the `barracuda` manual

`https://github.com/robitex/barracuda`

Roberto Giacomelli
email: `giaconet.mailbox@gmail.com`

Date 2022-06-23 — Version v0.0.12 — 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.

`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),
- UPC-A.

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 may change respect to the result of development activity.

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 → 3
- print your first barcode → 4
- installing `barracuda` on your system → 6

Part 2: LATEX packages

- `barracuda` LATEX package → 7

Part 3: Barcode Reference

- barcode symbologies reference → 7

Part 4: Developer zone

- the Lua framework → 8
- encoder identification rule → 9
- API reference → 9
- `ga` specification → 9

Part 5: Real examples

- working example and use cases → 18

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 window of your system, typing the command:

```
$ texdoc luatex
```

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.

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 the Lua library.

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

1.4.1 A Lua script

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

In Lua a constructor or even a method call syntax it's a little bit different from the usual form because we have to use the *colon notation*: `object:method(args)`

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:

`first-run.lua`

```
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`

Into the same directory of your script you will see a new file called `my_barcode.svg` with the drawing:

8 006194 056290

Coming back to the script, the first statement loads 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 four mandatory argumetns: 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 in the `option=value` format.

In more detail, thanks to treename identifier explained at section 4.3 is possible to build more encoders of the same symbology each with a different set of parameters.

It's also possible to run a Lua script with `texlua`, the Lua interpreter improved with certain LuaTEX libraries delivered by any modern TEX distribution. `texlua` saves you to install Lua if you are a TEX user.

The command to run `first-run.lua` is the same as before, just a substitution of the name `lua` with `texlua`, but an adjustment is required if we want to run the script with TEX delivered `barracuda` library leaving untouched the system outside `texmf`.

An alternative path searching procedure consists to find the main file of `barracuda` with an internal LuaTEX library called `kpse`:

```
-- texlua script
kpse.set_program_name("luatex")
local path_to_brcd = kpse.find_file("barracuda", "lua")
local barracuda = dofile(path_to_brcd)
barracuda:save("ean-13", "8006194056290", "my_barcode", "svg")
```

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 previous script: the Lua code lives inside the argument of a `\directlua` primitive, moreover we must use an horizontal box register as the output destination.

`first-run.tex`: LuaTEX version

```
% !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

A LATEX working minimal example would be:

`first-run.tex`: LuaLATEX version

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

8 006194 056290

---

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

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 tuning barcode parameters.

The low level workflow to draw a barcode object reveals more details on the internal architecture. In fact, we must do at least the following steps divided into three phases:

a.1 load the library,

a.2 get a reference to the `Barcode` abstract class,

a.3 build an encoder,

a.4 build a symbol passing data to an encoder's constructor,

b.1 get a reference to a new canvas object,

b.2 draw barcode on the canvas object,

c.1 load the driver,

c.2 print the figure as an external `svg` file.

In the phase **a** a barcode symbols is created, then in phase **b** a canvas object is filled with the graphic elements of the symbol, and finally in the phase **c** the canvas is sent to the driver output channel.

Following the procedure step by step, the resulting code is as the following listing, where the encoder is EAN variant 13:

```
-- a lua script
local barracuda = require "barracuda" -- step a.1
local barcode = barracuda:barcode()   -- step a.2
local ean13, err_enc = barcode:new_encoder("ean-13")      -- step a.3
assert(ean13, err_enc)
local symb, err_symb = ean13:from_string("8006194056290") -- step a.4
assert(symb, err_symb)

local canvas = barracuda:new_canvas() -- step b.1
symb:draw(canvas) -- step b.2

local drv = barracuda:get_driver() -- step c.1
local ok, err_out = drv:save("svg", canvas, "my_barcode") -- step c.2
assert(ok, err_out)
```

Anyway, more abstract methods allow the user to write a more compact code. For instance, phase **b** can be fuse with **c**, thanks to a a reference to the driver object included in the `canvas` object:

```
-- phase b + c
local canvas = barracuda:new_canvas() -- step bc.1
symb:draw(canvas) -- step bc.2
local ok, err_out = canvas:save("svg", "my_barcode") -- step bc.3
assert(ok, err_out)
```

As we have been seen before an high level method provides a way to unify all the phases:

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

Low level code offers more control while high level programming is quite compact. Late in the manual you will find the objects and methods reference at section 4.4.

1.6 Installing `barracuda`

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.

Figure 1: Barcode class hierarchy.

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 reported as not installed, run the command:

```
$ tlmgr install barracuda
```

If you have installed TeX Live via your Linux repository, try your distribution's package manager an update or check for optional packages not yet installed.

It's also possible to install `barracuda` manually with these steps:

1. Grab the sources from CTAN or from the official repository `https://github.com/robitex/barracuda`.
2. Unzip it at the root of one of 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{code128}{123ABC}\\[2ex]
\barracuda[text_star=true]{code39}{123ABC}
\end{document}
```

123ABC

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

3 Barcode Reference

3.1 Common, Global and Local Barcode Options

Every barcode encoder inherits from `Barcode` abstract class methods and options. If we change its option values, the changes will be global for all the encoders except if the encoder has not an own local option overwritten before.

The same schema applying also for encoder and the barcode symbols build apart from it. Every symbol inherits methods and options from its encoder.

Such three levels option system is designed to allow the user to set up option not only in a certain point in the tree object, but also any time in the code. When changes are accepted by an object they become valid for that time on.

The architecture of barcode classes is shown in more details in figure 1. At the top of the hierarchy there is the `Barcode` class. It's an abstract class in the sense that no symbols can be printed by that class.

At an intermediate level we found a `Builder` with an instance of one of its `Encoder` class. When we call method `new_encoder()` provided by `Barcode` class, what really happen is the loading of the `Builder` if not just loaded before, that is the actual library of the specific simbology, and a linked `Encoder` object incorporates its own options.

At the last level are placed the symbol instances derived both from the `Builder` and `Encoder`, the first provides methods while the second provides option values. Only these objects are printable in a barcode graphic.

Common options of `Barcode` are the following:

Option Id	Type/default	Description
`ax`	numeric/0	Relative x-coordinate for insertion point of the barcode symbol
`ay`	numeric/0	Relative y-coordinate for insertion point of the barcode symbol
`debug_bbox`	enum/`none`	Draw symbol bounding box with a thin dashed line
	`none`	do nothing
	`symb`	draw the bbox of the symbol
	`qz`	draw the bbox at quietzone border
	`qzsymb`	draw symbol and quietzone bboxes

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

3.2 Code39

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

3.3 Code128

4 Developer zone

4.1 The Barracuda Framework

The `barracuda` package framework consists in independent modules: a barcode class hierarchy encoding a text into a barcode symbology; a geometrical library called `libgeo` modeling 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 parameters at any level: barcode abstract class, encoder globally, down to a single symbol object.

The Barcode class is completely independent from the output driver and vice versa.

4.2 Error Management

Functions in Lua may return more than one parameters. `barracuda` methods takes advantage by this feature for the error management. In fact, `barracuda` as a library, remind the responsibility to the caller in order to choose what to do in case an error is reported.

When a method may fail depending on the correctness of the input, it returns two parameters alternatively valid: the first is the expected result while the second is the error description.

This behavior perfectly match the arguments required by the `assert()` built-in function.

4.3 Encoder Treename

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

4.4 API reference of Lua modules

TODO

4.5 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. Coordinates or dimensions must be expressed as *scaled point*, the basic unit of measure of TeX equivalent to $1/65536$ pt.

4.5.1 Hard coded an horizontal line

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:

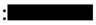
```
1 393216 33 0 2097152 0
```

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

```
linewidth 393216   ; set line width to 393216sp
hline 0 2097152 0  ; draw hline x1 x2 y
```

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

The result is: 

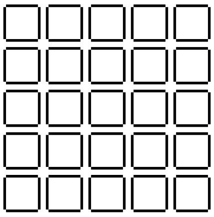
4.5.2 Encoding `ga` with the `gaCanvas` class

A more abstract way to write a `ga` stream is provided by the `gaCanvas` class of the `libgeo` module. Every operation with identifier `opcode` is mapped to a method named `encode_<opcode>()` of a canvas object:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local canvas = barracuda:new_canvas()
    local pt = canvas.pt
    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 side-by-side the listing):

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local canvas = barracuda:new_canvas()
    local pt = canvas.pt
    local side = 16*pt
    local s = side/2 - 1.5*pt
    local l = side/2 - 2*pt
    local dim = 4
    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)
    canvas:ga_to_hbox("mybox")
    canvas:save("svg", "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 produces a more fast execution and the figure maintains the bounding box computed until that point. The `set_bbox` operation imposes a bounding box in comparison to the current one of the figure.

The initial bounding box is simply empty. As a consequence, different strategies can be used to optimize runtime execution, such as in the previous 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"
```

4.5.3 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

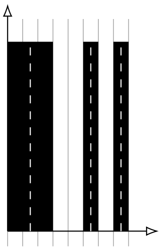
OpCode	Mnemonic key	Graphic object/Operands
130	text	A text with several glyphs ax <abs> ay <abs> xpos ypos <point> [c <chars>]+
131	text_xspaced	A text with glyphs equally spaced on its vertical axis x1 <coord> xgap <len> ay <abs> ypos <coord> [c <chars>]+
132	text_xwidth	Glyphs equally spaced on vertical 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

4.6 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 length called `mod`, defined by the array and figure 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 a gray vertical grid stepping one module and white dashed lines at every vbar 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 constructors one of which is `from_int()`. Its arguments are the multiplier integer `ngen`, the module length `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
```

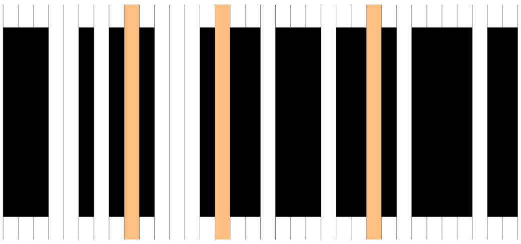
4.6.1 `Vbar` class arithmetic

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 `Vbars`. More in detail, to add two `Vbars` 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:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local Vbar = barracuda:libgeo().Vbar
    local canvas = barracuda:new_canvas()
    local mm = canvas.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
    canvas:encode_vbar_queue(queue, 0, 0, 25*mm)
    canvas:ga_to_hbox "mybox"
}\leavevmode\box\mybox
\bye
```



[a]

---

[a] Respect to the showed code some graphical helps has been added: a vertical grid marks the module wide steps and light colored bars mark the space added between two `Vbars`.

4.7 ga programming

To provide a better learning experience several ga stream examples is discussed, each of which must be compiled with LuaTEX.

4.7.1 Example 1: a rectangle

Suppose we want to draw a simple rectangle. In the ga reference of section 4.5.3 there is a dedicated instruction `<rect>`. Let's give it a try:

Example 1: dealing with raw ga stream

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local pt = 65536
    local ga = {48, 0, 0, 72*pt, 36*pt}
    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 more safely a `gaCanvas` object running its `encode_rect()` method:

```
...
local canvas = barracuda:new_canvas()
```

```
assert(canvas:encode_rect(0, 0, 2*side, side))
assert(canvas:ga_to_hbox("mybox"))
...
```

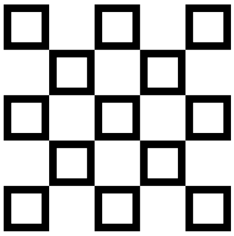
4.7.2 Example 2: a chessboard

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 canvas = barracuda:new_canvas()
    local mm = canvas.mm
    local s, t = 7.5*mm, 1.5*mm
    canvas:encode_linewidth(t)
    assert(canvas:encode_rect(t/2, t/2, s-t/2, s-t/2))
    assert(canvas:ga_to_hbox("mybox"))
}\leavevmode\box\mybox
\bye
```

Then repeat the game for the entire grid:

```
% !TeX program = LuaTeX
\newbox\mybox
\directlua{
    local barracuda = require "barracuda"
    local canvas = barracuda:new_canvas()
    local mm = canvas.mm
    local s, t = 6*mm, 1*mm
    assert(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
                local x1, y1 = x + t/2, y + t/2
                local x2, y2 = x + s - t/2, y + s - t/2
                assert(canvas:encode_rect(x1, y1, x2, y2))
            end
        end
    end
    drv:ga_to_hbox(canvas, "mybox")
}\leavevmode\box\mybox
\bye
```

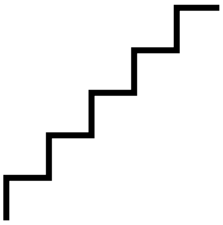


4.7.3 Example 3: a staircase

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 = 16*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(2.25*pt)
    canvas:encode_polyline(point)
    canvas:ga_to_hbox("mybox")
}\leavevmode\box\mybox
\bye
```



A feature of `encode_<opcode>()` methods is their *polymorphic* behavior for their first argument. They accept different types as an object of a geometric class or the raw geometric data.

Method `encode_polyline` is not an exception: it accepts a `Polyline` object provided by the `libgeo` module, or instead a flat array of coordinates. For instance the previous code may be re-implement as:

```
% !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(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)
    canvas:ga_to_hbox("mybox")
}\leavevmode\box\mybox
\bye
```

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

5 Practical examples and use cases

Previous sections as shown how `barracuda` is capable to draw simple graphics. This section is dedicated to barcode applications.