Generating barcodes with LuaTeX

Patrick Gundlach

Abstract

TeX is great at typesetting, Lua is great with calculations. When we combine those two, we can do complex typesetting tasks easily. In this article, I present a way to create GTIN-13 barcodes (also known as EAN 13) with LuaTeX. The aim of this article is to present how to call Lua from TeX, some basic Lua programming and two ways in which Lua and TeX can interact.

1 Introduction

There are several ways to generate barcodes from TeX: one is the PSTricks barcode package; a few packages rely on special fonts; and one I found generates barcodes with vertical rules, but the source is not suitable for beginners and therefore rather hard to extend. The Lua solution I present is supposed to be easier to understand for non-TeX programmers, but this is subjective, of course. For the purpose of this demonstration, only EAN 13 barcodes are handled, optionally calculating the checksum (the last digit) if the requested barcode has only twelve digits. This is an example output of our program:

4 242002 518169

The LaTeX interface should be as simple as this:

```
\documentclass{article}
\usepackage{ltxbarcode}
\begin{document}
\barcode{424200251816}
% or -- with checksum:
\barcode{4242002518169}
\end{document}
```

The "glue" style file `ltxbarcode.sty` is short as well. You can place it in the same directory as the main LaTeX file above:

```
\ProvidesPackage{ltxbarcode}

\directlua{require("ltxbarcode")}

\newcommand\barcode[1]{%
  \directlua{
     ltxbarcode.generate_barcode("#1")
  }}
```

The package loads the file `ltxbarcode.lua`, which is given and explained in full detail below. `require` appends the extension `.lua` by itself. Then it defines the command `\barcode`, whose sole task is to jump into Lua mode (`\directlua`) and call the Lua function `generate_barcode()`, passing along the argument given to the LaTeX command. The prefixed namespace `ltxbarcode` is automatically created with `require()` at the beginning.

There is a small pitfall here. You would normally write this code as:

```
\newcommand\barcode[1]{%
  \directlua{
   ltxbarcode.generate_barcode(
     "\luatexluaescapestring{#1}"
     )}}
```

to protect the Lua string from the first macro argument (`#1`) containing characters that might possibly break the Lua parser. If the macro argument contains, for example, a double quote character, Lua will see it as the end of the quoted string and choke on the rest of the argument. Since we are only passing ordinary digits, the string is safe in our code above. (Protection is a good idea, though.) The long name from above (`\luatexluaescapestring`) is the command `\luaescapestring` found in the reference manual prefixed with `luatex` to avoid name clashes. The only command from LuaTeX not prefixed is `\directlua`. This is only valid for TeXLive's current LuaLaTeX format. The plain LuaTeX format in TeX Live has all commands unprefixed. I fervently hope that other distributions behave exactly the same.

Before we have a close look at our Lua module (the file we load in our LaTeX package), let's take an excursion on how to communicate between the Lua mode and LuaTeX.

2 From Lua to TeX

Passing arguments from TeX to Lua is easy, as seen above; e.g., the call to `generate_barcode()`. But the other way is a bit more interesting, as one has to keep in mind when the code gets executed. The Lua code in `\directlua` gets executed the moment LuaTeX finds the closing brace of that command. It will then be replaced by the special buffers that this command fills. Example:

```
  \directlua{
     tex.sprint("\\hbox{%")
     tex.sprint("hello world%")
     tex.sprint("}")
    }
```

and the TeX code

```
\hbox{%
hello world%
}
```

are more or less equivalent. Thus, strings can be split into smaller chunks and automatically concatenated (with line endings as separators) automatically at the end of the `\directlua` call. What is not possible, though, is the following:

```
\directlua{
  tex.sprint("\\setbox0\\hbox{hello world!}")
  % does NOT work because box 0 is not set yet:
  tex.sprint(
    string.format(
      "The width of box 0 is now \%d",
        tex.box[0].width ))
}
```

Inside `\directlua` it is not possible to mix TEX and Lua calculations like this, because the TEX value is not known until the end of the `\directlua` call. So you cannot operate on the box dimensions before TEX typesets that box. Keep this in mind as we compare the two approaches I will show now.

3 Solution one: `tex.sprint()`

The first approach is to calculate the barcode itself (this is a simple routine) and construct a set of `\hbox`, `\vrule` and `\kern` commands with the Lua function `tex.sprint()`. Before we dive into the main function, we start with the head of the Lua file (named `ltxbarcode.lua`). Most Lua modules start with a call to `module()`. We make all but the main functions *local*, meaning that they are only visible inside the module.

```
module(...,package.seeall)
local add_checksum_if_necessary, mkpattern,
     split_number, calculate_unit, pattern_to_wd_dp
```

Now comes the heart of the module. We use the method described above to generate a sequence of TEX commands that get executed right after the `\directlua` call. The idea is to draw the barcode with vrules and kerns and add the digits below in a separate box.

```
function generate_barcode( str )
  -- If we only pass 12 digits, the 13th will be added.
  str = add_checksum_if_necessary(str)

  -- The smallest bar/gap is 1/7th the width of a digit.
  -- It is font dependent.
  local u = calculate_unit()

  -- We start with the hbox for the bars:
  tex.sprint(
    [[\newbox\barcodebox\setbox\barcodebox\hbox{%]]
    )

  -- The pattern is a string of digits that represent
  -- the width of a bar or a gap. 0 is a special marker
  -- for a longer bar of width 1. The widths are
  -- multiplied by 1/7th of the width of a digit, because
  -- the sum of the widths for a single digit add up to 7.
  -- A sample pattern starts with:
  -- 80103211112312132113231132111010132...
  -- See function mkpattern() for a detailed explanation.
  local pattern = mkpattern(str)

  -- For each element in the pattern we generate a gap or
  -- a bar of the width denoted by that element. A depth
  -- >0 is used for the bars in the middle and both sides.
  -- This is technically not necessary, but added to have
  -- visually pleasing barcodes.
  local wd,dp -- width and depth of a bar
  for i=1,string.len(pattern) do
    wd,dp = pattern_to_wd_dp(pattern,i)
    -- The even elements are the vertical bars (vrules),
    -- the odd ones are the gaps (kerns).
    if i % 2 == 0 then
      tex.sprint(string.format(
        [[\vrule width %dsp height 2cm depth %s]],
        wd * u,dp))
    else
      tex.sprint(string.format(
          [[\kern %dsp]],wd * u))
    end
  end
  -- We now have the hbox with the bars and
  -- add the hbox with the digits.
  tex.sprint([[}\vbox{\hsize\wd\barcodebox \box\
       barcodebox\kern -1.7mm\hbox{%]])

  -- The digits below the barcode are split into three
  -- groups: one in front of the first bar, the first
  -- half of the other digits are left of the center
  -- bar, and the remaining digits are to the right
  -- of the center bar.
  local first,second,third = split_number(str)
  tex.sprint(string.format(
     [[%s\kern %dsp %s\kern %dsp%s}}]],
     first, 5 * u, second, 4 * u, third ))
end
```

The main function uses several helper functions. One of them calculates the width of the smallest bar and the smallest gap, which is exactly 1/7th the width of a digit. We make use of LuaTEX's font library where we can get access to the current font. The glyph number 48 is the digit zero; theoretically, this is encoding-dependent, but in practice it works in all cases.

```
function calculate_unit()
  -- The relative widths of a digit represented by the
  -- barcode add up to 7.
  local currentfont = font.fonts[font.current()]
  local digit_zero = currentfont.characters[48]
  return digit_zero.width / 7
end
```

The next function determines the width and the depth of a vertical rule. The height is fixed (we could have made that customizable, but the reader should be left with some task to do).

```
function pattern_to_wd_dp( pattern,pos )
  local wd,dp
  wd = tonumber(string.sub(pattern,pos,pos))
  if wd == 0 then
    dp = "2mm"
    wd = 1
  else
    dp = "0mm"
  end
  return wd,dp
end
```

The calculation of the checksum is straightforward. We sum up all the digits, every other digit is multiplied by 3 (counted from the last digit backwards) and the checksum is the amount you need to add to get to the next multiple of 10. The sum is only calculated if not given by the user. (A future version could check a user-supplied value.)

```
function add_checksum_if_necessary( str )
  if string.len(str) == 13 then
    return str
  end

  local sum = 0
  local len = string.len(str)
  for i=len,1,-1 do
    if (len - i ) % 2 == 0 then
      sum = sum + tonumber(string.sub(str,i,i)) * 3
    else
      sum = sum + tonumber(string.sub(str,i,i))
    end
  end
  local checksum = (10 - sum % 10) % 10
  return str .. tostring(checksum)
end
```

The following pattern generation is the heart of the algorithm. The barcode is divided into smaller parts where two bars and two gaps represent a single digit. The widths of these vary between "one" and "four", multiplied by any sensible width. The widths for a single digit add up to 7 of these units and are expressed by a simple pattern such as *2221* for the digit 1. The first digit in a barcode is not represented by a bar–gap pair, but rather encoded in the representation of the next six digits. If, for example, the first digit is a *1*, the third, fifth and sixth "digits" have to be reversed. See the array `mirror_t` in the code below. In the example above the reversed pattern is *1222*. We add some space to the left of the barcode for the first digit and also mark the left and right edge with the special mandatory pattern *111*. Actually it is *010* which we recognize later to increase the length of these bars.

```
function mkpattern( str )
  -- These are the digits represented by the bars.
  -- 3211 for example means a gap of three units,
  -- a bar two units wide, another gap of width one
  -- and a bar of width one.
  local digits_t = {"3211","2221","2122","1411",
         "1132","1231","1114","1312","1213","3112"
     }

  -- The first digit is encoded by the appearance of the
  -- next six digits. A value of 1 means that the
  -- generated gaps/bars are to be inverted.
  local mirror_t = {"------","--1-11","--11-1",
            "--111-","-1--11","-11--1","-111--",
             "-1-1-1","-1-11-","-11-1-"}

  -- Convert the digit string into an array.
  local number = {}
  for i=1,string.len(str) do
    number[i] = tonumber(string.sub(str,i,i))
  end

  -- The first digit in a barcode determines how the
  -- next six digit patterns are displayed.
  local prefix = table.remove(number,1)
  local mirror_str = mirror_t[prefix + 1]

  -- The variable pattern will hold the constructed
  -- pattern. We start with a gap that is wide enough
  -- for the first digit in the barcode and the special
  -- code 111, here written as 010 as a signal to
  -- create longer rules later.
  local pattern = "8010"
  local digits_str

  for i=1,#number do
    digits_str = digits_t[number[i] + 1]
    if string.sub(mirror_str,i,i) == "1" then
      digits_str = string.reverse(digits_str)
    end
    pattern = pattern .. digits_str
    -- The middle two bars.
    if i==6 then pattern = pattern .. "10101" end
  end
  -- Append the right 111 pattern as above.
  return pattern .. "010"
end
```

The last function splits the barcode into three parts so we can display the digits below the barcode with some gaps in between.

```
function split_number( str )
  return string.match(
    str,"(%d)(%d%d%d%d%d%d)(%d%d%d%d%d%d)"
    )
end
```

The net result of this code is a TeX string like this:

```
\newbox\barcodebox\setbox\barcodebox\hbox{%
\kern 374492sp
\vrule width 46811sp height 2cm depth 2mm
\kern 46811sp
...
\vrule width 46811sp height 2cm depth 2mm
}\vbox{\hsize\wd\barcodebox\box\barcodebox\kern -1.7
     mm\hbox{%
8\kern 234057sp 008940\kern 187246sp027004}}
```

This is what TEX sees after the closing brace of `\directlua`. While this solution works fine in our small example, it can get a bit tedious, because of the string passing and the necessity to escape all occurrences of the well-known *funny* TEX chars such as `%` and others. Luckily with LuaTEX, our new swiss army knife in the TEX world, we have another approach to that problem.

4 Solution two: direct typesetting with low-level nodes

The other approach to that problem looks like using a sledge-hammer to crack a nut. But once one becomes used to it and some helper functions defined, this solution is well-suited for many tasks when we are using Lua for program logic. The idea is to create the fundamental data structures TEX uses internally for representing the typeset material: a *node*. A node can represent a glyph, a rule, a glue, a whatsit and all other items we know from *The TEXbook*. The typeset digit '0' for example could be represented by a table with these entries:

entry	value
`id`	37
`char`	48
`font`	15
`lang`	0

There are other optional entries in that table, but only the *prev* and the *next* entries are necessary for building a more complex data structure. The table above can be constructed from Lua like this:

```
n = node.new("glyph") -- internal id: 37
n.char = 48
n.font = 15
n.lang = 0
```

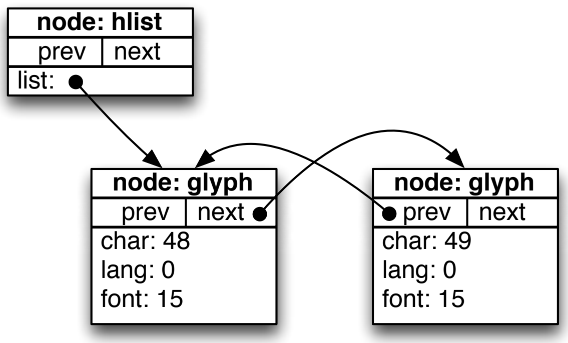
To construct a horizontal box with the digit created above a call to `node.hpack()` is sufficient:

```
hbox = node.hpack(n)
```

Which is the same as `\hbox{0}` except that the box is only kept in TEX's memory and not put into the PDF. It gets more complex when you want more than one item to be placed in a box. You then need to create the nodes and chain them together into a list. Every node has *prev* and *next* table entries which are to be set to the predecessor and successor nodes. So in the case of the two digits 0 and 1 placed in a horizontal box, it would look like:

```
digit_0 = node.new("glyph")
digit_1 = node.new("glyph")
-- not shown: fill the tables as above
digit_0.next = digit_1
digit_1.prev = digit_0
hbox = node.hpack(digit_0)
```

The result is a data structure that can be visualized by the following graphic:



The *list* entry of the hlist (hbox) points to the *node list* starting with the digit 0. The idea for our second approach is to create a node list that represents the vertical bars and gaps (rule and kern nodes) and digits. We create a few more helper functions as well as the new main function:

```
local add_to_nodelist, mkrule, mkkern, mkglyph
function generate_barcode_lua( str )
  str = add_checksum_if_necessary(str)

  local u = calculate_unit()
  local nodelist

  -- The even elements are the rules,
  -- the odd ones are the gaps.
  local pattern = mkpattern(str)
  local wd,dp
  for i=1,string.len(pattern) do
    wd,dp = pattern_to_wd_dp(pattern,i)
    if i % 2 == 0 then
      nodelist = add_to_nodelist(
        nodelist,mkrule(
          wd * u,tex.sp("2cm"),tex.sp(dp)))
    else
      nodelist = add_to_nodelist(
        nodelist,mkkern(wd * u))
    end
  end
  -- barcode_top will become the vbox as in the
  -- first solution.
  local barcode_top = node.hpack(nodelist)
  barcode_top = add_to_nodelist(
    barcode_top,mkkern(tex.sp("-1.7mm")))

  -- The following list holds the displayed digits.
  nodelist = nil
  for i,v in ipairs({split_number(str)}) do
    for j=1,string.len(v) do
      nodelist = add_to_nodelist(
        nodelist,mkglyph(string.sub(v,j,j)))
    end
    if i == 1 then
      nodelist = add_to_nodelist(
          nodelist,mkkern(5 * u))
    elseif i == 2 then
      nodelist = add_to_nodelist(
          nodelist,mkkern(4 * u))
    end
  end
 local barcode_bottom = node.hpack(nodelist)
 -- barcode_top now has three elements: the hbox
 -- from the rules and kerns, the kern of −1.7mm
 -- and the hbox with the digits below the bars.
 barcode_top = add_to_nodelist(
      barcode_top,barcode_bottom)
 local bc = node.vpack(barcode_top)

 -- node.write() puts a vbox into the output.
 node.write(bc)
end
```

The overall structure is exactly the same as in the previous section. The main difference is the use of the helper functions `mkrule()`, `mkkern()` and `mkglyph()` to create rules, kerns and glyphs and the call to `add_to_nodelist()`. The constructed node list is written to the PDF with the Lua call `node.write()`.

```
function add_to_nodelist( head,entry )
  if head then
    -- Add the entry to the end of the nodelist
    -- and adjust prev/next pointers.
    local tail = node.tail(head)
    tail.next = entry
    entry.prev = tail
  else
    -- No nodelist yet, so just return the new entry.
    head = entry
  end
  return head
end
```

If the node list exists, the new entry is appended to the last node of that list. We could get to the end of the list by following successive pointers until we reach the one with the “empty” pointer *nil*, but we use the LuaTEX function `node.tail()` instead. Then we adjust the next and prev pointers of the tail and the new entry and return the head of the node list.

```
function mkrule( wd,ht,dp )
  local r = node.new("rule")
  r.width = wd
  r.height = ht
  r.depth = dp
  return r
end

function mkkern( wd )
  local k = node.new("kern")
  k.kern = wd
  return k
end

function mkglyph( char )
  local g = node.new("glyph")
  g.char = string.byte(char)
  g.font = font.current()
  g.lang = tex.language
  return g
end
```

These three functions don’t need much explanation. They generate the nodes of the requested types. It might surprise at first glance that the glyph node needs a language and a font entry, because in ordinary TEX we usually don’t care about this. But remember that the nodes are the low-level data structures created when all of TEX’s input is already processed, except for the hyphenation and justification of the paragraph.

As a final note on the source, the Lua file described here can be downloaded from `https://gist.github.com/1513746`.

5 Conclusion

There are two ways to pass typesetting information from Lua to TEX: first, with a collection of `tex.sprint()` calls, and second, with a set of nodes.

Once you are in the Lua world, it feels wrong to pass information to TEX with `tex.sprint()` calls. You still have to deal with category codes, with grouping and with all the headaches that character escaping brings.

In the procedural world of Lua, the *right* way to do typesetting is to construct the input with low-level data structures and helper functions and let TEX’s algorithms do the rest. Once you start thinking in terms of nodes and node lists, you can focus on arranging items on the page and not let TEX’s input language get in your way.

These days, TEX’s input language seems anachronistic to many people, while procedural languages like Lua are familiar. TEX’s algorithms are still unsurpassed, so when you combine Lua’s power with TEX’s typesetting capabilities, a whole new generation of applications become possible.

References

[1] Patrick Gundlach. TEX without TEX. `http://wiki.luatex.org/index.php/TeX_without_TeX`, 2011.

[2] Taco Hoekwater. LuaTEX reference manual. `http://mirror.ctan.org/systems/luatex/base/manual`, 2011.

[3] Herbert Voß. The current state of the PSTricks project. *TUGboat*, 31(1), 2010.

◇ Patrick Gundlach
Eisenacher Straße 101
10781 Berlin
Germany
`patrick (at) gundla dot ch`