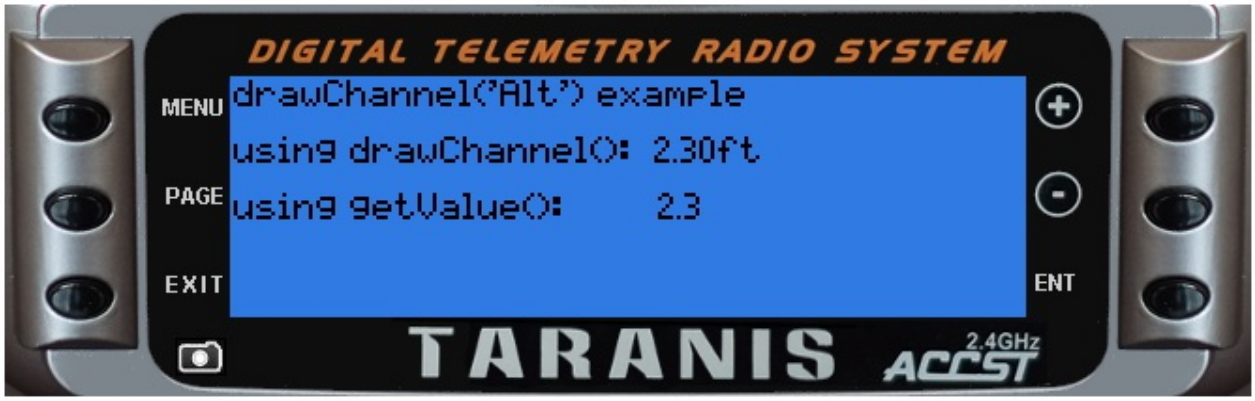
Return value

none

Examples

lcd/drawChannel-example

```
local function run(e)
  lcd.clear()
  lcd.drawText(1, 1, "drawChannel('Alt') example",0)
  lcd.drawText(1, 16, "using drawChannel(): ", 0)
  lcd.drawChannel(lcd.getLastPos()+20, 16 , "Alt", 0)
  lcd.drawText(1, 31, "using getValue(): ", 0)
  lcd.drawText(lcd.getLastPos() + 22, 31, getValue("Alt"), 0)
end

return{run=run}
```



lcd.drawCombobox(x, y, w, list, idx [, flags])

Draw a combo box

@status current Introduced in 2.0.0

Parameters

- `x,y` (positive numbers) top left corner position
- `w` (number) width of combo box in pixels
- `list` (table) combo box elements, each element is a string
- `idx` (integer) index of entry to highlight
- `flags` (unsigned number) drawing flags, the flags can not be combined:
  - `BLINK` combo box is expanded
  - `INVERS` combo box collapsed, text inversed
  - `0 or not present` combo box collapsed, text normal

Return value

none

Notice

Only available on Taranis

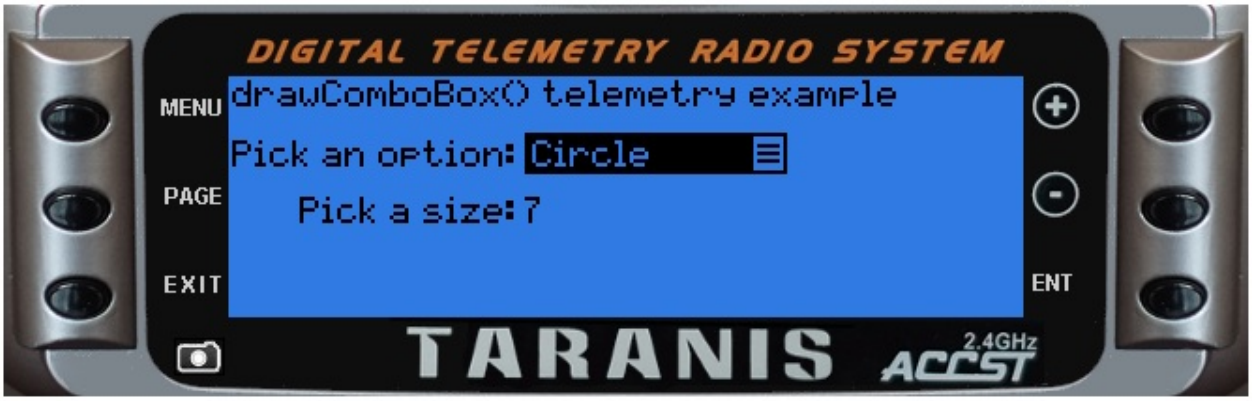
Examples

lcd/drawCombobox-example

```
local comboOptions
local selectedOption
local selectedSize
local editMode
local activeField
local fieldMax

local function valueIncDec(event,value,min,max,step)
    if editMode then
      if event==EVT_PLUS_FIRST or event==EVT_PLUS_REPT then
```

```
        if value<=max-step then
          value=value+step
        end
      elseif event==EVT_MINUS_FIRST or event==EVT_MINUS_REPT then
        if value>=min+step then
          value=value-step
        end
      end
    end
    return value
  end

local function fieldIncDec(event,value,max,force)
    if editMode or force==true then
      if event==EVT_PLUS_FIRST then
        value=value+max
      elseif event==EVT_MINUS_FIRST then
        value=value+max+2
      end
      value=value%(max+1)
    end
    return value
  end

  local function getFieldFlags(p)
    local flg = 0
    if activeField==p then
      flg=INVERS
      if editMode then
        flg=INVERS+BLINK
      end
    end
    return flg
  end

local function init()
  fieldMax = 1
  comboOptions = {"Triangle","Circle","Square"}
  selectedOption = 0
  activeField = 0
  selectedSize = 0
end

local function run(event)
  lcd.clear()
  -- draw from the bottom up so we don't overwrite the combo box if open
  lcd.drawText(19, 32, "Pick a size:", 0)
  lcd.drawText(lcd.getLastPos() + 2, 32, selectedSize, getFieldFlags(1))
  lcd.drawText(1, 1, "drawComboBox() telemetry example",0)
  lcd.drawText(1, 17, "Pick an option:", 0)
  lcd.drawCombobox(lcd.getLastPos() + 2, 15, 70, comboOptions, selectedOption, getFiel
dFlags(0))
```

```
  if event == EVT_ENTER_BREAK then
    editMode = not editMode
  end
  if editMode then
    if activeField == 0 then
      selectedOption = fieldIncDec(event, selectedOption, 2)
    elseif activeField == 1 then
      selectedSize = valueIncDec(event, selectedSize, 0, 10, 1)
    end
  else
    activeField = fieldIncDec(event, activeField, fieldMax, true)
  end
end

return{run=run, init=init}
```



lcd.drawFilledRectangle(x, y, w, h [, flags])

Draw a solid rectangle from top left corner (x,y) of specified width and height

@status current Introduced in 2.0.0

Parameters

- `x,y` (positive numbers) top left corner position
- `w` (number) width in pixels
- `h` (number) height in pixels
- `flags` (unsigned number) drawing flags

Return value

none

Examples

lcd/drawFilledRectangle-example

```
local function run()
  lcd.clear()
  lcd.drawText(10,22,"drawFilledRectangle()",DBLSIZE)
  lcd.drawFilledRectangle(5, 5, 103, 50, GREY_DEFAULT)
  lcd.drawFilledRectangle(152, 33, 50, 25, SOLID)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
ENT
drawFilledRectangle
TARANIS
2.4GHz
ACCST

lcd.drawGauge(x, y, w, h, fill, maxfill [, flags])

Draw a simple gauge that is filled based upon fill value

@status current Introduced in 2.0.0, changed in 2.2.0

Parameters

- `x,y` (positive numbers) top left corner position
- `w` (number) width in pixels
- `h` (number) height in pixels
- `fill` (number) amount of fill to apply
- `maxfill` (number) total value of fill
- `flags` (unsigned number) drawing flags

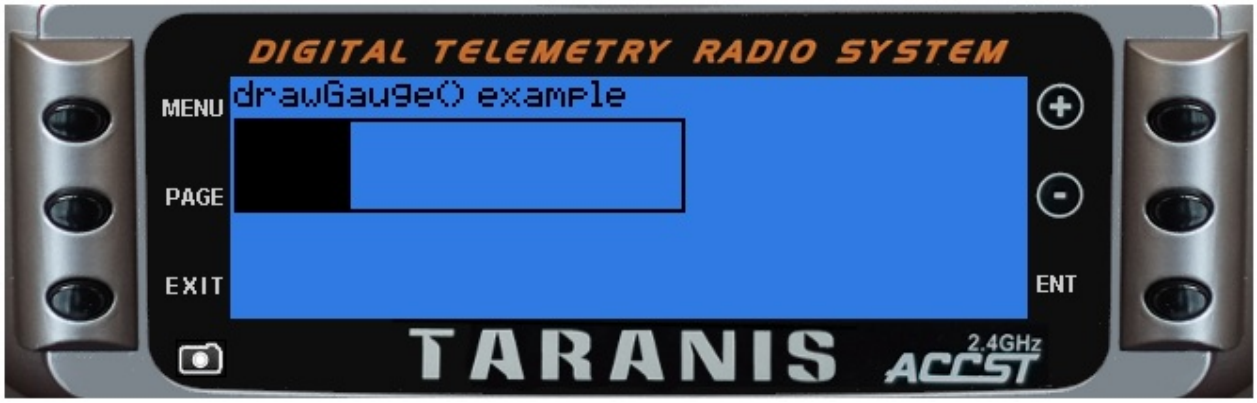
Return value

none

Examples

lcd/drawGauge-example

```
local function run(event)
  lcd.clear()
  lcd.drawText(1,1,"drawGauge() example", 0)
  lcd.drawGauge(1, 11, 120, 25, 250, 1000)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
drawGauge() example
PAGE
EXIT
ENT
TARANIS
2.4GHz
ACCST

lcd.drawLine(x1, y1, x2, y2, pattern, flags)

Draw a straight line on LCD

@status current Introduced in 2.0.0

Parameters

- `x1,y1` (positive numbers) starting coordinate
- `x2,y2` (positive numbers) end coordinate
- `pattern` TODO
- `flags` TODO

Return value

none

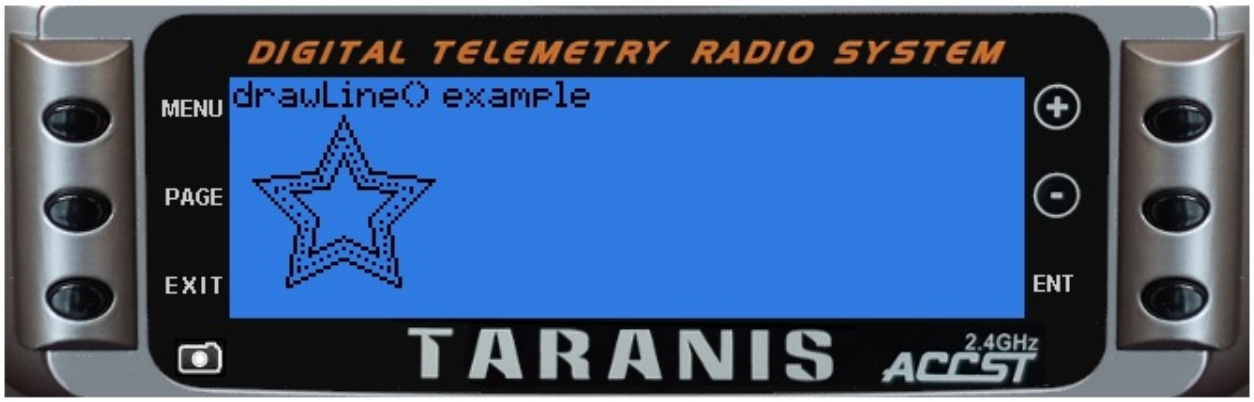
Notice

If the start or the end of the line is outside the LCD dimensions, then the whole line will not be drawn (starting from OpenTX 2.1.5)

Examples

lcd/drawLine-example

```
local alpha = (2 * math.pi) / 10

local function getPoint(centerX, centerY, radius, point)
  local omega = alpha * point
  local r = radius*(point % 2 + 1)/2
  local X = (r * math.sin(omega)) + centerX
  local Y = (r * math.cos(omega)) + centerY
  return X, Y
end

local function drawStar(centerX, centerY, radius, pattern, flags)
  local point = 10
  local startX, startY = getPoint(centerX, centerY, radius, point)
  for point = 1, 10 do
    local nextX, nextY = getPoint(centerX, centerY, radius, point)
    lcd.drawLine(startX, startY, nextX, nextY, pattern, flags)
    startX = nextX
    startY = nextY
  end
end

local function run(event)
  lcd.clear()
  lcd.drawText(1,1,"drawLine() example", 0)
  drawStar(30, 35, 25, SOLID, FORCE)
  drawStar(30, 35, 20, DOTTED, FORCE)
  drawStar(30, 35, 15, SOLID, FORCE)
end

return{run=run}
```



lcd.drawNumber(x, y, value [, flags])

Display a number at (x,y)

@status current Introduced in 2.0.0, `SHADOWED` introduced in 2.2.1

Parameters

- `x,y` (positive numbers) starting coordinate
- `value` (number) value to display
- `flags` (unsigned number) drawing flags:
  - `0 or not specified` display with no decimal (like abs())
  - `PREC1` display with one decimal place (number 386 is displayed as 38.6)
  - `PREC2` display with tow decimal places (number 386 is displayed as 3.86)
  - other general LCD flag also apply
  - `SHADOWED` Horus only, apply a shadow effect

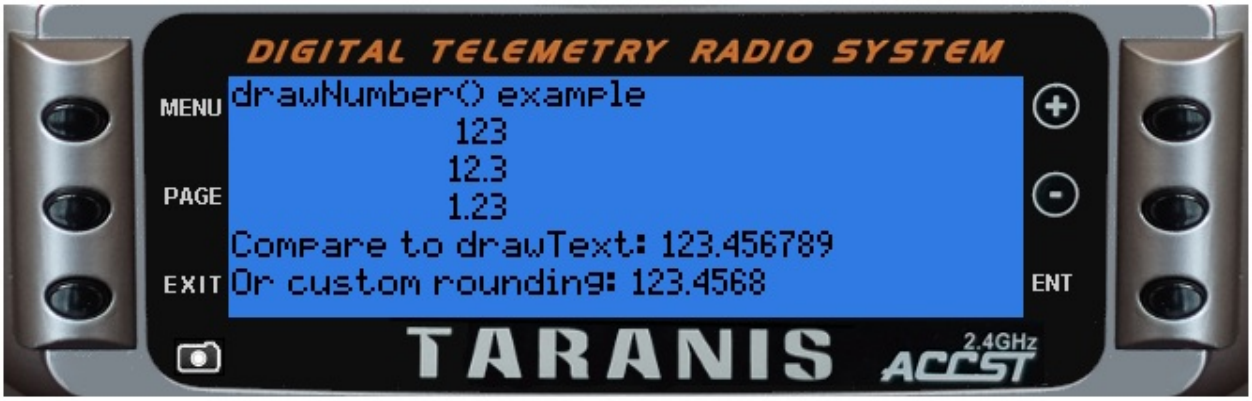
Return value

none

Examples

lcd/drawNumber-example

```
function round(num, decimals)
  local mult = 10^(decimals or 0)
  return math.floor(num * mult + 0.5) / mult
end

local function run(event)
  lcd.clear()
  lcd.drawText(1,1,"drawNumber() example", 0)
  local myNumber = 123.456789
  lcd.drawNumber(75, 11, myNumber, 0)
  lcd.drawNumber(75, 21, myNumber, PREC1)
  lcd.drawNumber(75, 31, myNumber, PREC2)
  lcd.drawText(1, 41, "Compare to drawText: " .. myNumber, 0)
  lcd.drawText(1, 51, "Or custom rounding: " .. round(myNumber, 4), 0)
end

return{run=run}
```



lcd.drawPixmap(x, y, name)

Draw a bitmap at (x,y)

@status current Introduced in 2.0.0

Parameters

- `x,y` (positive numbers) starting coordinates
- `name` (string) full path to the bitmap on SD card (i.e. “/IMAGES/test.bmp”)

Return value

none

Notice

Only available on Taranis X9 series. Maximum image size if 106 x 64 pixels (width x height).

Examples

lcd/drawPixmap-example

```
local function run(event)
  lcd.clear()
  lcd.drawText(1,1,"drawPixmap() example", 0)
  lcd.drawPixmap(96, 0, "/bmp/lua.bmp")
end

return{run=run}
```

lcd.drawPoint(x, y)

Draw a single pixel at (x,y) position

@status current Introduced in 2.0.0

Parameters

- `x` (positive number) x position
- `y` (positive number) y position

Return value

none

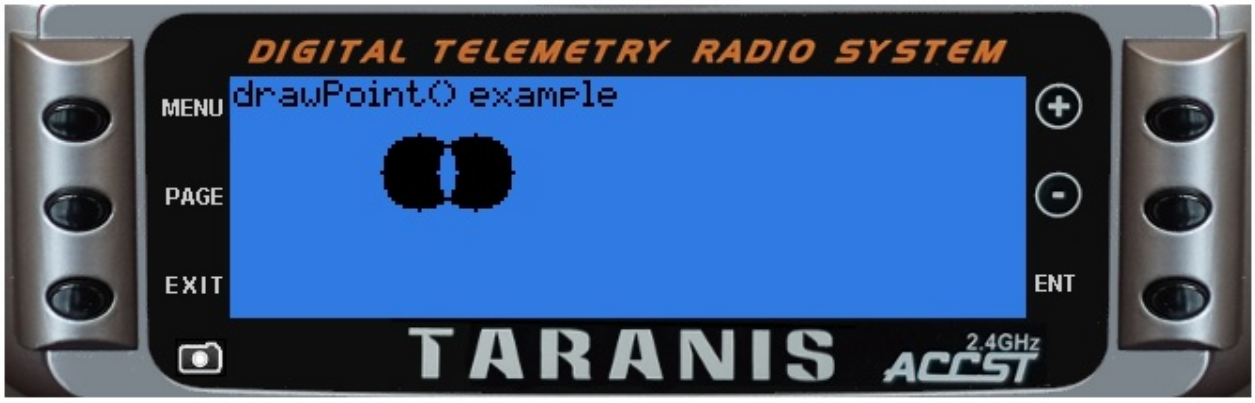
Notice

Taranis has an LCD display width of 212 pixels and height of 64 pixels. Position (0,0) is at top left. Y axis is negative, top line is 0, bottom line is 63. Drawing on an existing black pixel produces white pixel (TODO check this!)

Examples

lcd/drawPoint-example

```
local function circle(xCenter, yCenter, radius)
  local y, x
  for y=-radius, radius do
    for x=-radius, radius do
        if(x*x+y*y <= radius*radius) then
            lcd.drawPoint(xCenter+x, yCenter+y)
        end
    end
  end
end


local function run(event)
  lcd.clear()
  lcd.drawText(1,1,"drawPoint() example", 0)
  circle(50, 25, 10)
  circle(65, 25, 10)
end

return{run=run}
```



lcd.drawRectangle(x, y, w, h [, flags [, t]])

Draw a rectangle from top left corner (x,y) of specified width and height

@status current Introduced in 2.0.0, changed in 2.2.0

Parameters

- `x,y` (positive numbers) top left corner position
- `w` (number) width in pixels
- `h` (number) height in pixels
- `flags` (unsigned number) drawing flags
- `t` (number) thickness in pixels, defaults to 1 (only on Horus)

Return value

none

Examples

lcd/drawRectangle-example

```
local function run()
lcd.clear()
lcd.drawText(10,22,"drawRectangle()",DBLSIZE)
lcd.drawRectangle(5, 5, 150, 50, SOLID)
lcd.drawRectangle(6, 6, 150, 50, GREY_DEFAULT)
lcd.drawRectangle(7, 7, 150, 50, SOLID)
lcd.drawRectangle(8, 8, 150, 50, GREY_DEFAULT)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
drawRectangle
ENT
TARANIS
2.4GHz
ACCST

lcd.drawScreenTitle(title, page, pages)

Draw a title bar

@status current Introduced in 2.0.0

Parameters

- `title` (string) text for the title
- `page` (number) page number
- `pages` (number) total number of pages. Only used as indicator on the right side of title bar. (i.e. idx=2, cnt=5, display `2/5`)

Return value

none

Notice

Only available on Taranis

Examples

lcd/drawScreenTitle-example

```
local function run(event)
  lcd.clear()
  lcd.drawText(20, 20,"drawScreenTitle", DBLSIZE + BLINK)
  lcd.drawScreenTitle("This screen has one page", 1, 1)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
This screen has one page
1/1
PAGE
drawScreenTitle
EXIT
ENT
TARANIS
2.4GHz
ACCST

lcd.drawSource(x, y, source [, flags])

Displays the name of the corresponding input as defined by the source at (x,y)

@status current Introduced in 2.0.0

Parameters

- `x,y` (positive numbers) starting coordinate
- `source` (number) source index
- `flags` (unsigned number) drawing flags

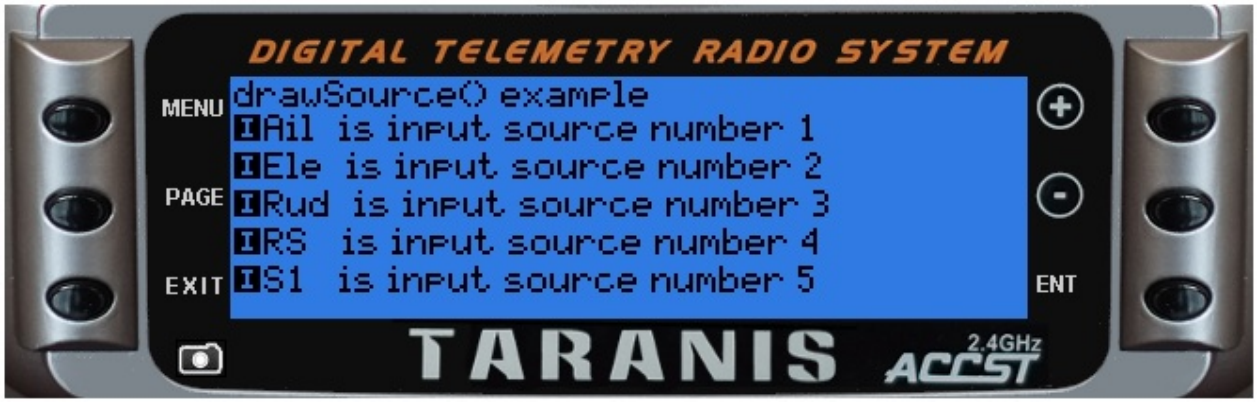
Return value

none

Examples

lcd/drawSource-example

```
local function run(event)
  local source
  lcd.clear()
  lcd.drawText(1, 1,"drawSource() example", 0)
  for source = 1, 5 do
   lcd.drawSource(1, source * 10, source, 0)
   lcd.drawText(lcd.getLastPos(), source * 10, " is input source number " .. source)
  end
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
ENT
drawSource() example
Ail is input source number 1
Ele is input source number 2
Rud is input source number 3
RS is input source number 4
S1 is input source number 5
TARANIS
2.4GHz
ACCST

lcd.drawSwitch(x, y, switch, flags)

Draw a text representation of switch at (x,y)

@status current Introduced in 2.0.0

Parameters

- `x,y` (positive numbers) starting coordinate
- `switch` (number) number of switch to display, negative number displays negated switch
- `flags` (unsigned number) drawing flags, only SMLSIZE, BLINK and INVERS.

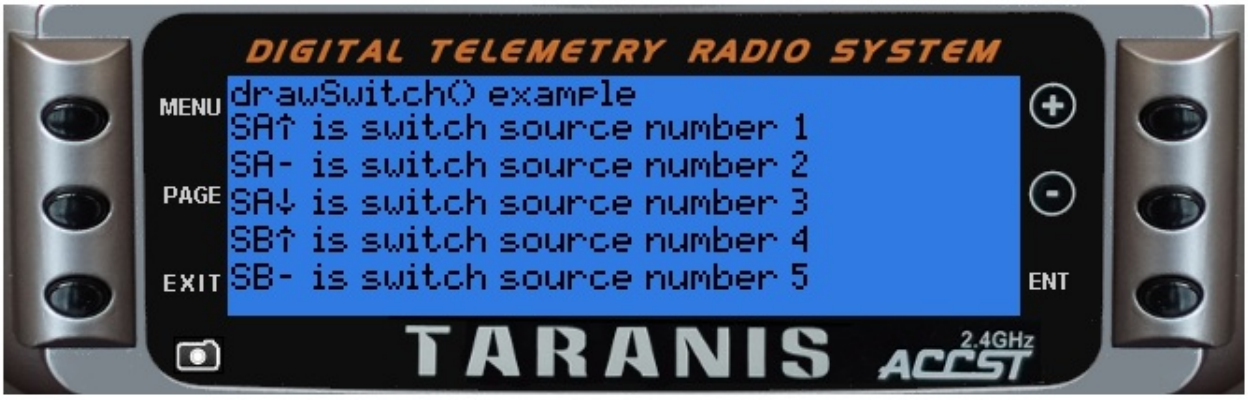
Return value

none

Examples

lcd/drawSwitch-example

```
local function run(event)
  local source
  lcd.clear()
  lcd.drawText(1, 1,"drawSwitch() example", 0)
  for source = 1, 5 do
   lcd.drawSwitch(1, source * 10, source, 0)
   lcd.drawText(20, source * 10, " is switch source number " .. source)
  end
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
PAGE
EXIT
drawSwitch() example
SA↑ is switch source number 1
SA- is switch source number 2
SA↓ is switch source number 3
SB↑ is switch source number 4
SB- is switch source number 5
ENT
TARANIS
2.4GHz
ACCST

lcd.drawText(x, y, text [, flags])

Draw a text beginning at (x,y)

@status current Introduced in 2.0.0, `SHADOWED` introduced in 2.2.1

Parameters

- `x,y` (positive numbers) starting coordinate
- `text` (string) text to display
- `flags` (unsigned number) drawing flags. All values can be combined together using the + character. ie BLINK + DBLSIZE. See the Appendix for available characters in each font set.
  - `0 or not specified` normal font
  - `XXLSIZE` jumbo sized font
  - `DBLSIZE` double size font
  - `MIDSIZE` mid sized font
  - `SMLSIZE` small font
  - `INVERS` inverted display
  - `BLINK` blinking text
  - `SHADOWED` Horus only, apply a shadow effect

Return value

none

Examples

lcd/drawText-example

```
local function run(event)
  lcd.clear()
  lcd.drawText(1, 1,"drawText() example", 0)
  lcd.drawText(1, 11,"0 - default", 0)
  lcd.drawText(1, 21,"BLINK", BLINK)
  lcd.drawText(1, 31,"INVERS + BLINK", INVERS + BLINK)
  lcd.drawText(120, 1,"XXLSIZE", DBLSIZE)
  lcd.drawText(120, 21,"MIDSIZE", MIDSIZE)
  lcd.drawText(120, 36,"SMLSIZE", SMLSIZE)
end

return{run=run}
```

lcd.drawTimer(x, y, value [, flags])

Display a value formatted as time at (x,y)

@status current Introduced in 2.0.0, `SHADOWED` introduced in 2.2.1

Parameters

- `x,y` (positive numbers) starting coordinate
- `value` (number) time in seconds
- `flags` (unsigned number) drawing flags:
  - `0 or not specified` normal representation (minutes and seconds)
  - `TIMEHOUR` display hours
  - other general LCD flag also apply
  - `SHADOWED` Horus only, apply a shadow effect

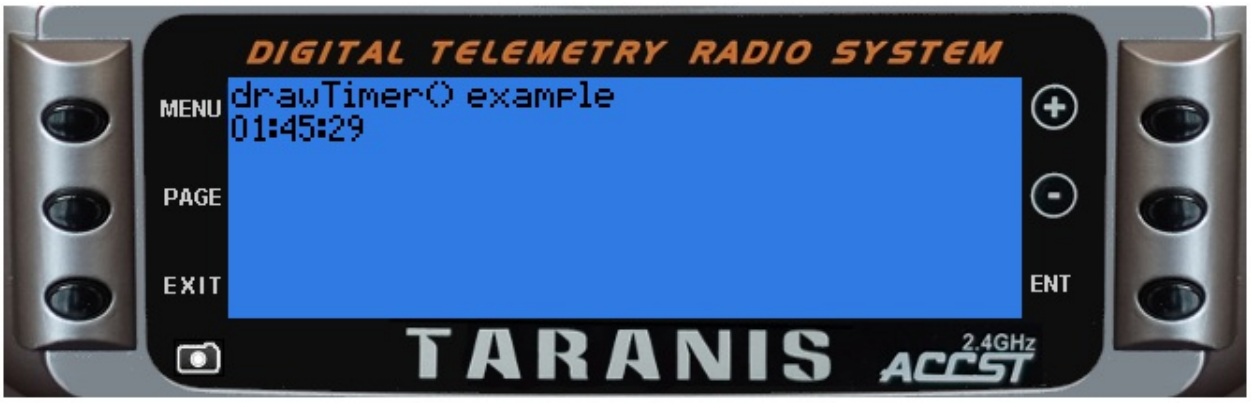
Return value

none

Examples

lcd/drawTimer-example

```
local upTime

local function background()
  upTime = getTime() / 100
end

local function run(event)
  background()
  lcd.clear()
  lcd.drawText(1, 1,"drawTimer() example", 0)
  lcd.drawTimer(1, 10, upTime, TIMEHOUR)
end

return{run=run}
```


DIGITAL TELEMETRY RADIO SYSTEM
MENU
drawTimer() example
01:45:29
PAGE
EXIT
ENT
TARANIS
2.4GHz
ACCST

lcd.getLastLeftPos()

Returns the leftmost x position from previous drawtext or drawNumber output

@status current Introduced in 2.2.0

Parameters

none

Return value

- `number` (integer) x position

Notice

Only available on Taranis

lcd.getLastPos()

Returns the rightmost x position from previous output

@status current Introduced in 2.0.0

Parameters

none

Return value

- `number` (integer) x position

Notice

Only available on Taranis

For added clarity, it is recommended to use lcd.getLastRightPos()

lcd.getLastRightPos()

Returns the rightest x position from previous drawtext or drawNumber output

@status current Introduced in 2.2.0

Parameters

none

Return value

- `number` (integer) x position

Notice

Only available on Taranis

This is strictly equivalent to former lcd.getLastPos()

lcd.refresh()

Refresh the LCD screen

@status current Introduced in 2.2.0

Parameters

none

Return value

none

Notice

This function only works in stand-alone and telemetry scripts.

lcd.setColor(area, color)

Set a color for specific area

@status current Introduced in 2.2.0

Parameters

- `area` (unsigned number) specific screen area in the list bellow
  - `CUSTOM_COLOR`
  - `TEXT_COLOR`
  - `TEXT_BGCOLOR`
  - `TEXT_INVERTED_COLOR`
  - `TEXT_INVERTED_BGCOLOR`
  - `LINE_COLOR`
  - `SCROLLBOX_COLOR`
  - `MENU_TITLE_BGCOLOR`
  - `MENU_TITLE_COLOR`
  - `MENU_TITLE_DISABLE_COLOR`
  - `HEADER_COLOR`
  - `ALARM_COLOR`
  - `WARNING_COLOR`
  - `TEXT_DISABLE_COLOR`
  - `HEADER_COLOR`
  - `CURVE_AXIS_COLOR`
  - `CURVE_CURSOR_COLOR`
  - `TITLE_BGCOLOR`
  - `TRIM_BGCOLOR`
  - `TRIM_SHADOW_COLOR`
  - `MAINVIEW_PANES_COLOR`
  - `MAINVIEW_GRAPHICS_COLOR`
  - `HEADER_BGCOLOR`
  - `HEADER_ICON_BGCOLOR`
  - `HEADER_CURRENT_BGCOLOR`
  - `OVERLAY_COLOR`
- `color` (number) color in 5/6/5 rgb format. The following prefined colors are available
  - `WHITE`
  - `GREY`

- `LIGHTGREY`
- `DARKGREY`
- `BLACK`
- `YELLOW`
- `BLUE`
- `RED`
- `DARKRED`

Return value

none

Notice

Only available on Horus

Bitmap Functions

Bitmap.getSize(name)

Return width, height of a bitmap object

@status current Introduced in 2.2.0

Parameters

- `bitmap` (pointer) point to a bitmap previously opened with Bitmap.open()

Return value

- `multiple` returns 2 values:
  - (number) width in pixels
  - (number) height in pixels

Notice

Only available on Horus

Bitmap.open(name)

Loads a bitmap in memory, for later use with lcd.drawBitmap(). Bitmaps should be loaded only once, returned object should be stored and used for drawing. If loading fails for whatever reason the resulting bitmap object will have width and height set to zero.

Bitmap loading can fail if:

- File is not found or contains invalid image
- System is low on memory
- Combined memory usage of all Lua script bitmaps exceeds certain value

@status current Introduced in 2.2.0

Parameters

- `name` (string) full path to the bitmap on SD card (i.e. “/IMAGES/test.bmp”)

Return value

- `bitmap` (object) a bitmap object that can be used with other bitmap functions

Notice

Only available on Horus

Part IV - Converting OpenTX 2.0 Scripts

The handling of telemetry data is significantly improved in OpenTX 2.1. However, in order to support the additional flexibility of having multiple sensors of the same type, many Lua scripts referencing GPS and Lipo sensor data will require revision.

This section also covers some of the requirements for scripts that are necessary for them to function properly under both OpenTX 2.1 and OpenTX 2.0.

General Issues in converting scripts written for OpenTX 2.0

Deprecated Functions

lcd.Lock() is deprecated, will be obsolete in 2.2. Lua scripts must now explicitly call lcd.Clear() and re-draw the whole display if necessary.

> TODO: research killEvents() and use of keys in telemetry scripts

Obsolete Telemetry Field Names

OpenTX since version 2.1 provides more flexibility in the number and type of supported remote sensors. As a result, several field name constants are obsolete and need to be modified in scripts originally written for OpenTX 2.0.

GPS field names are covered in Handling GPS Sensor Data

Lipo voltage field names (LVSS) are covered in Handling Lipo Sensor Data

Maintaining compatibility with OpenTX 2.0

Automatic invocation of the background function - Beginning in OpenTX 2.1 the background() function is called automatically prior to each invocation of the run() function. Under 2.0 you must explicitly call your background function within your run function.

Handling GPS Sensor data

Overview

With OpenTx 2.2 it is possible to have multiple GPS sensors, each with their own set of telemetry values which may have user-assigned names.

Value names are case sensitive and may include some or all of the following:

- GPS (latitude and longitude as a lua table containing [lat] and [lng])
- GSpd (speed in knots)
- GAlt (altitude in meters)
- Date (gps date converted to local time as a lua table containing [year] [mon] [day] [hour] [min] [sec])
- Hdg (heading in degrees true)

This example demonstrates getting latitude and longitude from a sensor with the default name of 'GPS'

```
local gpsValue = "unknown"

local function rnd(v,d)
    if d then
     return math.floor((v*10^d)+0.5)/(10^d)
    else
     return math.floor(v+0.5)
    end
end

local function getTelemetryId(name)
    field = getFieldInfo(name)
    if field then
      return field.id
    else
      return -1
    end
end

local function init()
  gpsId = getTelemetryId("GPS")
end

local function background()
  gpsLatLon = getValue(gpsId)
  if (type(gpsLatLon) == "table") then
    gpsValue = rnd(gpsLatLon["lat"],4) .. ", " .. rnd(gpsLatLon["lon"],4)
  else
    gpsValue = "not currently available"
  end
end

local function run(e)
  lcd.clear()
  background() -- update current GPS position
  lcd.drawText(1,1,"OpenTX 2.2 GPS example",0)
  lcd.drawText(1,11,"GPS:", 0)
  lcd.drawText(lcd.getLastPos()+2,11,gpsValue,0)
end

return{init=init,run=run,background=background}
```

Handling Lipo Sensor Data

With OpenTx 2.2 it is possible to have multiple Lipo sensors, each with a user-assigned name. The call to getValue() returns a table with the current voltage of each of the cells it is monitoring.

This example demonstrates getting Lipo cell voltage from a sensor with the default name of 'Cels'

Example:

```
local cellValue = "unknown"
local cellResult = nil
local cellID = nil

local function getTelemetryId(name)
    field = getFieldInfo(name)
    if field then
      return field.id
    else
      return -1
    end
end

local function init()
  cellId = getTelemetryId("Cels")
end

local function background()
  cellResult = getValue(cellId)
  if (type(cellResult) == "table") then
    cellValue = ""
    for i, v in ipairs(cellResult) do
      cellValue = cellValue .. i .. ": " .. v .. " "
    end
  else
    cellValue = "telemetry not available"
  end
end

local function run(e)
  background()
  lcd.clear()
  lcd.drawText(1,1,"OpenTX 2.2 cell voltage example",0)
  lcd.drawText(1,11,"Cels:", 0)
  lcd.drawText(lcd.getLastPos()+2,11,cellValue,0)
end

return{init=init,run=run,background=background}
```

Part V - Converting OpenTX 2.1 Scripts

This section also covers some of the requirements for scripts that are necessary for them to function properly under both OpenTX 2.2.

New features

- LUA Widgets (Horus only)
- LUA Themes (Horus only)

Changes

- Lua Themes and Widgets run in a separate Lua environment. They are isolated from the other Lua environment which runs other scripts. This means they can not share variables, etc... (Horus only)
- Function scripts can have a `background()` function defined (similar to the Telemetry scripts). It will be called periodically when the switch that activates it is FALSE.
- Horus doesn't support Telemetry scripts.
- Telemetry and Mix scripts maximum file name length (without extension) was reduced from 8 to 6 characters.
- Telemetry and Mix scripts maximum number of inputs reduced from 8 to 6

LCD Functions

- Function `lcd.lock()` was removed.
- New function `lcd.refresh()`.
- Default number alignment changed from RIGHT to LEFT.
- `lcd.getLastPos()` is not available on Horus
- Functions only available on Horus:
  - `lcd.drawBitmap()`
  - `lcd.setColor()`
  - `lcd.RGB()`
- Functions only available on Taranis:

    - `lcd.drawPixmap`
    - `lcd.drawScreenTitle`
    - `lcd.drawCombobox`

General Functions

- `RIGHT` added
- Rotary encoder events added:
    - `EVT_ROT_BREAK`
    - `EVT_ROT_LONG`
    - `EVT_ROT_LEFT`
    - `EVT_ROT_RIGHT`

Part VI - Advanced Topics

The advanced topics section covers file i/o, data sharing, and debugging techniques

Lua data sharing across scripts

Overview:

OpenTX considers all function, mix, and telemetry scripts to be 'permanent' scripts that share the same runtime environment. They are typically loaded at power up or when a new model is selected. However, they are also reinitialized when a script is added or removed during model editing.

Lua scoping rules:

Any variable or function not declared local is implicitly global. Care must be taken to avoid unintentional global declarations, and ensure that the globals you intentionally declare have unique names to avoid conflicts with scripts written by others.

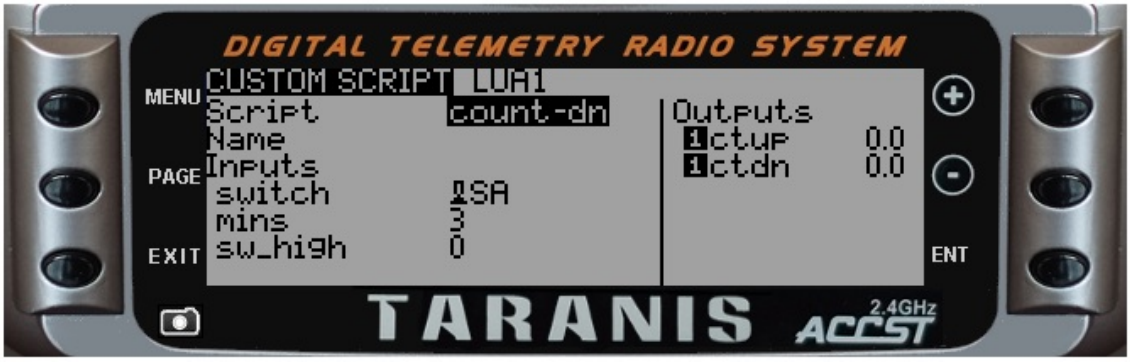
Example:

This example consists of three scripts

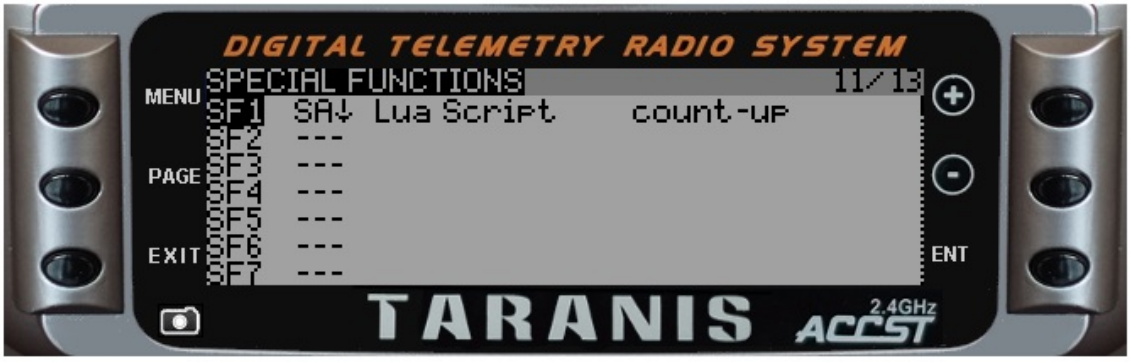
- **count-dn.lua** - this is a mix script than can be run stand alone to announce time remaining based on a user-defined switch and duration. It updates two global variables (gCountUp and gCountDown). It also creates output values (ctup and ctdn) which are for demonstration purposes only.
- **count-up.lua** - this is an optional function script which will do count up announcements based on harded coded values.
- **shocount.lua** - this is an optional telemetry script which simply shows the current values of the gCountUp and gCountDown variables.

Installation:

- count-dn.lua
  - copy to /SCRIPTS/MIXES
  - configure on the transmitter CUSTOM SCRIPT page
    - suggested switch = "SA"
    - suggested mins = 3
    - suggested sw_high = 0
  - screen image:



- count-up.lua
  - copy to /SCRIPTS/FUNCTIONS
  - configure on the transmitter SPECIAL FUNCTIONS page
    - suggested switch SA(down)
  - screen image:



- shocount.lua
  - copy to /SCRIPTS/TELEMETRY
  - configure on the transmitter TELEMETRY page
  - screen image:

Script sources:

count-dn.lua

```
-- these globals can be referenced in function and telemetry scripts
gCountUp = 0
gCountDown = 0
```

```
local target
local running = false
local complete = false
local announcements = { 720, 660, 600, 540, 480, 420, 360, 300, 240, 180, 120, 105, 90
, 75, 60, 55, 50, 45, 40, 35, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17,
16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}
local annIndex -- index into the announcements table (1 based)
local minUnit -- used by playNumber() for unit announcement

local input =
    {
        { "switch", SOURCE},         -- switch used to activate count down
        { "mins", VALUE, 1, 12, 2 },     -- minutes to count down
        { "sw_high", VALUE, 0, 1, 1 } -- 0 = active when low, otherwise active when hi
gh
    }

local output = {"ctup", "ctdn" }

local function init()
  local version = getVersion()
  if version < "2.1" then
    minUnit = 16  -- unit for minutes in OpenTX 2.0
  elseif version < "2.2" then
    minUnit = 23  -- unit for minutes in OpenTX 2.1
  else
    minUnit = 25  -- unit for minutes in OpenTX 2.2
  end
end

local function countdownIsRunning(switch, sw_high)
  -- evaluate switch - return true if we should be counting down
  if (sw_high > 0) then
    return (switch > -1000)
  else
    return (switch < 1000)
  end
end

local function run(switch, mins, sw_high)
  local timenow = getTime() -- 10ms tick count
  local minutes
  local seconds

  if (not countdownIsRunning(switch, sw_high)) then
      running = false
      complete = false
      return 0, 0 -- ***** NOTE: early exit *****
  end

  if (complete) then
    return 0, 0 -- must reset the switch before we go again
```

```
  end

  if (not running) then
    running = true
    target = timenow + ((mins * 60) * 100)
    annIndex = 1
  end

  gCountDown = math.floor(((target - timenow) / 100) + .7) --  + is adj. to for announ
cement lag
  gCountUp = (mins * 60) - gCountDown

  while gCountDown < announcements[annIndex] do
    annIndex = annIndex + 1 -- catch up
  end

  if gCountDown == announcements[annIndex] then
    minutes = math.floor(gCountDown / 60)
    seconds = gCountDown % 60
    if minutes > 0 then
      playNumber(minutes, minUnit, 0)
    end
    if seconds > 0 then
      playNumber(seconds, 0, 0)
    end
    annIndex = annIndex + 1
  end

  if gCountDown <= 0 then
    playNumber(0,0,0)
    running = false
    gCountDown = 0
    complete = true
  end

  return gCountUp * 10.24, gCountDown * 10.24
end

return { input=input, output=output, init=init, run=run }
```

count-up.lua

```
gCountUp = 0

local min = 5
local max = 30
local last = 0
local announcements = { 5, 10, 15, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 }
local annIndex = 1

local function run(e)
  if not (gCountUp == last) then
    last = gCountUp
    for key, value in pairs(announcements) do
        if value == last then
          playNumber(last, 0, 0)
        end
    end
  end
end

return{run=run}
```

shocount.lua

```
-- these globals can be referenced in mix, function, and telemetry scripts
gCountUp = 0
gCountDown = 0

local function run(e)
  lcd.clear()
  lcd.drawText(1,1,"OpenTx Lua Data Sharing",0)

  lcd.drawText(1,11,"gCountUp:", 0)
  lcd.drawText(lcd.getLastPos()+2,11,gCountUp,0)
  lcd.drawText(1, 21, "gCountDown:", 0)
  lcd.drawText(lcd.getLastPos()+2,21,gCountDown,0)
end

return{run=run}
```

Debugging techniques

Debugging your code before testing

A good editor is key

It is always good practice to check your code on syntax before even testing it. There are several good LUA editors on the market, some of them for free. The ZeroBrane (https://studio.zerobrane.com/) suite is quite powerful, and very simple to use. In the rest of this article we will assume you use ZeroBrane, but the same techniques can be used in any powerful code editor.

You can set ZeroBrane to use the Scripts directory of your simulated transmitter SDCard image as a default directory, and it will show you all the files in a nice navigation tree.

If you open a LUA file, you will already have some markup in your screen, indicating errors or important info. In ZeroBrane for instance, a not declared variable will always get underlined, so that you are made aware you forgot to declare it, or you redeclared it by accident afterwards again.

Checking if the code can be compiled

The editor will have an "execute code" option, that will try to run the code on it's own interpreter (code processing engine). If there are any syntax errors, it will not be able to execute the code, and inform you about the errors. A common error in LUA is using a single equal sign (=) in a condition in an 'if' statement, whereas in LUA that should be a double equal sign (==). The interpreter will inform you about such an error ocurring, and mention the line where you made the error.

Since the OpenTX LUA environment has some own functions, like lcd.drawText(), the interpreter will 'complain' it cannot call an unspecified function, but it will check the entire syntax nonetheless.

Ready to run the code

In zerobrane, if you tried to run the code, it will first save it if it could be interpreted correctly. A common workflow would be:

- do some code corrections / additions
- try to run the code in the editor

- if the code gets compiled, the edited LUA file gets saved automatically
- run the code in the transmitter simulator
- check for the desired functionality
- restart this cycle

The lua debug viewer

In the later versions of the companion software, a LUA debug screen is available. So once you start your just syntaxically verified and saved LUA script, you can follow some of it's output and actions in the debug screen. It will tell you where and in what line an eventual crash occured.

Using a script loader

If you made some code changes, chances are that you have to do a whole sequence of key-clicks and actions on the transmitter simulator in order to retest the same script after those changes.

You can substatially reduce the effort of all this by using a script loader in your transmitter. This is nothing more then a function that will load and run your code. If you press the enter button, it will unload the current code, and ask if you want to run the code again. So, with just two clicks, you can unload the running code and reload your updated code. On the Taranis simulator, you can also reload the LUA scripts environment with just a buttonclick.

An example of such a script is found under the notes. You can adapt it for other types of scripts of course.

Notes

Script Loader

This script loader will load the file /SCRIPTS/TELEM/telem1.lua, run it, and wait for an Enter Break event. Once received, it will unload the code and wait for a next Enter Break event.

```
local fileToLoad="/SCRIPTS/TELEM/telem1.lua"
local active = true

local thisPage={}
local page={}

local function clearTable(t)
  if type(t)=="table" then
    for i,v in pairs(t) do
      if type(v) == "table" then
        clearTable(v)
      end
      t[i] = nil
    end
  end
  collectgarbage()
  return t
end

thisPage.init=function(...)
  if active then
    page=dofile(fileToLoad)
    page.init(...)
  end
  return true
end

thisPage.background=function(...)
  if active then
    page.background(...)
  end
  return true
end

thisPage.run=function(...)
  if active then
    page.run(...)
    active= not (...==EVT_ENTER_BREAK)
  else
    lcd.drawText( 15, 2, fileToLoad, 0 )
    lcd.drawText( 15, 20, "disabled", 0 )
    lcd.drawText( 15, 40,"press enter-button to activate",0)
    clearTable(page)
    active= (...==EVT_ENTER_BREAK)
    thisPage.init()
  end
  return not (...==EVT_MENU_BREAK)
end

return thisPage
```

Speed and Memory Optimization Tricks

Faster getValue()

Normally one uses getValue() function with the source/filed name like so:

```
local foo = getValue("bar")
```

This works and is recommended method for portability. But if a particular script needs to get the value of certain field a lot, then it is faster to use this syntax:

```
local my_id = getFiledInfo("bar").id    -- here we get the numerical id of the filed "
bar"

local function run_a_lot()
  local my_value = getValue(my_id)      -- exactly the same effect as local my_value =
 getValue("bar"), but faster
end
```

Why is this method faster? With the function getFieldInfo(name) we get the `numerical id` of the wanted filed. The function has to find the requested value by its name in the table of all available sources. That search takes some time.

When we use this syntax the search is only done once. In comparison in the first example the search must be performed every time `getValue("bar")` is called.

So when the `getValue(my_id)` is called the search can be skipped and the requested value if fetched directly.

Of course there is a trade-of, the second example uses little more memory (for variable `my_id`).

Part VII - Appendix

Various additional documents

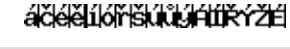
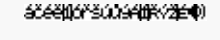
Fonts

Taranis X7 & X9 series

English (Default)

Font Set	Height	Available Characters
XXLSIZE	38px	
DBLSIZE	16px	
MIDSIZE	12px	
0 Default	8px	
SMLSIZE	6px	

Czech

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	
MIDSIZE	
0 Default	
SMLSIZE	

Finnish

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	ÀàÒÙÜß
MIDSIZE	ÀàÒÙÜß
SMLSIZE	ÀàÒÙÜß
0 Default	ÀàÒÙÜß

French

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	éèàâîç
MIDSIZE	éèàâîç
0 Default	éèàâîç
SMLSIZE	éèàâîç

German

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	ÄäÖöÜüß
MIDSIZE	ÄäÖöÜüß
0 Default	ÄäÖöÜüß
SMLSIZE	ÄäÖöÜüß

Italian

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	àù
MIDSIZE	àù
0 Default	àù
SMLSIZE	àù

Polish

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	ąćęłńóśźżĄĆĘŁŃÓŚŹŻ
MIDSIZE	ąćęłńóśźżĄĆĘŁŃÓŚŹŻ
0 Default	ąćęłńóśźżĄĆĘŁŃÓŚŹŻ
SMLSIZE	

Portuguese

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	ÁáÂâÃãÀàÇçÉéÊêÍíÓóÔôÕõÚú
MIDSIZE	ÁáÂâÃãÀàÇçÉéÊêÍíÓóÔôÕõÚú
0 Default	ÁáÂâÃãÀàÇçÉéÊêÍíÓóÔôÕõÚú
SMLSIZE	

Spanish

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	ÑñÑ
MIDSIZE	ÑñÑ
0 Default	ÑñÑ
SMLSIZE	ÑñÑ

Swedish

Font Set	Available Characters
XXLSIZE	--
DBLSIZE	
MIDSIZE	
0 Default	
SMLSIZE	

OpenTx 2.2 Units reference

Index	Unit	Defined as
0	Raw unit (no unit)	UNIT_RAW
1	Volts	UNIT_VOLTS
2	Amps	UNIT_AMPS
3	Milliamps	UNIT_MILLIAMPS
4	Knots	UNIT_KTS
5	Meters per Second	UNIT_METERS_PER_SECOND
6	Feet per Second	UNIT_FEET_PER_SECOND
7	Kilometers per Hour	UNIT_KMH
8	Miles per Hour	UNIT_MPH
9	Meters	UNIT_METERS
10	Feet	UNIT_FEET
11	Degrees Celsius	UNIT_CELSIUS
12	Degrees Fahrenheit	UNIT_FAHRENHEIT
13	Percent	UNIT_PERCENT
14	Milliamp Hour	UNIT_MAH
15	Watts	UNIT_WATTS
16	Milliwatts	UNIT_MILLIWATTS
17	dB	UNIT_DB
18	RPM	UNIT_RPMS
19	G	UNIT_G
20	Degrees	UNIT_DEGREE
21	Radians	UNIT_RADIANS
22	Milliliters	UNIT_MILLILITERS
23	Fluid Ounces	UNIT_FLOZ
24	Hours	UNIT_HOURS
25	Minutes	UNIT_MINUTES
26	Seconds	UNIT_SECONDS

27		UNIT_CELLS
28		UNIT_DATETIME
29		UNIT_GPS
30		UNIT_BITFIELD
31		UNIT_TEXT