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 more complex programs.

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.

This TA4 Lua Controller manual is also available as PDF:

https://github.com/joe7575/techage/blob/master/manuals/ta4_lua_controller_EN.pdf

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
- the `func` tab for the Lua functions
- the `loop` tab for the main code block
- 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 functions

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

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 "TA4 Lua Server" block has to be used.

TA4 Lua Server

The Server block is used to store data from Lua Controllers nonvolatile. It can also be used for communication purposes between several Lua Controllers. Only configured players have access to the server. Therefore, the server has a menu to enter player names.

For special Server functions, see "Server and Terminal Functions"

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 functions for the TA4 Lua Controller, see "Server and Terminal Functions"

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, Cyclic Task, 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, clean the display, or reset other blocks:

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

# reset blocks
$clear_screen("123")      -- "123" is the number of the display
$send_cmnd("2345", "off")  -- turn off the blocks with the number "2345"
```

Cyclic Task

During the continuous operational phase the `loop()` function is cyclically called. 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 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 function `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 $get_input("3456") == "on" then  -- check input from block "3456"
        -- do some action...
    end
end
```

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

Lua Controller Functions

In addition to Lua standard function the Lua Controller provides the following functions:

Controller local Functions

- `$print(text)` - Output a text string on the 'outp' tab of the controller menu. E.g.: `$print("Hello "..name)`
- `$loopcycle(seconds)` - This function allows to change the call frequency of the controller loop() function, witch is per default one second. For more info, see "Cyclic Task"
- `$events(bool)` - Enable/disable event handling. For more info, see "Events"
- `$get_ms_time()` - Returns the time with millisecond precision
- `get_gametime()` - Returns the time, in seconds, since the world was created
- `$time_as_str()` - Read the time of day (ingame) as text string in 24h format, like "18:45"
- `$time_as_num()` - Read the time of day (ingame) as integer number in 24h format, like 1845
- `$get_input(num)` - Read an 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 "456" is configured to send on/off commands to the TA4 Lua Controller with number "345".
- The TA4 Lua Controller will receive these commands as input value.
- The program on the SaferLua Controller can always read the last input value from the Player Detector with number "456" by means of:

```
sts = $get_input("456")
```

Techage Command Functions

- `$read_data(num, ident, add_data)` - Read any kind of data from another block with the given number *num*. *ident* specifies the data to be read. *add_data* is for additional data and normally not needed. The result is block dependent (see table below):

ident	returned data	comment
"state"	one of: "running", "stopped", "blocked", "standby", "fault", or "unloaded"	Techage machine state, used by many machines
"state"	one of: "red", "amber", "green", "off"	Signal Tower state
"state"	one of: "empty", "loaded", "full"	State of a chest or Sensor Chest
"fuel"	number	fuel value of a fuel consuming block
"depth"	number	Read the current depth value of a quarry block (1..80)
"load"	number	Read the load value in percent (0..100) of a tank/storage block, an accu block, of of the Signs Bot Box.
"delivered"	number	Read the current delivered power value of a generator block. A power consuming block (accu) provides a negative value
"action"	player-name, action-string	only for Sensor Chests
"stacks"	Array with up to 4 Stores with the inventory content (see example)	only for Sensor Chests
"count"	number	Read the item counter of the TA4 Item Detector block
"count"	number of items	Read the total amount of TA4 chest items. An optional number as `add_data` is used to address only on inventory slot (1..8, from left to right).
"itemstring"	item string of the given slot	Specific command for the TA4 8x2000 Chest to read the item type (technical name) of one chest slot, specified via `add_data` (1..8). Example: s = $read_data("223", "itemstring", 1)

- `$send_cmnd(num, cmnd, data)` - Send a command to another block. *num* is the number of the remote block, like "1234". *cmnd* is the command, *data* is additional data (see table below):

cmnd	data	comment
"on", "off"	nil	turn a node on/off (machine, lamp,...)
"red, "amber", "green", "off"	nil	set Signal Tower color
"port"	`<color>=on/off`	Enable/disable a Distributor filter slot.. Example: `yellow=on` colors: "red", "green", "blue", "yellow"
"text"	text string	Text to be used for the Sensor Chest menu
"reset"	nil	Reset the item counter of the TA4 Item Detector block
"pull"	item string	Start the TA4 pusher to pull/push items. Example: `default:dirt 8`
"config"	item string	Configure the TA4 pusher. Example: `wool:blue`

- `$display(num, row, text)` Send a text string to the display with number *num*. *row* is the display row, a value from 1 to 5, or 0 to add the text string at the bottom (scroll screen mode). *text* is the string to be displayed. If the first char of the string is a blank, the text will be horizontally centered.
- `$clear_screen(num)` Clear the screen of the display with number *num*.
- `$position(num)` Returns the position as string "'(x,y,z)" of the device with the given *num*.

Server and Terminal Functions

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

- `$server_write(num, key, value)` - Store a value on the server under the key *key*. *key* must be a string. *value* can be either a number, string, boolean, nil or data structure. **This function does not allow nested data structures**. *num* is the number of the Server.
  Example: `$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 Functions

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

- `$get_msg()` - Read a received message. The function returns the sender number and the message. (see example "Emails")
- `$send_msg(num, msg)` - Send a message to another Controller. *num* is the destination number. (see example "Emails")
- `$chat(text)` - Send yourself a chat message. *text* is a text string.
- `$door(pos, text)` - Open/Close a door at position "pos".
  Example: `$door("123,7,-1200", "close")`.
  Hint: Use the Techage Info Tool to determine the door position.
- `$item_description("default:apple")` Get the description (item name) for a specified itemstring, e. g. determined via the TA4 8x2000 Chest command `itemstring`: `str = $read_data("223", "itemstring", 1)` `descr = $item_description(str)`

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, 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, text)
end
```

Monitoring Chest & Furnace

More realistic example to read Pusher states and output them on a display:

init() code:

```
DISPLAY = "1234"  -- adapt this to your display number
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 = $read_data("1034", "state") -- read pusher status
    if sts == "blocked" then $display(DISPLAY, 2, "Cactus full") end

    -- Tree chest overrun
    sts = $read_data("1065", "state")  -- read pusher status
    if sts == "blocked" then $display(DISPLAY, 3, "Tree full") end

    -- Furnace fuel empty
    sts = $read_data("1544", "state")  -- read pusher status
    if sts == "standby" then $display(DISPLAY, 4, "Furnace fuel") end
end
```

Simple Calculator

A simple calculator (adds entered numbers) by means of a Lua Controller and a Terminal.

init() code:

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

TERM = "360" -- terminal number, to be adapted!
sum = 0
$put_term(TERM, "sum = "..sum)
```

loop() code:

```
s = $get_term() -- read text from terminal
if s then
    val = tonumber(s) or 0  -- convert to number
    sum = sum + val
    text = string.format("+%d = %d", val, sum) -- format output string
    $put_term(TERM, text)  -- output to terminal
end
```

Welcome Display

In addition to the controller, you also need a player detector and a display. When the Player Detector detects a player the player name is shown on the display:

init() code:

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

SENSOR = "365"   -- player detector number, to be adapted!
DISPLAY = "367"  -- display number, to be adapted!

$clear_screen(DISPLAY)
```

loop() code:

```
if event then
    name = $read_data(SENSOR, "name")
    if name == "" then -- no player arround
        $clear_screen(DISPLAY)
    else
        $display(DISPLAY, 2, " Welcome")
        $display(DISPLAY, 3, " "..name)
    end
end
```

Sensor Chest

The following example shows the functions/commands to be used with the Sensor Chest:

init() code:

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

SENSOR = "372"   -- sensor chest number, to be adapted!

$send_cmnd(SENSOR, "text", "press both buttons and\nput something into the
chest")
```

loop() code:

```
if event and $get_input(SENSOR) == "on" then
    -- read inventory state
    state = $read_data(SENSOR, "state")
    $print("state: "..state)
    -- read player name and action
    name, action = $read_data(SENSOR, "action")
    $print("action"..": "..name.." "..action)
    -- read inventory content
    stacks = $read_data(SENSOR, "stacks")
    for i,stack in stacks.next() do
        $print("stack: "..stack.get("name").."  "..stack.get("count"))
    end
    $print("")
end
```

Emails

For an email system you need a TA4 Lua Server and a TA4 Lua Controller with Terminal per player. The TA4 Lua 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 = "360"
CONTROLLER = "359"
NAME = "Tom"
SERVER = "363"
-- End: update to your conditions

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