the barracuda manual

https://github.com/robitex/barracuda

Roberto Giacomelli

email: giaconet.mailbox@gmail.com

Date 2020-04-04 — Version v0.0.11 — Beta stage

Abstract

Welcome to the barracuda software project devoted to barcode printing.

This manual shows you how to print barcodes in your TEX documents and how to export such graphic content to an external file, using barracuda.

barracuda is written in Lua and is free software released under the GPL 2 License.

Contents

1 Getting started

1.1 Introduction

Barcode symbols are usually a sequence of vertical lines representing encoded data that can be retrived with special laser scanner or more simpler with a smartphone running dedicated apps. Almost every store item has a label with a printed barcode for automatic identification purpose.

So far, `barracuda` supported symbologies are as the following:

- Code 39,
- Code 128,
- EAN family (ISBN, ISSN, EAN 8, EAN 13, and the add-ons EAN 2 and EAN 5),
- ITF 2of5, interleaved Two of Five (ITF14, i2of5 in general).

The package provides different output graphic format. At the moment they are:

- PDF Portable Document Format (a modern TeX engine is required),
- SVG Scalable Vector Graphic.

The name `barracuda` is an assonance to the name Barcode. I started the project back in 2016 for getting barcode in my TeX generated PDF documents, studying the LuaTeX technology such as direct *pdfliteral node* creation.

At the moment `barracuda` is in *beta* stage. In this phase the Lua API can change respect to the result of development research.

1.2 Manual Content

The manual is divided into five part. In part 1.1 introduces the package and gives to the user a proof of concept to how to use it. The next parts present detailed information about option parameter of each barcode symbology and methods description to change the *module* width of a EAN-13 barcode. It's also detailed how the Lua code works internally and how to implement a barcode symbology not already included in the package.

The manual plan is:

Part 1: Getting started

- general introduction → 2
- print your first barcode → 3
- installing `barracuda` on your system → 5

Part 2: LaTeX packages

- `barracuda` LaTeX package → 5

Part 3: Barcode Reference

- barcode symbologies reference → 5

Part 4: Developer zone

- Lua framework description → 6
- encoder identification rule → 6
- API reference → 6
- `ga` specification → 6

Part 5: Real examples

- working example and use cases → 15

1.3 Required knowledge and useful resources

`barracuda` is a Lua package that can be executed by any Lua interpreter. To use it, it's necessary a minimal knowledge of Lua programming language and a certain ability with the terminal of your computer system in order to run command line task or make software installation.

It's also possible to run `barracuda` directly within a TeX source file, and compile it with a suitable typesetting engine like LuaTeX. In this case a minimal TeX system knowledge is required. As an example of this workflow you simply can look to this manual because itself is typesetted with LuaLaTeX, running `barracuda` to include barcodes as a vector graphic object.

A third way is to use the LaTeX package `barracuda.sty` with its high level macros. A minimal knowledge of the LaTeX format is obviously required.

Here is a collection of useful learning resources:

Lua: to learn Lua the main reference is the book called PIL that stands for Programming in Lua from one of the language's Author Roberto Ierusalimschy.

LuaTeX: the typesetting engine manual can be opened running the `texdoc` utility in a terminal session, typing `luatex` as the sole argument.

1.4 Running Barracuda

The starting point to work with `barracuda` is always a plain text file with some code processed by a command line program with a Lua interpreter.

The paradigm of `barracuda` is the Object Oriented Programming. Generally speaking every object must be created with a function called *costructor* and every action must be run calling an object *method*.

In this section you'll take a taste of `barracuda` coding in three different execution context: a Lua script, a LuaTeX document and a LaTeX source file using the macro package `barracuda.sty` providing an high level interface to Lua code.

High level package like `barracuda.sty` make to write Lua code unnecessary. It will be always possible return to Lua code in order to resolve complex barcode requirements.

1.4.1 A Lua script

As a practical example to produce an EAN 13 barcode, open a text editor of your choice on an empty file and save it as `first-run.lua` with the content of the following two lines of code:

```
local barracuda = require "barracuda"
barracuda:save("ean-13", "8006194056290", "my_barcode", "svg")
```

What you have done is to write a *script*. If you have installed a Lua interpreter along with `barracuda`, open a terminal and run it with the command:

```
$ lua first-run.lua
```

You will see in the same directory of your script, appearing a new file called `my_barcode.svg` with the drawing:

Coming back to the script first of all, it's necessary to load the library `barracuda` with the standard Lua function `require()` that returns an object–more precisely a reference to a table where are stored all the package machinery.

With the second line of code, an EAN 13 barcode is saved as `my_barcode.svg` using the method `save()` of the `barracuda` object. The `save()` method takes in order the barcode symbology identifier called *treename*, an argument as a string or as a whole number that represents data to be encoded, the output file name and the optional output format. With a fifth optional argument we can pass options to the barcode encoder as a Lua table.

Each encoder has an own identifier called treename explained at section 4.2. In short, in `barracuda` we can build more encoders of the same symbology with different parameters.

1.4.2 A LuaTEX source file

`barracuda` can also runs with LuaTEX and any others Lua powered TEX engines. The source file is a bit difference respect to the previuos script: the Lua code lives inside the argument of a `\directlua` primitive, moreover we must use an horizontal box register as output destination.

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
        local require "barracuda"
        barracuda:hbox("ean-13", "8006194056290", "mybox")
}\leavevmode\box\mybox
\bye
```

The method `hbox()` works only with LuaTEX. It takes three[1] arguments: encoder *treename*, encoding data as a string, the TEX horizontal box name.

1.4.3 A LuaLATEX source file

LATEX working minimal example would be:

```
% !TeX program = LuaLaTeX
\documentclass{article}
\usepackage{barracuda}
\begin{document}
\barracuda{ean-13}{8006194056290}
\end{document}
```

1.5 A more deep look

`barracuda` is designed to be modular and flexible. For example it is possible to draw different barcodes on the same canvas or tune barcode parameters.

The main workflow to draw a barcode object reveals more details on internal structure. In fact, to draw an EAN 13 barcode we must do at least the following steps:

1. load the library,
2. get a reference to the `Barcode` abstract class,
3. build an `ean` encoder of the variant `13`,
4. build an EAN 13 symbol passing data to a costructor,
5. get a reference to a new canvas object,
6. draw barcode on the canvas object,
7. get a reference of the driver object,
8. print the graphic material saving an external `svg` file.

Following that step by step procedure the corresponding code is translated in the next listing:

```
-- lua script
local barracuda = require "barracuda" -- step 1
local barcode = barracuda:barcode() -- step 2

local ean13, err_enc = barcode:new_encoder("ean-13") -- step 3
assert(ean13, err_enc)
```

[1]A fourth argment is optional as a table with user defined barcode parameters.

```
local symb, err_symb = ean13:from_string("8006194056290") -- step 4
assert(symb, err_symb)

local canvas = barracuda:new_canvas() -- step 5
symb:draw(canvas) -- step 6

local driver = barracuda:get_driver() -- step 7
local ok, err_out = driver:save("svg", canvas, "my_barcode") -- step 8
assert(ok, err_out)
```

Late the manual will give objects and methods references at section 4.3.

1.6 Installing

1.6.1 Installing for Lua

Manually copy `src` folder content to a suitable directory of your system that is reachable to the system Lua interpreter.

1.6.2 Installing for TeX Live

If you have TeX Live installed from CTAN or from DVD TeX Collection, before any modification to your system check if the package is already installed looking for *installed* key in the output of the command:

```
$ tlmgr show barracuda
```

If ‘barracuda‘ is not present, run the command:

```
$ tlmgr install barracuda
```

If you have installed TeX Live via Linux OS repository try your distribution’s package management system running a software update.

It’s also possible to install the package manually:

1. Grab the sources from CTAN or `https://github.com/robitex/barracuda`.
2. Unzip it at the root of one or your TDS trees (local or personal).
3. You may need to update some filename database after this, see your TeX distribution’s manual for details.

2 Barracuda LaTeX Package

The LaTeX package delivered with `barracuda` is still under an early stage of development. The only macro available is `\barracuda[option]{encoder}{data}`. A simple example is the following source file for LuaLaTeX:

```
% !TeX program = LuaLaTeX
\documentclass{article}
\usepackage{barracuda}
\begin{document}
\leavevmode
\barracuda{code39}{123ABC}\\
\barracuda{code128}{123ABC}
\end{document}
```

Every macro `\barracuda` typesets a barcode symbol with the encoder defined in the first argument, encoding data defined by the second.

3 Barcode Reference

For each barcode symbologies this section reports parameters and optional methods of it.

3.1 Code39

`Code39` is one of the oldest symbologies ever created. It does not include any checksum digit and the only encodable characters are digits, uppercase letters and a few symbol like `+` or `$`.

4 Developer zone

4.1 The Barracuda Framework

The `barracuda` package framework consists in indipendent modules: a barcode class hierarchy encoding a text into a barcode symbology; a geometrical library called `libgeo` modelling several graphic objects; an encoding library for the `ga` format (graphic assembler) and several driver to *print* a `ga` stream into a file or in a TEX hbox register.

To implement a barcode encoder you have to write a component called *encoder* defining every parameters and implementing the encoder builder, while a driver must understand ga opcode stream and print the corresponding graphic object.

Every barcode encoder come with a set of parameters, some of them can be reserved and can't be edit after the encoder was build. So, you can create many instances of the same encoder for a single barcode type, with its own parameter set.

The basic idea is getting faster encoders, for which the user may set up paramenters at any level: barcode abstract class, encoder globally, down to a single symbol object.

The Barcode class is completely indipendent from the ouput driver and viceversa.

4.2 Encoder Treename

In `barracuda` in order to draw a barcode symbol it's necessary to create an `Encoder` object

4.3 API reference of Lua modules

TODO

4.4 `ga` specification

This section defines and explains with code examples the `ga` instruction stream. `ga` stands for *graphic assembler*, a sort of essential language that describes geometrical object like lines and rectangles mainly for a barcode drawing library on a cartesian plane (O, x, y).

The major goal of any `barracuda` encoder is to create the `ga` stream corresponding to a vector drawing of a barcode symbol.

In details, a `ga` stream is a numeric sequence that like a program defines what must be draw. It is not a fully binary sequence–which is a byte stream and ideally is what a `ga` stream would really be–but a sequence of integers or floating point numbers.

In Lua this is very easy to implement. Simply append a numeric value to a table that behave as an array. Anyway `ga` must be basically a binary format almost ready to be sent or received by means of a network channel.

In the Backus–Naur form a valid `ga` stream grammar is described by the following code:

```
<valid ga stream> ::= <instructions>
<instructions> ::= <instruction>
                 | <instruction> <instructions>
<instruction> ::= <opcode>
                | <opcode> <operands>

<opcode> ::= <state>
           | <object>
           | <func>
```

```
<state> ::= 1 .. 31; graphic properties
<object> ::= 32 .. 239; graphic objects
<func> ::= 240 .. 255; functions

<operands> ::= <operand>
             | <operand> <operands>
<operand> ::= <len>
            | <coord>
            | <qty>
            | <char seq>
            | <enum>
            | <abs>
            | <points>
            | <bars>

<len> ::= f64; unit measure scaled point sp = 1/65536pt
<coord> ::= f64; unit measure scaled point sp = 1/65536pt
<qty> ::= u64
<char seq> ::= <chars> 0
<chars> ::= <char>
          | <char> <chars>
<char> ::= u64
<enum> ::= u8
<abs> ::= f64
<points> ::= <point>
           | <point> <points>
<point> ::= <x coord> <y coord>
<x coord> ::= <coord>
<y coord> ::= <coord>
<bars> ::= <bar>
         | <bar> <bars>
<bar> := <coord> <len>

; u8 unsigned 8 bit integer
; u64 unsigned 64 bit integer
; f64 floating point 64 bit number
```

Every `<instruction>` changes the graphic state, for instance the current line width, or defines a graphic object, depending on the `opcode` value. Every coordinate or dimension must be expressed as *scaled point*, the basic unit of measure of TEX equivalent to $1/65536\,\mathrm{pt}$.

For example, the `opcode` for the `<linewidth>` operation is 1, while for the `<hline>` operation is 33. An horizontal line 6pt width from the point (0pt, 0pt) to the point (32pt, 0pt) is represented by this ga stream:

```
1 393216 33 0 2097152 0
```

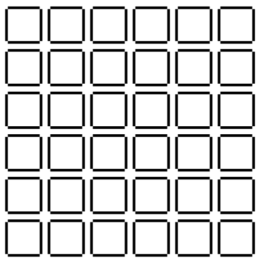
Introducing `<mnemonic code>` in `<opcode>` place and separate the operations in a multiline fashion, the same sequence become more readable and more similar to an assembler listing:

```
<linewidth> 393216
<hline> 0 2097152 0
```

To prove and visualize the meaning of the stream, we can simply use the native graphic driver of `barracuda` compiling this LuaTEX source file:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local ga = {1, 393216, 33, 0, 2097152, 0}
```

Figure 1: A drawing example that shows how to manually set up the bounding box of the figure, keeping automatic computation disabled for more fast stream processing.



```
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(ga, "mybox")
}\leavevmode\box\mybox
\bye
```

The result is:

A more abstract way to write a ga stream consists in the gacanvas class of the module libgeo. Every operation has been mapped to a method named `encode_<mnemonic_opcode>()`:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local canvas = barracuda:new_canvas()
    local pt = 65536
    canvas:encode_linewidth(6*pt)
    canvas:encode_hline(0, 32*pt, 0)
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(canvas, "mybox")
    tex.print("[")
    for _, n in ipairs(canvas:get_stream()) do
        tex.print(tostring(n))
    end
    tex.print("]")
} results in \box\mybox
\bye
```

The stream is printed beside the drawing in the output PDF file. Therefore the same ga stream can also generate a different output, for instance a SVG file. For this purpose execute the `save()` method of the `Driver` class (the drawing is showed in figure 1):

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local canvas = barracuda:new_canvas()
    local pt = 65536
    local side = 16*pt
    local s = side/2 - 1.5*pt
    local l = side/2 - 2*pt
    local dim = 5
    canvas:encode_linewidth(1*pt)
    canvas:encode_disable_bbox()
    for c = 0, dim do
        for r = 0, dim do
            local x, y = c*side, r*side
            canvas:encode_hline(x-l, x+l, y-s)
            canvas:encode_hline(x-l, x+l, y+s)
            canvas:encode_vline(y-l, y+l, x-s)
            canvas:encode_vline(y-l, y+l, x+s)
        end
    end
    local b1 = -s - 0.5*pt
    local b2 = dim*side + s + 0.5*pt
    canvas:encode_set_bbox(b1, b1, b2, b2)
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(canvas, "mybox")
    drv:save("svg", canvas, "grid")
}\leavevmode\box\mybox
\bye
```

An automatic process updates the bounding box of the figure meanwhile the stream is read instruction after instruction. The `<disable_bbox>` operation produce a more fast execution and the figure will get the bounding box computed until that instruction. The `<set_bbox>` operation imposes a minimal bounding box comparing to the current figure dimensions.

The initial bounding box is simply an empty figure. As a conseguence, different strategies can be used to optimize runtime execution, such as in the previuos code example, where bounding box is always disabled and it is set up at the last `canvas` method call. More often than not, we know the bounding box of the barcode symbol including quiet zones.

Every encoding method of `gaCanvas` class gives two output result: a boolean value called `ok` plus an error `err`. If `ok` is `true` then `err` is `nil` and, viceversa, when `ok` is `false` then `err` is a string describing the error.

The error management is a responsability of the caller. For instance, if we decide to stop the execution this format is perfectly suitable for the Lua function `assert()`, otherwise we can explicity check the output pair:

```
local pt = 65536
assert(canvas:encode_linewidth(6*pt)) --> true, nil
local ok, err = canvas:encode_hline(nil, 32*pt, 0)
-- ok = false
-- err = "[ArgErr] 'x1' number expected"
```

4.4.1 ga reference

Properties of the graphic state

OpCode	Mnemonic key	Graphic property	Operands
1	`linewidth`	Line width	`w <len>`
2	`linecap`	Line cap style	`e <enum>`
			`0: Butt cap`
			`1: Round cap`
			`2: Projecting square cap`
3	`linejoin`	Line join style	`e <enum>`
			`0: Miter join`
			`1: Round join`
			`2: Bevel join`
5	`dash_pattern`	Dash pattern line style	`p <len> n <qty> [bi <len>]+`
			`p: phase lenght`
			`n: number of array element`
			`bi: dash array lenght`
6	`reset_pattern`	Set the solid line style	-
29	`enable_bbox`	Compute bounding box	-
30	`disable_bbox`	Do not compute bounding box	-
31	`set_bbox`	Overlap current bounding box	`x1 y1 <point> x2 y2 <point>`

Lines

OpCode	Mnemonic key	Graphic object	Operands
32	line	Line	x1 y1 <point> x2 y2 <point>
33	hline	Horizontal line	x1 x2 <point> y <coord>
34	vline	Vertical line	y1 y2 <point> x <coord>

Group of bars

OpCode	Mnemonic key	Graphic object	Operands
36	vbar	Vertical bars	y1 <coord> y2 <coord> b <qty> [xi wi <bars>]+ y1: bottom y-coord y2: top y-coord b: number of bars xi: axis x-coord of bars number i wi: width of bars number i
37	hbar	Horizontal bars	x1 <coord> x2 <coord> b <qty> [yi wi <bars>]+ unimplemented
38	polyline	Opened polyline	n <qty> [xi yi <points>]+ n: number of points xi: x-coord of point i yi: y-coord of point i
39	c_polyline	Closed polyline	n <qty> [xi yi <points>] unimplemented

Rectangles

OpCode	Mnemonic key	Graphic object	Operands
48	rect	Rectangle	x1 y1 <point> x2 y2 <point>
49	f_rect	Filled rectangle	x1 y1 <point> x2 y2 <point> unimplemented
50	rect_size	Rectangle	x1 y1 <point> w <len> h <len> unimplemented
51	f_rect_size	Filled rectangle	x1 y1 <point> w <len> h <len> unimplemented

Text

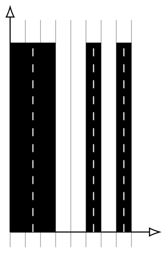
OpCode	Mnemonic key	Graphic object/Operands
130	text	A text with several glyphs ax <abs> ay <abs> xpos ypos <point> [c <chars>]+
131	text_xspaced	A text with glyphs equally spaced on its vertical axis x1 <coord> xgap <len> ay <abs> ypos <coord> [c <chars>]+
132	text_xwidth	Glyphs equally spaced on vertical axis between two x coordinates ay <abs> x1 <coord> x2 <coord> y <coord> c <chars>
140	_text_group	Texts on the same baseline ay <abs> y <coord> n <qty> [xi <coord> ai <abs> ci <chars>]+ unimplemented

4.5 `Vbar` class

This section show you how to draw a group of vertical lines, the main component of every 1D barcode symbol. In the `barracuda` jargon a group of vertical lines is called `Vbar` and is defined by a flat array of pair numbers sequence: the first one is the x-coordinate of the bar while the second is its width.

For instance, consider a `Vbar` of three bars for which width is a multiple of the fixed lenght called `mod`, defined by the array and fugure showed below:

```
-- {     x1,    w1,      x2,    w2,     x3,     w3}
   {1.5*mod, 3*mod, 5.5*mod, 1*mod, 7.5*mod, 1*mod}
```



For clearness, to the drawing were added gray vertical grid stepping one module and white dashed lines at every bar axis.

Spaces between bars can be seen as white bars. In fact, an integer number can represents the sequence of black and white bars with the rule that the single digit is the width module multiplier. So, the previous `Vbar` can be defined by 32111 with module equals to 2 mm.

The class `Vbar` of module `libgeo` has several costructors one of which is `from_int()`. Its arguments are the multipler integer `ngen`, the module lenght `mod` and the optional boolean flag `is_bar`, true if the first bar is black (default to true):

```
b = Vbar:from_int(32111, 2*mm)
```

A `Vbar` object has a local axis x and is unbounded. Constructors place the axis origin at the left of the first bar. Bars are infinite vertical straight lines. In order to draw a `Vbar` addition information must be passed to `encode_vbar()` method of the `gaCanvas` class: the global position of the local origin x_0, and the bottom and top limit y_1 y_2:

```
canvas:encode_vbar(ovbar, x0, y1, y2)
```

The following listing is the complete source code to draw the `Vbar` taken as example in this section:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local Vbar = barracuda:libgeo().Vbar
    local drv = barracuda:get_driver()
    local mm = drv.mm
    local b = Vbar:from_int(32111, 2*mm)
    local canvas = barracuda:new_canvas()
    canvas:encode_vbar(b, 0, 0, 25*mm)
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

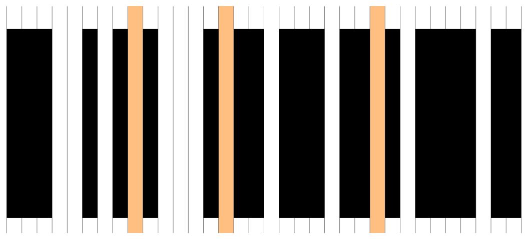
4.5.1 `Vbar` class arithmetic

Can two `Vbar` objects be added? Yes, they can! And also with numbers. Thanks to metamethod and metatable feature of Lua, `libgeo` module can provide arithmetic for `Vbar`s. More in detail, to add two `Vbar`s deploy them side by side while to add a number put a distance between the previous or the next object, depending on the order of addends.

Anyway, every sum creates or modifies a `VbarQueue` object that can be encoded in a `ga` stream with the method `encode_vbar_queue()`. The method arguments' are the same needed to encode a `Vbar`: an axis position x_0 and the two y-coordinates bound y_1 and y_2.

A `VbarQueue` code example is the following (in the drawing some graphical aid has been added: a vertical grid marks the module wide step and light colored bars mark the space added between two `Vbar`s):

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local Vbar = barracuda:libgeo().Vbar
    local drv = barracuda:get_driver()
    local mm = drv.mm
    local mod = 2 * mm
    local queue = Vbar:from_int(32111, mod)
    for _, ngen in ipairs {131, 21312, 11412} do
        queue = queue + mod + Vbar:from_int(ngen, mod)
    end
    local canvas = barracuda:new_canvas()
    canvas:encode_vbar_queue(queue, 0, 0, 25*mm)
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```



4.6 ga code examples

To support a better understanding and provide some ga stream examples, this section discusses some code examples, each of which must be compiled with LuaTEX.

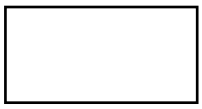
4.6.1 Example number 1

Suppose we want to draw a simple rectangle. In the ga reference at section 4.4.1 there is a dedicated instruction `<rect>`. Let's give it a try:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local pt = 65536
    local side = 36*pt
    local ga = {48, 0, 0, 2*side, side}
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(ga, "mybox")
}\leavevmode\box\mybox
\bye
```

Dealing with low level ga stream is not necessary. We can use a gaCanvas object:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local pt = 65536
    local side = 36*pt
    local canvas = barracuda:new_canvas()
    canvas:encode_rect(0, 0, 2*side, side)
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

In both cases the drawing looks like: 

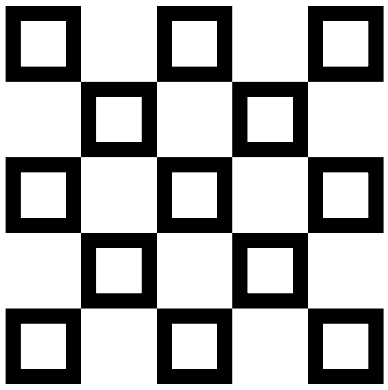
4.6.2 Example number 2

A more complex drawing is a chessboard. Let's begin to draw a single cell with a square 1cm wide:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local drv = barracuda:get_driver()
    local mm = drv.mm
    local s, t = 10*mm, 2*mm
    local canvas = barracuda:new_canvas()
    canvas:encode_linewidth(t)
    canvas:encode_rect(t/2, t/2, s-t/2, s-t/2)
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

and repeat the game for the chess grid:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local drv = barracuda:get_driver()
    local mm = drv.mm
    local s, t = 10*mm, 2*mm
    local canvas = barracuda:new_canvas()
    canvas:encode_linewidth(t)
    for row = 1, 5 do
        for col = 1, 5 do
            local l = (row + col)/2
            if l == math.floor(l) then
                local x = (col - 1) * s
                local y = (row - 1) *s
                canvas:encode_rect(x+t/2, y+t/2, x+s-t/2, y+s-t/2)
            end
        end
    end
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```



The final drawing is showed here:

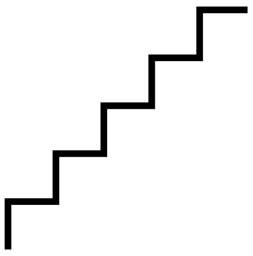
4.6.3 Example number 3

A drawing of a zig zag staircase can be represented by a `ga` stream with a `<polyline>` operation. The `gaCanvas` method we have to call is `encode_polyline()` that accept a Lua table as a flat structure with the coordinates of every point of the polyline:

```
{ x1, y1, x2, y2, ..., xn, yn }
```

It is what we do with this code:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local pt = 65536
    local side = 18*pt
    local dim = 5
    local x, y = 0, 0
    local point = {x, y}
    local i = 3
    for _ = 1, dim do
        y = y + side
        point[i] = x; i = i + 1
        point[i] = y; i = i + 1
        x = x + side
        point[i] = x; i = i + 1
        point[i] = y; i = i + 1
    end
    local canvas = barracuda:new_canvas()
    canvas:encode_linewidth(2.5*pt)
    canvas:encode_polyline(point)
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

The result is: 

A feature of `encode_<opcode>` methods is a *polymorfic* behaviour. They accept different types for own arguments. `encode_polyline` is not an exception: it accept also a `Polyline` object of the `libgeo` module instead of a flat array of coordinates. This feature is showed by the following code that re-implement the previuos version:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local pt = 65536
    local side = 18*pt
    local dim = 5
    local Polyline = barracuda:libgeo().Polyline
    local pl = Polyline:new()
    pl:add_point(0, 0)
    for _ = 1, dim do
        pl:add_relpoint(0, side)
        pl:add_relpoint(side, 0)
    end
    local canvas = barracuda:new_canvas()
    canvas:encode_linewidth(2.5*pt)
    canvas:encode_polyline(pl)
    local drv = barracuda:get_driver()
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

Pretty sure that this new version is more clear and intuitive.

5 Example and use cases

TODO