Page design in LaTeX3

Morten Høgholm
LaTeX3 Project
`morten dot hoegholm (at) latex dash project dot org`

Abstract

Choosing a page layout in LaTeX is easy for the user with the help of packages like geometry and typearea. However, users face various problems when wishing to change the layout parameters mid-document — something which happens quite often: title pages, rotated pages (with rotated header and footer), special pages or spreads including large images, and other situations where manual fiddling is a difficult and error prone process.

This article describes an interface for defining, storing, and retrieving complete page layouts. It will take a look under the hood to see how the data structures and programming constructs provided by the LaTeX3 kernel ease the programming task.

1 Introduction

Setting and understanding the page layout parameters in standard LaTeX is not the easiest of tasks. There is no interface at all so it must be done by setting all parameters manually and then hope you got them right. The packages geometry and typearea both provide an interface for changing the parameters for the entire document but there are situations where one may wish to use a different layout for just a few pages of the document:

- Title and back pages, where content is often centered on the physical page.
- Rotated pages, where some users want to also rotate header and footer.
- Page spreads containing material crossing page borders.

Additionally, the ubiquitousness of PDF documents on the Internet has opened up a new window of opportunity: changing page size mid-document.

2 The current solutions

Before trying to figure out a way to deal with page layout in LaTeX3, it is probably a good idea to take a closer look at the existing solutions within the LaTeX 2_ε framework.

2.1 The base distribution

The LaTeX kernel doesn't really have much of an interface when it comes to modifying the layout. The only way to change anything is to set the dimensions by hand by means of commands such as

```
\setlength\paperheight{29.7cm}
\setlength\paperwidth{21cm}
```

Using the "raw" LaTeX commands like this is not an ideal interface for a designer. The layout package from the tools bundle alleviates this a bit by drawing the layout for you so that you can check if it looks as intended. This way you may also discover conflicting settings, as the kernel does not check this itself.

Should one wish to change some parameters temporarily mid-document their values must either be stored or the changes done locally. However, the parameters are global (in the sense that they are set at the top level in the document preamble) and it is bad practice to do local changes to global parameters. In short: there are no technical hindrances for the adventurous user to change the parameters at will but it is at best a tedious and error prone procedure.

Another problem is that recto and verso pages are exactly the same except for `\evensidemargin` and `\oddsidemargin`. As a consequence, certain types of designs become much harder to do in LaTeX than you'd expect. For example, defining a page layout where every odd page is six lines shorter (because, say, this is reserved for supplementary text) is slightly difficult because one has to manually change the text height from page to page.

2.2 The geometry and typearea packages

The most popular package for changing the page layout is the geometry package [4]. It provides an easier interface, one using a ⟨*key*⟩ = ⟨*value*⟩ syntax whereby the designer/user is shielded from the direct form of the LaTeX parameters. The syntax is almost self-explanatory:

```
\usepackage{geometry}
\geometry{
  paper = a4paper ,
  textwidth = 6in ,
  lines = 42
}
```

Not surprisingly this makes the textwidth 6 in and puts 42 lines of text on A4 paper. geometry does not try to enforce typographic rules of thumb on the user except for choosing margin ratios suitable for printed books. Other than that, the user is responsible for everything.

A different approach is taken by the typearea package from the koma-script bundle [3]. Based on the document font, it tries to produce a textwidth of about 60–70 characters and then defines the layout in accordance with good typographic practice, such as a 1:2 relationship between top and bottom margin.

For a more in-depth discussion of both geometry and typearea, see [2]. Common to both packages is that they work only for the entire document: All settings must be done in the preamble and thus we have not yet solved one of our initial problems: changing the parameters mid-document.

3 A new solution

In order to come up with a solution to the problems, let's take a step back and look at what sort of areas may appear on a page.

3.1 The parts of a page

Instead of defining recto and verso pages we will define one standard page description so that all different page types have the same set of parameters, each parameter corresponding to a letter as shown in Figure 1. The indices w, h, and s denote width, height, and separation respectively. The central thing on the page is the textblock T itself. The textblock is surrounded on all sides by header, footer, left area and right area. The left and right areas are where margin notes may be positioned or as seen in some designs where the designer uses the right margin to place captions, section headings etc. and outside these areas are the actual margins. Finally the final paper size is often different from the stock size so we must take this into account too. We can notice several relationships:

$$\begin{aligned}
S_w &= t_L + t_R + p_w \\
S_h &= t_T + t_B + p_h \\
p_w &= m_L + L_w + L_s + T_w + R_w + R_s + m_R \\
p_h &= m_T + H_h + H_s + T_h + F_s + F_h + m_B
\end{aligned}$$

These relationships are similar to what can be found in geometry and ease the task of autocompletion and error checking.

The output routine will have to know which values to use and the easiest way is probably to use a set of global parameters with names such as

```
\g_page_std_textheight_dim
\g_page_std_textwidth_dim
```

etc. For those unfamiliar with the LaTeX3 naming conventions, these are global dimension registers belonging to the page module and going by the name std_⟨*parameter*⟩. In a similar manner we will make each page type have an identical set of parameters, except these go by names like

```
\l_page_⟨type⟩_textheight_dim
\l_page_⟨type⟩_textwidth_dim
```

etc., i.e., they are local parameters and then the shipout routine can handle setting the global parameters to the value of the local ones. Defining a complete set of parameters for each page type uses quite a number of dimension registers but since LaTeX3 uses ε-TeX (or better) we are not tied to solutions working within the tightly confined space of the Knuthian TeX engine.

Since we have decided that all page types share the same parameters, we must also solve the problem of storing and retrieving parameter values since they must change between, for instance, recto and verso pages. For this task we turn our attention to some of the tools provided by the LaTeX3 kernel.

3.2 Tools

TeX comes with only a few kinds of registers and anything dealing with lists must be done in macros. The tools for list processing in the LaTeX 2ε kernel are restricted to token lists and comma separated lists, and even then there are only mapping functions. The LaTeX3 kernel alleviates this by both extending the list of data structures with sequences and property lists as well as providing a complete set of tools for each data type: mapping, push/pop operations, etc. For the problem at hand we will use property lists.

3.2.1 Property lists

A property list is a list structure consisting of a series of keys, each with its own information field, of the form

```
\⟨key 1⟩{⟨info 1⟩}
\⟨key 2⟩{⟨info 2⟩} ...
```

For example, a small test file for the cross referencing module of LaTeX3 contains a property list `\g_xref_mylabel_plist` which expands to

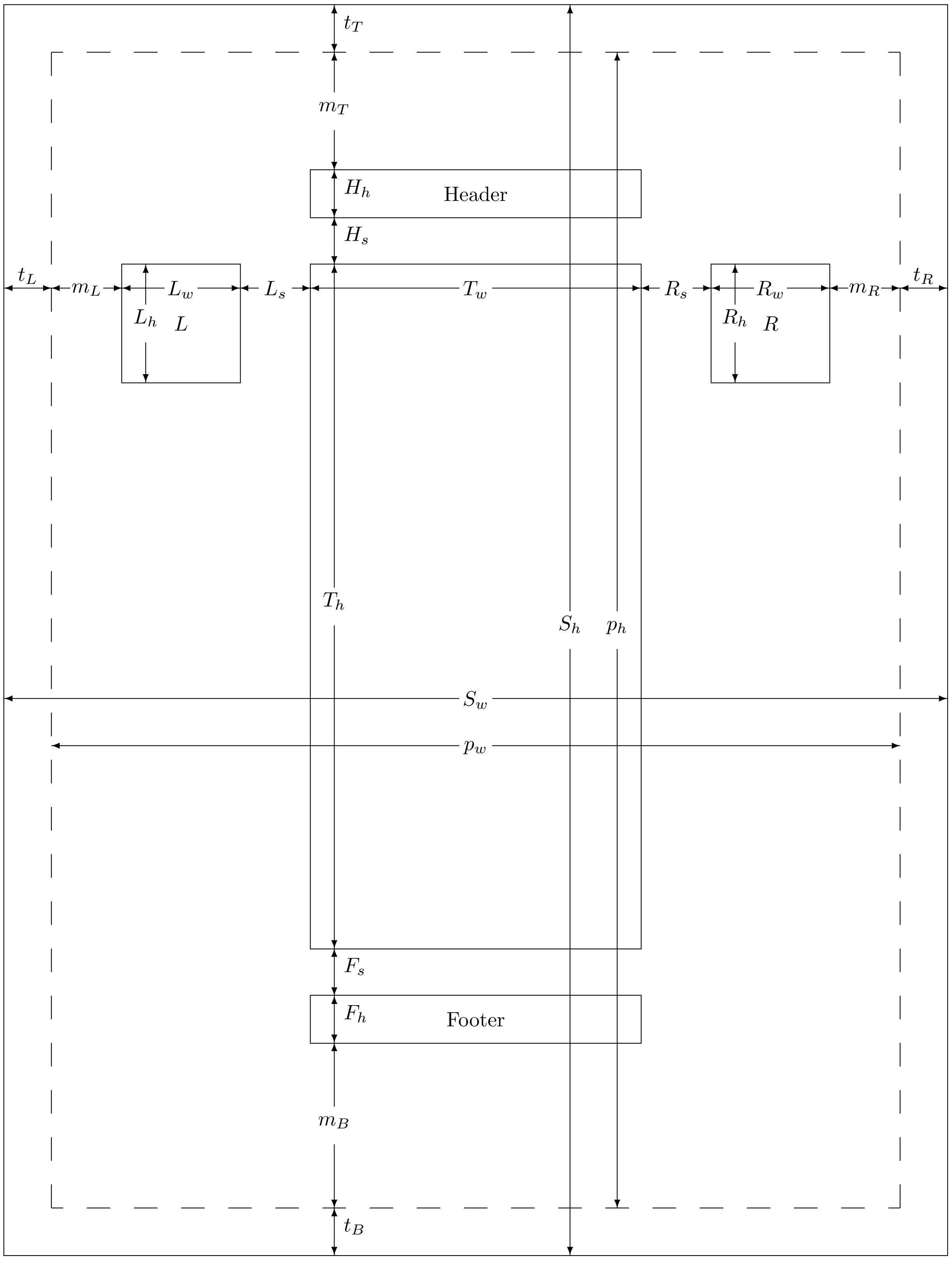

Figure 1: The parts of a page: S is stock size, p is paper size, t are trims, m is margin, T is the text block, L and R are marginal note areas, and F and H are footer and header resp.

```
\xref_name_key {This is a name}
  \xref_valuepage_key {1}
    \xref_page_key {i}
```

Property lists are a natural way to store series/collections of information. One such application could be the babel language strings or, as we shall see, an array of length registers with saved values.

For our purposes we will store the property lists with names like `\g_page_recto_plist` with the following contents:

```
\l_page_recto_textheight_dim
  {\l_page_recto_textheight_dim}
\l_page_recto_textwidth_dim
  {\l_page_recto_textwidth_dim}
...
```

The reason is that we can have some application setting these parameters locally and then we can use the extended list processing tools of the LaTeX3 kernel to map a function on each info-pair and extract the data. Here's a small example of how this is done:

```
\tlp_gset:Nx \g_page_recto_tlp {
  \prop_map_function:NN
    \g_page_recto_plist
    \page_extract_dimensions:Nn
}
\def_new:NNn\page_extract_dimensions:Nn 2{
  \exp_not:N\dim_set:Nn #1{\dim_use:N #2}
}
```

`\tlp_gset:Nx` does an `\xdef` on its second argument and stores it in the global *token list pointer* `\g_page_recto_tlp`. `\prop_map_function:NN` is an expandable mapping operation which places its second argument in front of each info-pair of the property list, which is given as the first argument. In our example this argument is the auxiliary function `\page_extract_dimensions:Nn` which is called to get the current value of the parameter and preparing it to be set. The end result is a macro containing

```
\dim_set:Nn\l_page_recto_textheight_dim
  {536.0pt}
\dim_set:Nn\l_page_recto_textwidth_dim
  {310.0pt}
...
```

At the time of the shipout this list is run and then we also run a simple

```
\dim_gset:Nn
  \g_page_std_textheight_dim
  {\l_page_recto_textheight_dim}
\dim_gset:Nn
  \g_page_std_textwidth_dim
  {\l_page_recto_textwidth_dim}
...
```

Note that all of this does not touch the original property list, so a number of operations can be performed by mapping functions onto it. For example, one could reset all parameters to some special value (typically negative) and then a second mapping function could check if all parameters have indeed been set.

3.2.2 Templates

LaTeX3 provides the concept of *templates*, which are, in short, parameterized functions. As arguments, a template takes a list of named parameters given in the well-known 'keyval' syntax plus additional arguments, which are often user input. The template converts the named parameters into macro or register assignments and uses these when running the actual code in the template. While a template can be used directly, one often defines various *named* instances of a template, which is the template function run with a specific set of parameters. Defining and using an instance instead of running the entire template at runtime is faster in terms of execution time but also has the advantage that one can store several different versions of a template and then use specific instances depending on the needs at hand.

As a small example, let's imagine a template type for producing split level fractions such as $^3\!/_7$. The template type receives three arguments from the user:

1. Numerator.
2. Separator. In case of `\NoValue`, i.e., no argument, use a default symbol instead.
3. Denominator.

So for this template type one can define several different templates depending on the needs of the user and the font in question. With fonts not containing superior and inferior numbers one will have to manually raise and lower the characters whereas more advanced font sets such as Minion Pro contain these characters and then one can use a much simpler template, which basically just inserts the user input as-is. Figure 2 shows a complete example of a simple template and a possible user interface to the template using xparse, while Figure 3 shows the resulting output. The template defined in this example uses `\textfractionsolidus` as the default separator symbol.

In the example you can see how one can run the template directly, how to define instances and how to use instances as part of defining document syntax with xparse. An interesting observation is that using the solidus from Times works rather well with Computer Modern instead of the unusually large solidus found there.

```
\documentclass[12pt]{article}
\usepackage{template,xparse,textcomp,l3box}
\CodeStart
\dim_new:N \l_splitfrac_size_dim
\dim_new:N \l_splitfrac_pre_kern_dim
\dim_new:N \l_splitfrac_post_kern_dim
\box_new:N \l_splitfrac_tmpa_box
\DeclareTemplateType{splitfrac}{3}
\DeclareTemplate{splitfrac}{text}{3}{
  numerator-format    = f1 [#1] \splitfrac_numerator_format:n ,
  denominator-format  = f1 [#1] \splitfrac_denominator_format:n ,
  separator-format    = f1 [#1] \splitfrac_separator_format:n ,
  numerator-scale     = n  [.6]  \l_splitfrac_scale_tlp ,
  separator-symbol    = n [\textfractionsolidus]
                          \l_splitfrac_separator_symbol_tlp,
  separator-font      = n [\DelayEvaluation{\f@family}]  \l_splitfrac_separator_font_tlp ,
  separator-pre-kern  = l [\DelayEvaluation{-.1em}] \l_splitfrac_pre_kern_dim ,
  separator-post-kern = l [\DelayEvaluation{-.15em}] \l_splitfrac_post_kern_dim
}
{
  \DoParameterAssignments
  \sbox\l_splitfrac_tmpa_box{
    \splitfrac_separator_format:n {
      \fontfamily{\l_splitfrac_separator_font_tlp }\selectfont
      \IfNoValueTF{#2}{\l_splitfrac_separator_symbol_tlp}{#2}
    }  }
  \mbox{
    \dim_set:Nn\l_splitfrac_size_dim {
      \l_splitfrac_scale_tlp \dim_eval:n{\f@size pt}  }
    \raisebox{\box_ht:N \l_splitfrac_tmpa_box -\height }{
      \splitfrac_numerator_format:n {
        \fontsize{\l_splitfrac_size_dim }{\baselineskip}\selectfont
        #1  }  }
    \kern\l_splitfrac_pre_kern_dim
    \box_use:N \l_splitfrac_tmpa_box
    \kern\l_splitfrac_post_kern_dim
    \raisebox{-\box_dp:N \l_splitfrac_tmpa_box}{
      \splitfrac_denominator_format:n {
        \fontsize{\l_splitfrac_size_dim }{\baselineskip}\selectfont
        #3  }  }  }
}
\DeclareDocumentCommand\splitfrac{mom}{
  \IfExistsInstanceTF{splitfrac}{\f@family}
  {\UseInstance{splitfrac}{\f@family}}
  {\UseTemplate{splitfrac}{text}{}}
  {#1}{#2}{#3}
}
\CodeStop
\begin{document}
\UseTemplate{splitfrac}{text}{}{34}{\NoValue}{15},
\fontfamily{ptm}\selectfont\textonehalf,
\UseTemplate{splitfrac}{text}{
  separator-pre-kern=0pt,
  separator-post-kern=0pt}{1}{\NoValue}{2},
\DeclareInstance{splitfrac}{ptm}{text}{
  separator-pre-kern=0pt,
  separator-post-kern=0pt}
\UseInstance{splitfrac}{ptm}{1}{\NoValue}{2},
\splitfrac{34}{56},
\normalfont
\splitfrac{34}{56},
\UseTemplate{splitfrac}{text}{
  separator-font=ptm,
  separator-pre-kern=0pt,
  separator-post-kern=0pt}{34}{\NoValue}{15}
\end{document}
```

Figure 2: Example document for split level fractions using templates.

$^{34}/_{15}$, **½**, ½, ½, **$^{34}/_{56}$**, $^{34}/_{56}$, $^{34}/_{15}$

Figure 3: Output from running the example document shown in Figure 2.

Templates are an integral part of the design aspect in LaTeX3 and worth a closer look; see the documentation in [1]. We will not discuss them further here, but this brief introduction should be enough to give you an idea of the potential. At this point it should come as no surprise that our present page layout parameters are well suited for templates.

3.3 Putting the pieces together

We've now presented the tools and are ready to define templates for setting the page layouts. A template for setting all parameters is likely to be very large, so we'll just show a minimal example. We define a template type `pagelayout` which takes one argument: the name for the page type. Next we declare the template `minimal` which has two keys: `textheight` and `textwidth` with default values of 8 in and 6.5 in respectively (same as in the standard LaTeX class file minimal). The template then sets the local parameters for the page type and stores them in the token list pointer as shown Section 3.2.1. This simple template setup looks like this:

```
\DeclareTemplateType{pagelayout}{1}
\DeclareTemplate{pagelayout}{minimal}{1}{
  textheight = l [8in]
             \g_page_std_textheight_dim ,
  textwidth  = l [6.5in]
             \g_page_std_textwidth_dim ,
}{
  \DoParameterAssignments
  \dim_set:cn {l_page_#1_textheight_dim}
             {\g_page_std_textheight_dim}
  \dim_set:cn {l_page_#1_textwidth_dim}
             {\g_page_std_textwidth_dim}
  \page_store_page_dimensions:n {#1}
}
```

Using the template is fairly straightforward. The defaults are used unless you use a key in which case the new value is used. For demonstration purposes, we will make the recto and verso layouts different in height; the recto is one inch longer than the verso.

```
\UseTemplate{pagelayout}{minimal}
  { textheight= 9in } {minimal-recto}
\UseTemplate{pagelayout}
            {minimal}{}{minimal-verso}
\UsePageLayout{2}
    {minimal-recto,minimal-verso}
```

The `\UsePageLayout` command instructs LaTeX to switch between two different parameter sets which are then given as a comma separated list in the second argument. The command takes effect on the following page so it can also be used mid-document.

The attentive reader may have noticed that the interface presented above does not allow for defining two page layouts at a time with common margin ratios etc., as would usually be required. This will be supported, but it requires a different template type, taking two arguments. Alternatively, a package like geometry in its LaTeX3 incarnation could easily stick to the same user interface as it has now and simply pass the information on to two templates at a time if two-sided documents are produced.

4 Concluding remarks

The interface for page layouts described in this article exists only as an unreleased prototype at the time of writing. There is still a bit of work to do on it, especially in the area of integrating it with the experimental output routine xor. Currently xor stands at 5000+ lines so this is something which has to be done very carefully! When this is done it will be possible to specify page layouts on a per-page basis, which is especially useful for float pages.

Other than that we will produce various different templates plus provide tools for autocompletetion and error checking. When ready for release it will be put in our publically available Subversion (SVN) repository which can be reached by pointing your SVN client to

`http://www.latex-project.org/svnroot/experimental/trunk/`

References

[1] Various authors. LaTeX project web site directory for experimental code. `http://www.latex-project.org/svnroot/experimental/trunk`.

[2] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, Chris Rowley, Christine Detig, and Joachim Schrod. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 2004.

[3] Frank Neukam, Markus Kohm, Axel Kielhorn, and Jens-Uwe Morawski. The KOMA-Script Bundle, March 2005.

[4] Hideo Umeki. The `geometry` package, July 2002.