TA4 Lua Controller

The TA4 Lua Controller is a small computer, programmable in Lua to control your machinery. In contrast to the ICTA Controller this controller allows to implement larger and smarter control and monitoring tasks.

But to write Lua scripts, some knowledge with the programming language Lua is required.

Minetest uses Lua 5.1. The reference document for Lua 5.1 is here. The book Programming in Lua (first edition) is also a perfect source for learning Lua.

Table of Contents

TA4 Lua Controller Blocks

TA4 Lua Controller

The controller block has a menu form with the following tabs:

- the `init` tab for the initialization code block (see "Initialization, Loops, and Events")
- the `func` tab for the Lua functions (see "Initialization, Loops, and Events")
- the `loop` tab for the main code block (see "Initialization, Loops, and Events")
- the `outp` tab for debugging outputs via `$print()`
- the `notes` tab for your code snippets or other notes (like a clipboard)
- the `help` tab with information to the available commands (see "Techage Commands")

The controller needs power to work. A battery pack has to be placed neadby.

Battery

The battery pack has to be placed near the controller (1 block distance). The needed battery power is directly dependent on the CPU time the controller consumes. Because of that, it is important to optimize the execution time of the code (which helps the admin to keep server lags down :))

The controller will be restarted (init() is called) every time the Minetest server starts again. To store data non-volatile (to pass a server restart), the Central Server has to be used.

Central Server

The Server block is used to store data from the controllers nonvolatile. It can also be used for communication purposes between several Controllers. The Server has a form to enter valid usernames for server access.

For special Server commands, see "Server and Terminal commands"

TA4 Lua Controller Terminal

The Terminal is used to send command strings to the controller. In turn, the controller can send text strings to the terminal. The Terminal has a help system for internal commands. Its supports the following commands:

- `clear` = clear the screen
- `help` = output this message
- `pub` = switch terminal to public use (everybody can enter commands)
- `priv` = switch terminal to private use (only the owner can enter commands)
- `send <num> on/off` = send on/off event to e. g. lamps (for testing purposes)
- `msg <num> <text>` = send a text message to another Controller (for testing purposes)

For special Terminal commands for the TA4 Lua Controller, see "Terminal Commands"

TA4 Sensor Chest

tbd.

Lua Functions and Environment

Lua Functions and Limitations

The controller uses a subset of the language Lua, called SaferLua. It allows the safe and secure execution of Lua scripts, but has the following limitations:

- limited code length
- limited execution time
- limited memory use
- limited possibilities to call functions

SaferLua follows the standard Lua syntax with the following restrictions:

- no `while` or `repeat` loops (to prevent endless loops)
- no table constructor {..}, see "Arrays, Stores, and Sets" for comfortable alternatives
- limited runtime environment

SaferLua directly supports the following standard functions:

- math.floor
- math.abs

- math.max
- math.min
- math.random
- tonumber
- tostring
- unpack
- type
- string.byte
- string.char
- string.find
- string.format
- string.gmatch
- string.gsub
- string.len
- string.lower
- string.match
- string.rep
- string.sub
- string.upper
- string.split
- string.trim

For own function definitions, the menu tab 'func' can be used. Here you write your functions like:

```
function foo(a, b)
    return a + b
end
```

Each SaferLua program has access to the following system variables:

- ticks - a counter which increments by one each call of `loop()`
- elapsed - the amount of seconds since the last call of `loop()`
- event - a boolean flag (true/false) to signal the execution of `loop()` based on an occurred event

Arrays, Stores, and Sets

It is not possible to easily control the memory usage of a Lua table at runtime. Therefore, Lua tables can't be used for SaferLua programs. Because of this, there are the following alternatives, which are secure shells over the Lua table type:

Arrays

Arrays are lists of elements, which can be addressed by means of an index. An index must be an integer number. The first element in an *array* has the index value 1. *Arrays* have the following methods:

- add(value) - add a new element at the end of the array
- set(idx, value) - overwrite an existing array element on index `idx`
- get(idx) - return the value of the array element on index `idx`
- remove(idx) - remove the array element on index `idx`
- insert(idx, val) - insert a new element at index `idx` (the array becomes one element longer)
- size() - return the number of *array* elements
- memsize() - return the needed *array* memory space

- next() - `for` loop iterator function, returning `idx,val`
- sort(reverse) - sort the *array* elements in place. If *reverse* is `true`, sort in descending order.

Example:

```
a = Array(1,2,3,4)      --> {1,2,3,4}
a.add(6)                --> {1,2,3,4,6}
a.set(2, 8)             --> {1,8,3,4,6}
a.get(2)                --> function returns 8
a.insert(5,7)           --> {1,8,3,4,7,6}
a.remove(3)             --> {1,8,4,7,6}
a.insert(1, "hello")    --> {"hello",1,8,4,7,6}
a.size()                --> function returns 6
a.memsize()             --> function returns 10
for idx,val in a.next() do
    ...
end
```

Stores

Unlike *arrays*, which are indexed by a range of numbers, *stores* are indexed by keys, which can be a string or a number. The main operations on a *store* are storing a value with some key and extracting the value given the key. The *store* has the following methods:

- set(key, val) - store/overwrite the value `val` behind the keyword `key`
- get(key) - read the value behind `key`
- del(key) - delete a value
- size() - return the number of *store* elements
- memsize() - return the needed *store* memory space
- next() - `for` loop iterator function, returning `key,val`
- keys(order) - return an *array* with the keys. If *order* is `"up"` or `"down"`, return the keys as sorted *array*, in order of the *store* values.

Example:

```
s = Store("a", 4, "b", 5)  --> {a = 4, b = 5}
s.set("val", 12)           --> {a = 4, b = 5, val = 12}
s.get("val")               --> returns 12
s.set(0, "hello")          --> {a = 4, b = 5, val = 12, [0] = "hello"}
s.del("val")               --> {a = 4, b = 5, [0] = "hello"}
s.size()                   --> function returns 3
s.memsize()                --> function returns 9
for key,val in s.next() do
    ...
end
```

Keys sort example:

```
s = Store()             --> {}
s.set("Joe", 800)       --> {Joe=800}
s.set("Susi", 1000)     --> {Joe=800, Susi=1000}
s.set("Tom", 60)        --> {Joe=800, Susi=1000, Tom=60}
s.keys()                --> {Joe, Susi, Tom}
s.keys("down")          --> {Susi, Joe, Tom}
s.keys("up")            --> {Tom, Joe, Susi}
```

Sets

A *set* is an unordered collection with no duplicate elements. The basic use of a *set* is to test if an element is in the *set*, e.g. if a player name is stored in the *set*. The *set* has the following methods:

- add(val) - add a value to the *set*
- del(val) - delete a value from the *set*
- has(val) - test if value is stored in the *set*
- size() - return the number of *set* elements
- memsize() - return the needed *set* memory space
- next() - `for` loop iterator function, returning `idx,val`

Example:

```
s = Set("Tom", "Lucy")      --> {Tom = true, Lucy = true}
s.add("Susi")               --> {Tom = true, Lucy = true, Susi = true}
s.del("Tom")                --> {Lucy = true, Susi = true}
s.has("Susi")               --> function returns `true`
s.has("Mike")               --> function returns `false`
s.size()                    --> function returns 2
s.memsize()                 --> function returns 8
for idx,val in s.next() do
    ...
end
```

All three types of data structures allow nested elements, e.g. you can store a *set* in a *store* or an *array* and so on. But note that the overall size over all data structures can't exceed the predefined limit. This value is configurable for the server admin. The default value is 1000. The configured limit can be determined via `memsize()`:

```
memsize()  --> function returns 1000  (example)
```

Initialization, Loops, and Events

The TA4 Lua Controller distinguishes between the initialization phase (just after the controller was started) and the continuous operational phase, in which the normal code is executed.

Initialization

During the initialization phase the function `init()` is executed once. The `init()` function is typically used to initialize variables, e.g. clean the display, or reset other blocks:

```
-- initialize variables
counter = 1
table = Store()
player_name = "unknown"

# reset blocks
$clear_screen("0123")      -- "0123" is the number of the display
$send_cmnd("2345", "off")  -- turn off the blocks with the number "2345"
```

Loops

During the continuous operational phase the `loop()` function is typically called every second. Code witch should be executed cyclically has to be placed here. The cycle frequency is per default once per second, but can be changed via:

```
$loopcycle(0)   -- no loop cyle any more
$loopcycle(1)   -- call the loop function every second
$loopcycle(10)  -- call the loop function only every 10 seconds
```

The provided number must be an integer value. The cycle frequency can be changed in the `init()` function, but also in the `loop()` function.

Events

To be able to react directly on received commands, the TA4 Lua Controller supports events. Events are usually turned off, but can be activated with the command `events()`:

```
$events(true)    -- enable events
$events(false)   -- disable events
```

If an event occurs (a command was received from another block), the `loop()` is executed (in addition to the normal loop cycle). In this case the system variable 'event' is set:

```
if event then
    -- event has occurred
    if $input("3456") == "on" then  -- check input from block with the number
"3456"
        -- do some action...
    end
end
```

The first occurred event will directly processed, further events may be delayed. The TA4 Lua Controller allows a maximum of one event every 100 ms.

Techage Commands

For the communication with other blocks the controller supports the following commands:

Controller local commands

- `$print(text, text, text)` - Output a text string on the 'outp' tab of the controller menu. The function accepts up to three text arguments. E.g.: `$print("Hello ", name, " !")`
- `$loopcycle(seconds)` - This function allows to change the call frequency of the controllers loop() function, witch is per default one second. For more info, see "Loops and Events".
- `$events(bool)` - Enable/disable event handling. For more info, see "Loops and Events"
- `$get_ms_time()` - Returns time with millisecond precision.
- `$time_as_str()` - Read the time of day (ingame) als text string in 24h format, like "18:45".
- `$time_as_num()` - Read the time of day (ingame) als integer number in 24h format, like 1845.
- `$get_input(num)` - Read one input value provided by an external block with the given number *num*. The block has to be configured with the number of the controller to be able to send status messages (on/off commands) to the controller. *num* is the number of the remote block, like "1234".

Input Example

- a Player Detector with number "0001" is configured to send on/off commands to a block with number "0002".
- The TA4 Lua Controller with number "0002" will receive these commands as input messages.
- The program on the SaferLua Controller can always read the last input message from block with number "0001" by means of:

```
sts = $get_input("0001")
```

Techage commands

- `$get_status(num)` - Read the status from an external block with the given number *num*. Standard blocks return a status string like: 'running', 'stopped', 'blocked', 'standby', 'fault', or "unloaded".
- `$get_player_action(num)` - Read the player action status from a TA4 Sensor Chest with the given number *num*. The function returns three values: player-name, action, item-name.
- `$get_fuel_value(num)` - Read fuel value from fuel consuming blocks. The block returns a number value. *num* is the number of the remote block, like "1234".
- `$get_load_value(num)` - Read the load value from a tank/storage block. The block returns a number (0..100). *num* is the number of the remote block, like "1234".
- `$get_delivered_value(num)` - Read the delivered power value from a generator block. The block returns a positive or negative number. Blocks like accus provide a negative value while charging. *num* is the number of the remote block, like "1234".
- `$playerdetector(num)` - Read the name status from a Player Detector with the number *num*. If no player is nearby, the detector returns an empty string `""`.
- `$send_cmnd(num, text)` - Send a command to another block. *num* is the number of the remote block, like "1234". *text* is the command string like "on".
- `set_filter(num, slot, val)` - Enable/disable a Distributor filter slot. *num* is the number of the Distributor block. *slot* is a color and is used to select one of the Distributor sides: "red", "green", "blue", and "yellow". *val* is either "on" (enable filter) or "off" (disable filter).
- `$display(num, row, text,...)` Send a text string to the display with number *num*. *row* is the display row, a value from 1 to 5. *text* is the string to be displayed. This function allows up to 3 text strings.
- `$clear_screen(num)` Clear the screen of the display with number *num*.
- `$position(num)` Returns the position '(x,y,z)' of the device with the given *num*.

Server and Terminal commands

The Server is used to store data permanently/non-volatile. It can also be used to share data between several Controllers.

- `$server_write(num, key, value)` - Store a value on the server under the key *key*. *key* must be a string. *value* can be either a number, string, boolean, nil or data structure. **But this command does not allow nested data structures**. *num* is the number of the Server, like "1234". Example: `$server_write("0123", "state", state)`
- `$server_read(num, key)` - Read a value from the server. *key* must be a string. *num* is the number of the Server, like "1234".

The Terminal can send text strings as events to the Controller. In contrast the Controller can send text strings to the terminal.

- `$get_term()` - Read a text command received from the Terminal

- `$put_term(num, text)` - Send a text string to the Terminal. *num* is the number of the Terminal.

Further commands

Messages are used to transport data between Controllers. Messages are text strings or any other data plus the sender number. Incoming messages are stored in a message queue (up to 10) and can be read one after the other.

- `$get_msg()` - Read a received message. The function returns the sender number and the message.
- `$send_msg(num, msg)` - Send a message to another Controller. *num* is the destination number.

- `$chat(text,...)` - Send yourself a chat message. This function allows up to 3 text strings.
- `$door(pos, text)` - Open/Close a door at position "pos".
  Example: `$door("123,7,-1200", "close")`.
  Hint: Use the Techage Programmer or Info Tool to easily determine the door position.

Example Scripts

Simple Counter

Very simple example with output on the Controller menu.

init() code:

```
a = 1
```

loop() code:

```
a = a + 1
$print("a = ", a)
```

Hello World

"Hello world" example with output on the Display.

init() code:

```
a = Array("Hello", "world", "of", "Minetest")

$clear_screen("0669")

for i,text in a.next() do
    $display("0669", i+2, text)
end
```

For Loop with range(from, to)

Second "Hello world" example with output on the Display, implemented by means of a for/range loop.

init() code:

```
a = Array("Hello", "world", "of", "Minetest")

$clear_screen("0669")

for i in range(1, 4) do
text = a.get(i)
$display("0669", i+2, text)
end
```

Monitoring Chest & Furnace

More realistic example to output Pusher states on the Display

init() code:

```
DISPLAY = "1234"
min = 0
```

loop() code:

```
-- call code every 60 sec
if ticks % 60 == 0 then
    -- output time in minutes
    min = min + 1
    $display(DISPLAY, 1, min, " min")

    -- Cactus chest overrun
    sts = $get_status("1034") -- read pusher status
    if sts == "blocked" then $display(DISPLAY, 2, "Cactus full") end

    -- Tree chest overrun
    sts = $get_status("1089")  -- read pusher status
    if sts == "blocked" then $display(DISPLAY, 3, "Tree full") end

    -- Furnace fuel empty
    sts = $get_status("2895")  -- read pusher status
    if sts == "standby" then $display(DISPLAY, 4, "Furnace fuel") end
end
```

Emails

For an email system you need a Central Server and a TA4 Lua Controller with Terminal per player. The Central Server serves as database for player name/block number resolution.

- Each Player needs its own Terminal and Controller. The Terminal has to be connected with the Controller
- Each Controller runs the same Lua Script, only the numbers and the owner names are different
- To send a message, enter the receiver name and the text message like `Tom: hello` into the Terminal
- The Lua script will determine the destination number and send the message to the destination Controller
- All players who should be able to take part in the email system have to be entered into the Server form

init() code:

```
$loopcycle(0)
$events(true)

-- Start: update to your conditions
TERM = "27309"
CONTROLLER = "27310"
NAME = "Tom"
SERVER = "27312"
-- End: update to your conditions

$server_write(SERVER, NAME, CONTROLLER)
$server_write(SERVER, CONTROLLER, NAME)
```

loop() code:

```
-- read from Terminal and send the message
s = $get_term()
if s then
    name,text = unpack(string.split(s, ":", false, 1))
    num = $server_read(SERVER, name)
    if num then
        $send_msg(num, text)
        $put_term(TERM, "message sent")
    end
end

-- read message and output to terminal
num,text = $get_msg()
if num then
    name = $server_read(SERVER, num)
    if name then
        $put_term(TERM, name..": "..text)
    end
end
```