PULSAR PHOTONICS

# Photonic Elements

This document described the scripting functionality of the software system “Photonic Elements” (PE) in the version 2.9.1 (January 2015). You can find the current version of this document on the Pulsar website (www.pulsar-photonics.de)

# Scripting fundamentals

Following crash course is an extremely condensed extract from the official Lua reference manual. For more details refer to the Lua website and to the numerous examples contained in the demo scenes.

## LEXICAL CONVENTIONS

Lua is a case sensitive language. "and", "And" or "AND" are not the same. This is especially important for variables because “offset” and “Offset” are two completely different variables. There is a short list of reserved keywords, which cannot be used as a name for a variable !!

Following are **Lua keywords**:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

Following strings denote other tokens:

```
+     -     *     /     %     ^     #
==    ~=    <=    >=    <     >     =
(     )     {     }     [     ]
;     :     ,     .     ..    ...
```

Comments:

Comments are not executed are used to document the functionaly of a script

```
-- check if both values are the same
if (value1 == value2) then
  print('value1 and value2 are same!')
end
```

## VALUES AND TYPE

Lua is a dynamically typed language which means that variables do not have types and can be changed during the runtime of the script. There are 5 basic types in Lua:

| | |
|---|---|
| **nil** | type of the value nil whose main property is to be different from any other value. It usually represents the absence of a useful value |
| **boolean** | values false and true (both nil and false make a condition false;<br>any other value makes it true) |
| **number** | real numbers |
| **string** | arrays of characters (strings may contain any 8-bit character, including embedded zeros) |
| **table** | arrays that can hold values of any type except nil |

## VARIABLES

- There are 3 kinds of variables: global variables, local variables and table fields. Any variable is assumed to be global unless explicitly declared as local
- Before the first assignment to a variable, its value is nil
- Square brackets are used to index a table (e.g. value=table[x]). The first value in a table is at position 1

## STATEMENTS

Lua allows multiple assignments. The syntax for assignments defines a list of variables on the left side and a list of expressions on the right side. The elements in both lists are separated by commas:

```
x,y,z = myTable[1],myTable[2],myTable[3]
```

Relational operators (always result in **false** or **true**)

```
==      equality
~=      negation of equality
<       smaller than
>       bigger than
<=      smaller or equal than
>=      bigger or equal than
```

**If** control structure (by example):

```
if (var1 == var2) then
    print('value1 and value2 are same!')
end
```

**For** control structure (by example):

```
for i = 1,8,2 do                -- count from 1 to 8 with increments of 2
    print(i)
end
```

**While** control structure (by example):

```
i = 0
while i ~= 4 do
    i = i + 1
end
```

**Repeat** control structure (by example):

```
i = 0
repeat
    i = i + 1
until i==4
```

**Table** operations (by example):

```
myTable = {'firstValue',2,3}     -- builds a table with 3 values
print(myTable[1])                -- prints the first element in the table
```

```
table.insert(myTable,4)          -- appends the number 4 to the table
```

Concatenation (by example):

```
A = 'hello'
b = 'world'
c = a..b       -- c contains 'hello world'
```

Length operator #:

```
stringLength = #'hello world'    -- 11
tableSize    = #{1,2,3,4,5}      --  5
```

# API reference

The PE extensions for the lua scripting language are grouped into several sections.

**TCP – Tool center point**

These command grant access to the coordinate engine embedded into PE. You can save and load different coordinates on the workpiece or switch tools between laser/camera/sensors.

**JOB – Laser job interface**

With the job interface you can open 3D jobs exported from Rhino. Additionally you can load laser parameter and change specific values. The integrated job and pattern generator can be used to create job files programmatically on the fly.

**CAM – Camera hardware interface**

For inspection and work piece alignment the integrated camera module CM1 can be used in the PE software interface. Inside the scripting interface the camera can be used to take images automatically for later inspection and documentation.

**UI – User Interface**

These are special Functions to interact with the User-interface

## *TCP*

### changeTool *(tool)*

Move the new tool to the exact same position the machine is currently pointing at.

| | | |
|---|---|---|
| | tool | string |

### saveCoordinate *(name)*

The current position is internally saved under the name “name”. This can be just for later restoring. It is used if the machine is moved between different steps.

| | | |
|---|---|---|
| | name | string |

### restore *(name)*

A previously save coordinate will be restored and the machine moved to this exact position. The internal memory is not deleted, so this command can be executed multiple times.

| | | |
|---|---|---|
| | name | string |

### moveRelative *(dx,dy,dz)*

| | | |
|---|---|---|
| The axis of the machine will be moved relative to the current position.<br>All values are expected to be in mm. | | |
| | dx | float |
| | dy | float |
| | dz | float |

| | | | |
|---|---|---|---|
| **turnRelative** *(dc)* | | | |
| | The c - axis (if implemented) of the machine will be turn relative to the.<br>All values are expected to be in mm. | | |
| | dc | float | mm |

## *JOB*

| | | |
|---|---|---|
| **load** *(fileName)* | | |
| The file with the name *fileName* will be loaded from the job-directory. If a set of laser-parameter with the same name is available, this will be loaded too. | | |
| | filename | string |

| | | |
|---|---|---|
| **setLayerType** *(type)* | | |
| A job file can contain hatch and contour layers. With this command you can select which type will be processed. The default value is “all” | | |
| | type | string |

| | | |
|---|---|---|
| **loadParameters** *(filename)* | | |
| Open the file “filename”, which has to be in the job directory. | | |
| | filename | string |

| | | |
|---|---|---|
| **changeParameter** *(parameter, value)* | | |

Changes one value of the current set of parameters. Valid parameters are:

- powerSetting % 0-100
- markSpeed mm/s
- jumpSpeed mm/s
- delayOn µs
- delayOff µs
- ~~waveForm x only for SPI lasers~~
- burstTime µs time for single points
- ~~frequency kHz only for SPI lasers~~
- repeats x
- optionAxis “none” / “distance” / “number”

| | | |
|---|---|---|
| • axisDistance µm<br>• axisNumber x | | |
| | parameter | string |
| | Value | float |

**execute** *()*

The current job will be executed with the current set of laser parameters. If no job file was loaded / created, an error will be raised.
The execution will process all selected layer types of the current job stack.

## *Light*

**changeIntensity** *(intensity)*

Changes the intensity of the coaxial light in the CM1 camera module.

| | | |
|---|---|---|
| | intensity | integer |

**toggleLight** *(flag)*

If no parameter is given, the status of the light will be toggled. Otherwise the value of “flag” will be interpreted as boolean value.

| | | |
|---|---|---|
| | flag [opt] | integer |

**togglePointer** *([flag])*

If no parameter is given, the status of the laser pointer will be toggled. Otherwise the value of “flag” will be interpreted as boolean value.

| | | |
|---|---|---|
| | flag | integer |

## *Camera*

**executeAutofocus** *([range])*

The optical autofocus finder will be executed. The default value for the max. range will be 5mm in each direction.

| | | |
|---|---|---|
| | range [opt] | integer |

**saveImage** *(filename)*

The camera will snap an image an store the results as “png” file on the computer.

| | | |
|---|---|---|
| | filename | string |

## *UI*

| **Stop()** | | | |
|---|---|---|---|
| reacting to the external “Stop” -Button. Useful to break loops in critical situations | | | |
| return (0 or 1) | integer | Example:<br>for i =0,N-1 do<br>if UI.Stop()==0 then<br>break<br>end<br>-- do something<br>end | 0 <=> Button pushed |

| **wait***(time)* | | | |
|---|---|---|---|
| The Process is waiting until the time is over | | | |
| | time | integer | Time in Sec |

# Example scripts

The basic lua scripting commands are embedded into the script engine of Photonic Elements (“PE”). For access to the hardware elements and some additional functions for

### FOCUS FINDER

This script is used for finding the exact focus position of the focusing lens. The approach is to structure multiple lines in different focus positions. Two laser jobs are created on the fly: one for marking the processing area and one for the single line. With the settings variables in the begin of the script, the parameters can be adjusted.

```
-- Settings
deltaz = 0.1     -- focus shift per step [mm]
deltax = 5       -- distance between lines [mm]
nLines = 11      -- number of lines

settingFile = "focus.pep"


-- generate jobs
PAT.newJob()
PAT.addLine(0,-3,0,3)
jobLine = PAT.exportJob()

PAT.newJob()
PAT.addRectangleAt(0,0,(nLines+2)*deltax,7, hatch=("none","none"))
jobBorder = PAT.exportJOB()


-- mark border
JOB.loadFile(jobBorder)
JOB.loadParamter(settingFile)
JOB.execute()

-- save coordinate
TCP.setWorkpieceZero()
TCP.storeCoordinate("origin")

-- focus search
JOB.loadFile(jobLine)
JOB.loadParamter(settingFile)

for i = 0,nLines do
  offset = i - (nLines-1)/2
  x = deltax * offset
  z = deltaz * offset
  TCP.moveWorkpieceTo(x,0,z)
  JOB.execute()
end

-- move back
TCP.restoreCoordinate("origin")
```

## JOB STITCHER

In many applications the surface of the workpiece has to be structured with a periodic pattern. Because of the limited size of the laser scanfield, the job has to be splitted into multiple fields.

```
-- Settings
Nx = 10      -- number of jobs in x direction
Ny = 10      -- number of jobs in x direction
sx =  3      -- x size of laser job [mm]
sy =  3      -- x size of laser job [mm]

jobFile = "stitcher.pph"


-- load job border
JOB.loadFile(jobFile)

-- save coordinate
TCP.setWorkpieceZero()
TCP.storeCoordinate("origin")

for x = 0,Nx do
  for y = 0,Ny do
    TCP.moveWorkpieceTo(x*sx, y*sy)
    JOB.execute()
  end
end

-- move back
TCP.restoreCoordinate("origin")
```