LaTeX 2ε for class and package writers

Copyright © 1995 The LaTeX3 Project
All rights reserved

12 December 1995

Contents

1 Introduction **2**
1.1 Writing classes and packages for LaTeX 2ε . . . 2
1.2 Overview . . . 3
1.3 Further information . . . 3

2 Writing classes and packages **4**
2.1 Old versions . . . 4
2.2 Using docstrip and doc . . . 4
2.3 Is it a class or a package? . . . 5
2.4 Command names . . . 5
2.5 Box commands and colour . . . 6
2.6 Defining text and math characters . . . 6
2.7 General style . . . 7

3 The structure of a class or package **9**
3.1 Identification . . . 10
3.2 Using classes and packages . . . 10
3.3 Declaring options . . . 11
3.4 Declarations . . . 13
3.5 Example: a local letter class . . . 14
3.6 Example: a newsletter class . . . 14

4 Commands for class and package writers **15**
4.1 Identification . . . 16
4.2 Loading files . . . 17

5 Option declaration **17**
5.1 Within option code commands . . . 18
5.2 Moving options around . . . 19
5.3 Delaying code . . . 20

5.4 Option processing 21
5.5 Safe File Commands 23
5.6 Reporting errors, etc 24
5.7 Defining commands 25
5.8 Moving arguments 26
5.9 Layout parameters 26
5.10 Case changing 27

6 Upgrading LaTeX 2.09 classes and packages 28
6.1 Try it first! 28
6.2 Troubleshooting 28
6.3 Font commands 28
6.4 Obsolete commands 30

1 Introduction

This document is an introduction to writing classes and packages for LaTeX, with special attention given to upgrading existing LaTeX 2.09 packages to LaTeX 2$_\varepsilon$. The latter subject is also covered in an article by Johannes Braams published in TUGboat 15.3.

1.1 Writing classes and packages for LaTeX 2$_\varepsilon$

LaTeX is a document preparation system that enables the document writer to concentrate on the contents of their text, without bothering too much about the formatting of it. For example, chapters are indicated by `\chapter{`⟨*title*⟩`}` rather than by selecting 18pt bold.

The file that contains the information about how to turn logical structure (like '`\chapter`') into formatting (like '18pt bold ragged right') is a *document class*. In addition, some features (such as colour or included graphics) are independent of the document class and these are contained in *packages*.

One of the largest differences between LaTeX 2.09 and LaTeX 2$_\varepsilon$ is in the commands used to write packages and classes. In LaTeX 2.09, there was very little support for writing `.sty` files, and so writers had to resort to using low-level commands.

LaTeX 2$_\varepsilon$ provides high-level commands for structuring packages. It is also much easier to build classes and packages on top of each other, for example writing a local technical report class `cetechr` (for the Chemical Engineering department) based on `article`.

1.2 Overview

This document contains an overview of how to write classes and packages for LaTeX. It does *not* introduce all of the commands necessary to write packages: these can be found in either *LaTeX: A Document Preparation System* or *The LaTeX Companion*. But it does describe the new commands for structuring classes and packages.

Section 2.7 contains some general advice about writing classes and packages. It describes the difference between classes and packages, the command naming conventions, the use of `doc` and `docstrip`, how TeX's primitive file and box commands interact with LaTeX. It also contains some hints about general LaTeX style.

Section 3 describes the structure of classes and packages. This includes building classes and packages on top of other classes and packages, declaring options and declaring commands. It also contains example classes.

Section 4 lists the new class and package commands.

Section 6 gives detailed advice on how to upgrade existing LaTeX 2.09 classes and packages to LaTeX 2_ε.

1.3 Further information

For a general introduction to LaTeX, including the new features of LaTeX 2_ε, you should read *LaTeX: A Document Preparation System* by Leslie Lamport [3].

A more detailed description of the new features of LaTeX, and an overview of more than 150 packages, is to be found in *The LaTeX Companion* by Michel Goossens, Frank Mittelbach and Alexander Samarin [1].

The LaTeX system is based on TeX, which is described in *The TeXbook* by Donald E. Knuth [2].

There are a number of documentation files which accompany every copy of LaTeX. A copy of *LaTeX News* will come out with each six-monthly release of LaTeX, and is found in the files `ltnews*.tex`. The author's guide *LaTeX 2_ε for Authors* describes the new LaTeX document features; it is in `usrguide.tex`. The guide *LaTeX 2_ε Font Selection* describes the LaTeX font selection scheme for class- and package-writers; it is in `fntguide.tex`. Configuring LaTeX is covered by the guide *Configuration options for LaTeX 2_ε* in `cfgguide.tex` whilst the philosophy behind our policy on modifying LaTeX is described in *Modifying LaTeX* in `modguide.tex`.

We are gradually turning the source code for LaTeX into a LaTeX document *LaTeX: the program*. This document includes an index of LaTeX commands and can be typeset from `source2e.tex`.

For more information about TEX and LATEX, please contact your local TEX Users Group, or the international TEX Users Group. Here are some useful addresses.

TEX Users Group, 1850 Union Street #1637, San Francisco, CA 94123, USA
Fax: +1 415 982 8559 Email: `tug@tug.org`

UK TUG, 1 Eymore Close, Selly Oak, Birmingham B29 4LB, UK
Fax: +44 121 476 2159 Email: `uktug-enquiries@tex.ac.uk`

2 Writing classes and packages

This section covers some general points concerned with writing LATEX classes and packages.

2.1 Old versions

If you are upgrading an existing LATEX 2.09 style file then we recommend freezing the 2.09 version and no longer maintaining it. Experience with the various dialects of LATEX which existed in the early 1990's suggests that maintaining packages for different versions of LATEX is almost impossible. It will, of course, be necessary for some organisations to maintain both versions in parallel for some time but this is not essential for those packages and classes supported by individuals: they should support only the new standard LATEX 2_ε, not obsolete versions of LATEX.

2.2 Using docstrip and doc

If you are going to write a large class or package for LATEX then you should consider using the `doc` software which comes with LATEX. LATEX classes and packages written using this can be processed in two ways: they can be run through LATEX, to produce documentation; and they can be processed with `docstrip`, to produce the `.cls` or `.sty` file.

The `doc` software can automatically generate indexes of definitions, indexes of command use, and change-log lists. It is very useful for maintaining and documenting large TEX sources.

The documented sources of the LATEX kernel itself, and of the standard classes, etc, are `doc` documents; they are in the `.dtx` files in the distribution. You can, in fact, typeset the source code of the kernel as one long document, complete with index, by running LATEX on `source2e.tex`.

For more information on `doc` and `docstrip`, consult the files `docstrip.dtx`, `doc.dtx`, and *The LATEX Companion*. For examples of its use, look at the `.dtx` files.

2.3 Is it a class or a package?

The first thing to do when you want to put some new LaTeX commands in a file is to decide whether it should be a *document class* or a *package*. The rule of thumb is:

> If the commands could be used with any document class, then make them a package; and if not, then make them a class.

There are two major types of class: those like `article`, `report` or `letter`, which are free-standing; and those which are extensions or variations of other classes—for example, the `proc` document class, which is built on the `article` document class.

Thus, a company might have a local `ownlet` class for printing letters with their own headed note-paper. Such a class would build on top of the existing `letter` class but it cannot be used with any other document class, so we have `ownlet.cls` rather than `ownlet.sty`.

The `graphics` package, in contrast, provides commands for including images into a LaTeX document. Since these commands can be used with any document class, we have `graphics.sty` rather than `graphics.cls`.

2.4 Command names

LaTeX has three types of command.

There are the author commands, such as `\section`, `\emph` and `\times`: most of these have short names, all in lower case.

There are also the class and package writer commands: most of these have long mixed-case names such as the following.

```
\InputIfFileExists  \RequirePackage  \PassOptionsToClass
```

Finally, there are the internal commands used in the LaTeX implementation, such as `\@tempcnta`, `\@ifnextchar` and `\@eha`: most of these commands contain `@` in their name, which means they cannot be used in documents, only in class and package files.

Unfortunately, for historical reasons the distinction between these commands is often blurred. For example, `\hbox` is an internal command which should only be used in the LaTeX kernel, whereas `\m@ne` is the constant -1 and could have been `\MinusOne`.

However, this rule of thumb is still useful: if a command has `@` in its name then it is not part of the supported LaTeX language—and its behaviour may change in future releases! If a command is mixed-case, or is described in *LaTeX: A Document Preparation System*, then you can rely on future releases of LaTeX 2_ε supporting the command.

2.5 Box commands and colour

Even if you do not intend to use colour in your own documents, by taking note of the points in this section you can ensure that your class or package is compatible with the `color` package. This may benefit people using your class or package who have access to colour printers.

The simplest way to ensure 'colour safety' is to always use LaTeX box commands rather than TeX primitives, that is use `\sbox` rather than `\setbox`, `\mbox` rather than `\hbox` and `\parbox` or the `minipage` environment rather than `\vbox`. The LaTeX box commands have new options which mean that they are now as powerful as the TeX primitives.

As an example of what can go wrong, consider that in `{\ttfamily` ⟨*text*⟩`}` the font is restored just *before* the `}`, whereas in the similar looking construction `{\color{green}` ⟨*text*⟩`}` the colour is restored just *after* the final `}`. Normally this distinction does not matter at all; but consider a primitive TeX box assignment such as:

```
\setbox0=\hbox{\color{green} ⟨text⟩}
```

Now the colour-restore occurs after the `}` and so is *not* stored in the box. Exactly what bad effects this can have depends on how colour is implemented: it can range from getting the wrong colours in the rest of the document, to causing errors in the dvi-driver used to print the document.

Also of interest is the command `\normalcolor`. This is normally just `\relax` (i.e., does nothing) but you can use it rather like `\normalfont` to set regions of the page such as captions or section headings to the 'main document colour'.

2.6 Defining text and math characters

Because LaTeX 2ε supports different encodings, definitions of commands for producing symbols, accents, composite glyphs, etc. must be defined using the commands provided for this purpose and described in *LaTeX 2ε Font Selection*. This part of the system is still under development so such tasks should be undertaken with great caution.

Also, `\DeclareRobustCommand` should be used for encoding-independent commands of this type.

Note that it is no longer possible to refer to the math font set-up outside math mode: for example, neither `\textfont 1` nor `\scriptfont 2` are guaranteed to be defined in other modes.

2.7 General style

The new system provides many commands designed to help you produce well-structured class and package files that are both robust and portable. This section outlines some ways to make intelligent use of these.

2.7.1 Loading other files

LaTeX provides these commands: New description 1995/12/01

```
\LoadClass          \LoadClassWithOptions
\RequirePackage     \RequirePackageWithOptions
```

for using classes or packages inside other classes or packages. We recommend strongly that you use them, rather than the primitive `\input` command, for a number of reasons.

Files loaded with `\input` ⟨*filename*⟩ will not be listed in the `\listfiles` list.

If a package is always loaded with `\RequirePackage...` or `\usepackage` then, even if its loading is requested several times, it will be loaded only once. By contrast, if it is loaded with `\input` then it can be loaded more than once; such an extra loading may waste time and memory and it may produce strange results.

If a package provides option-processing then, again, strange results are possible if the package is `\input` rather than loaded by means of `\usepackage` or `\RequirePackage...`.

If the package `foo.sty` loads the package `baz.sty` by use of `\input baz.sty` then the user will get a warning:

```
LaTeX Warning: You have requested package `foo',
               but the package provides `baz'.
```

Thus, for several reasons, using `\input` to load packages is not a good idea.

Unfortunately, if you are upgrading the file `myclass.sty` to a class file then you have to make sure that any old files which contain `\input myclass.sty` still work.

This was also true for the standard classes (`article`, `book` and `report`), since a lot of existing LaTeX 2.09 document styles contain `\input article.sty`. The approach which we use to solve this is to provide minimal files `article.sty`, `book.sty` and `report.sty`, which simply load the appropriate class files.

For example, `article.sty` contains just the following lines:

```
\NeedsTeXFormat{LaTeX2e}
\@obsoletefile{article.sty}{article.cls}
\LoadClass{article}
```

You may wish to do the same or, if you think that it is safe to do so, you may decide to just remove `myclass.sty`.

2.7.2 Make it robust

We consider it good practice, when writing packages and classes, to use LaTeX commands as much as possible.

Thus, instead of using `\def...` we recommend using one of `\newcommand`, `\renewcommand` or `\providecommand`; `\CheckCommand` is also useful. Doing this makes it less likely that you will inadvertently redefine a command, giving unexpected results.

When you define an environment, use `\newenvironment` or `\renewenvironment` instead `\def\foo{...}` and `\def\endfoo{...}`.

If you need to set or change the value of a ⟨*dimen*⟩ or ⟨*skip*⟩ register, use `\setlength`.

To manipulate boxes, use LaTeX commands such as `\sbox`, `\mbox` and `\parbox` rather than `\setbox`, `\hbox` and `\vbox`.

Use `\PackageError`, `\PackageWarning` or `\PackageInfo` (or the equivalent class commands) rather than `\@latexerr`, `\@warning` or `\wlog`.

It is still possible to declare options by defining `\ds@`⟨*option*⟩ and calling `\@options`; but we recommend the `\DeclareOption` and `\ProcessOptions` commands instead. These are more powerful and use less memory. So rather than using:

```
\def\ds@draft{\overfullrule 5pt}
\@options
```

you should use:

```
\DeclareOption{draft}{\setlength{\overfullrule}{5pt}}
\ProcessOptions\relax
```

The advantage of this kind of practice is that your code is more readable and, more important, that it is less likely to break when used with future versions of LaTeX.

2.7.3 Make it portable

It is also sensible to make your files are as portable as possible. To ensure this; they should contain only visible 7-bit text; and the filenames should contain at most eight characters (plus the three letter extension). Also, of course, it **must not** have the same name as a file in the standard LaTeX distribution, however similar its contents may be to one of these files.

It is also useful if local classes or packages have a common prefix, for example the University of Nowhere classes might begin with `unw`. This helps to avoid every University having its own thesis class, all called `thesis.cls`.

If you rely on some features of the LaTeX kernel, or on a package, please specify the release-date you need. For example, the package error commands were introduced in the June 1994 release so, if you use them then you should put:

```
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
```

2.7.4 Useful hooks

Some packages and document styles had to redefine the command `\document` or `\enddocument` to achieve their goal. This is no longer necessary. You can now use the 'hooks' `\AtBeginDocument` and `\AtEndDocument` (see Section 5.3). Again, using these hooks makes it less likely that your code breaks with future versions of LaTeX. It also makes it more likely that your package can work together with someone else's.

3 The structure of a class or package

LaTeX 2$_\varepsilon$ classes and packages have more structure than LaTeX 2.09 style files did. The outline of a class or package file is:

Identification The file says that it is a LaTeX 2$_\varepsilon$ package or class, and gives a short description of itself.

Preliminary declarations Here the file declares some commands and can also load other files. Usually these commands will be just those needed for the code used in the declared options.

Options The file declares and processes its options.

More declarations This is where the file does most of its work: declaring new variables, commands and fonts; and loading other files.

3.1 Identification

The first thing a class or package file does is identify itself. Package files do this as follows:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{⟨package⟩}[⟨date⟩ ⟨other information⟩]
```

For example:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{latexsym}[1994/06/01 Standard LaTeX package]
```

Class files do this as follows:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{⟨class-name⟩}[⟨date⟩ ⟨other information⟩]
```

For example:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{article}[1994/06/01 Standard LaTeX class]
```

The ⟨*date*⟩ should be given in the form 'YYYY/MM/DD'. This date is checked whenever a user specifies a date in their `\documentclass` or `\usepackage` command. For example, if you wrote:

```
\documentclass{article}[1995/12/23]
```

then users at a different location would get a warning that their copy of `article` was out of date.

The description of a class is displayed when the class is used. The description of a package is put into the log file. These descriptions are also displayed by the `\listfiles` command. The phrase `Standard LaTeX` **must not** be used in the identification banner of any file other than those in the standard LaTeX distribution.

3.2 Using classes and packages

The first major difference between LaTeX 2.09 style files and LaTeX 2_ε packages and classes is that LaTeX 2_ε supports *modularity*, in the sense of building files from small building-blocks rather than using large single files.

A LaTeX package or class can load a package as follows:

```
\RequirePackage[⟨options⟩]{⟨package⟩}[⟨date⟩]
```

For example:

```
\RequirePackage{ifthen}[1994/06/01]
```

This command has the same syntax as the author command `\usepackage`. It allows packages or classes to use features provided by other packages. For example, by loading the `ifthen` package, a package writer can use the 'if... then... else...' commands provided by that package.

A LaTeX class can load one other class as follows:

```
\LoadClass[⟨options⟩]{⟨class-name⟩}[⟨date⟩]
```

For example:

```
\LoadClass[twocolumn]{article}
```

This command has the same syntax as the author command `\documentclass`. It allows classes to be based on the syntax and appearance of another class. For example, by loading the `article` class, a class writer only has to change the bits of `article` they don't like, rather than writing a new class from scratch.

New feature 1995/12/01

The following commands can be used in the common case that you want to simply load a class or package file with exactly those options that are being used by the current class.

```
\LoadClassWithOptions{⟨class-name⟩}[⟨date⟩]
\RequirePackageWithOptions{⟨package⟩}[⟨date⟩]
```

For example:

```
\LoadClassWithOptions{article}
\RequirePackageWithOptions{graphics}[1995/12/01]
```

3.3 Declaring options

The other major difference between LaTeX 2.09 styles and LaTeX 2_ε packages and classes is in option handling. Packages and classes can now declare options and these can be specified by authors; for example, the `twocolumn` option is declared by the `article` class.

An option is declared as follows:

```
\DeclareOption{⟨option⟩}{⟨code⟩}
```

For example, the `dvips` option (slightly simplified) to the `graphics` package is implemented as:

```
\DeclareOption{dvips}{\input{dvips.def}}
```

This means that when an author writes `\usepackage[dvips]{graphics}`, the file `dvips.def` is loaded. As another example, the `a4paper` option is declared in the `article` class to set the `\paperheight` and `\paperwidth` lengths:

```
\DeclareOption{a4paper}{%
   \setlength{\paperheight}{297mm}%
   \setlength{\paperwidth}{210mm}%
}
```

Sometimes a user will request an option which the class or package has not explicitly declared. By default this will produce a warning (for classes) or error (for packages); this behaviour can be altered as follows:

```
\DeclareOption*{⟨code⟩}
```

For example, to make the package `fred` produce a warning rather than an error for unknown options, you could specify:

```
\DeclareOption*{%
   \PackageWarning{fred}{Unknown option `\CurrentOption'}%
}
```

Then, if an author writes `\usepackage[foo]{fred}`, they will get a warning `Package fred Warning: Unknown option 'foo'.` As another example, the `fontenc` package tries to load a file ⟨*ENC*⟩`enc.def` whenever the ⟨*ENC*⟩ option is used. This can be done by writing:

```
\DeclareOption*{%
   \input{\CurrentOption enc.def}%
}
```

It is possible to pass options on to another package or class, using the command `\PassOptionsToPackage` or `\PassOptionsToClass`. For example, to pass every unknown option on to the `article` class, you can use:

```
\DeclareOption*{%
   \PassOptionsToClass{\CurrentOption}{article}%
}
```

If you do this then you should make sure you load the class at some later point, otherwise the options will never be processed!

So far, we have explained only how to declare options, not how to execute them. To process the options with which the file was called, you should use:

```
\ProcessOptions\relax
```

This executes the ⟨*code*⟩ for each option that was both specified and declared (see Section 5.4 for details of how this is done).

For example, if the `jane` package file contains:

```
\DeclareOption{foo}{\typeout{Saw foo.}}
\DeclareOption{baz}{\typeout{Saw baz.}}
\DeclareOption*{\typeout{What's \CurrentOption?}}
\ProcessOptions\relax
```

and an author writes `\usepackage[foo,bar]{jane}`, then they will see the messages `Saw foo.` and `What's bar?`

3.4 Declarations

Most of the work of a class or package is in defining new commands, or changing the appearance of documents. This is done in the body of the package, using commands such as `\newcommand` or `\setlength`.

LaTeX 2ε provides several new commands to help class and package writers; these are described in detail in Section 4.

There are three things that every class file *must* contain: these are a definition of `\normalsize` and values for `\textwidth` and `\textheight`. So a minimal document class file[1] looks like this:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{minimal}[1995/10/30 Standard LaTeX minimal class]
\renewcommand{\normalsize}{\fontsize{10pt}{12pt}\selectfont}
\setlength{\textwidth}{6.5in}
\setlength{\textheight}{8in}
```

However, this class file will not support footnotes, marginals, floats, etc. nor will it provide any of the 2-letter font commands such as `\rm`; thus most classes will contain more than this minimum!

[1] This class is now in the standard distribution, as `minimal.cls`.

3.5 Example: a local letter class

A company may have its own letter class, for setting letters in the company style. This section shows a simple implementation of such a class, although a real class would need more structure.

The class begins by announcing itself as `neplet.cls`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{neplet}[1995/04/01 NonExistent Press letter class]
```

Then this next bit passes any options on to the `letter` class, which is loaded with the `a4paper` option.

```
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{letter}}
\ProcessOptions\relax
\LoadClass[a4paper]{letter}
```

In order to use the company letter head, it redefines the `firstpage` page style: this is the page style that is used on the first page of letters.

```
\renewcommand{\ps@firstpage}{%
   \renewcommand{\@oddhead}{⟨letterhead goes here⟩}%
   \renewcommand{\@oddfoot}{⟨letterfoot goes here⟩}%
}
```

And that's it!

3.6 Example: a newsletter class

A simple newsletter can be typeset with LaTeX, using a variant of the `article` class. The class begins by announcing itself as `smplnews.cls`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{smplnews}[1995/04/01 The Simple News newsletter class]

\newcommand{\headlinecolor}{\normalcolor}
```

It passes most specified options on to the `article` class: apart from the `onecolumn` option, which is switched off, and the `green` option, which sets the headline in green.

```
\DeclareOption{onecolumn}{\OptionNotUsed}
\DeclareOption{green}{\renewcommand{\headlinecolor}{\color{green}}}

\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}

\ProcessOptions\relax
```

It then loads the class `article` with the option `twocolumn`.

```
\LoadClass[twocolumn]{article}
```

Since the newsletter is to be printed in colour, it now loads the `color` package. The class does not specify a device driver option since this should be specified by the user of the `smplnews` class.

```
\RequirePackage{color}
```

The class then redefines `\maketitle` to produce the title in 72pt Helvetica bold oblique, in the appropriate colour.

```
\renewcommand{\maketitle}{%
   \twocolumn[%
      \fontsize{72}{80}\fontfamily{phv}\fontseries{b}%
      \fontshape{sl}\selectfont\headlinecolor
      \@title
   ]%
}
```

It redefines `\section` and switches off section numbering.

```
\renewcommand{\section}{%
   \@startsection
      {section}{1}{0pt}{-1.5ex plus -1ex minus -.2ex}%
      {1ex plus .2ex}{\large\sffamily\slshape\headlinecolor}%
}

\setcounter{secnumdepth}{0}
```

It also sets the three essential things.

```
\renewcommand{\normalsize}{\fontsize{9}{10}\selectfont}
\setlength{\textwidth}{17.5cm}
\setlength{\textheight}{25cm}
```

In practice, a class would need more than this: it would provide commands for issue numbers, authors of articles, page styles and so on; but this skeleton gives a start. The `ltnews` class file is not much more complex than this one.

4 Commands for class and package writers

This section describes briefly each of the new commands for class and package writers. To find out about other aspects of the new system, you should also read *LaTeX: A Document Preparation System*, *The LaTeX Companion* and *LaTeX 2ε for Authors*.

4.1 Identification

The first group of commands discussed here are those used to identify your class or package file.

`\NeedsTeXFormat {`⟨*format-name*⟩`} [`⟨*release-date*⟩`]`

This command tells TEX that this file should be processed using a format with name ⟨*format-name*⟩. You can use the optional argument ⟨*release-date*⟩ to further specify the earliest release date of the format that is needed. When the release date of the format is older than the one specified a warning will be generated. The standard ⟨*format-name*⟩ is `LaTeX2e`. The date, if present, must be in the form YYYY/MM/DD.

Example:

```
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
```

`\ProvidesClass {`⟨*class-name*⟩`} [`⟨*release-info*⟩`]`
`\ProvidesPackage {`⟨*package-name*⟩`} [`⟨*release-info*⟩`]`

This declares that the current file contains the definitions for the document class ⟨*class-name*⟩ or package ⟨*package-name*⟩.

The optional ⟨*release-info*⟩, if used, must contain:

- the release date of this version of the file, in the form YYYY/MM/DD;
- optionally followed by a space and a short description, possibly including a version number.

The above syntax should be followed exactly so that this information can be used by `\LoadClass` or `\documentclass` (for classes) or `\RequirePackage` or `\usepackage` (for packages) to test that the release is not too old.

The whole of this ⟨*release-info*⟩ information is displayed by `\listfiles` and should therefore not be too long.

Example:

```
\ProvidesClass{article}[1994/06/01 v1.0 Standard LaTeX class]
\ProvidesPackage{ifthen}[1994/06/01 v1.0 Standard LaTeX package]
```

`\ProvidesFile {`⟨*file-name*⟩`} [`⟨*release-info*⟩`]`

This is similar to the two previous commands except that here the full filename, including the extension, must be given. It is used for declaring any files other than main class and package files.

Example:

```
\ProvidesFile{T1enc.def}[1994/06/01 v1.0 Standard LaTeX file]
```

Note that the phrase `Standard LaTeX` **must not** be used in the identification banner of any file other than those in the standard LaTeX distribution.

4.2 Loading files

This group of commands can be used to create your own document class or package by building on existing classes or packages. New feature 1995/12/01

`\RequirePackage` [⟨*options-list*⟩] {⟨*package-name*⟩} [⟨*release-info*⟩]
`\RequirePackageWithOptions` {⟨*package-name*⟩} [⟨*release-info*⟩]

Packages and classes should use these commands to load other packages.

The use of `\RequirePackage` is the same as the author command `\usepackage`.

Examples:

```
\RequirePackage{ifthen}[1994/06/01]
\RequirePackageWithOptions{graphics}[1995/12/01]
```

`\LoadClass` [⟨*options-list*⟩] {⟨*class-name*⟩} [⟨*release-info*⟩]
`\LoadClassWithOptions` {⟨*class-name*⟩} [⟨*release-info*⟩]

These commands are for use *only* in class files, they cannot be used in packages files; they can be used at most once within a class file. New feature 1995/12/01

The use of`\LoadClass` is the same as the use of `\documentclass` to load a class file.

Examples:

```
\LoadClass{article}[1994/06/01]
\LoadClassWithOptions{article}[1995/12/01]
```

The two `WithOptions` versions simply load the class (or package) file with exactly those options that are being used by the current file (class or package). See below, in 5.2, for further discussion of their use. New feature 1995/12/01

5 Option declaration

The following commands deal with the declaration and handling of options to document classes and packages.

There are some commands designed especially for use within the ⟨*code*⟩ argument of these commands (see below).

`\DeclareOption` {⟨*option-name*⟩} {⟨*code*⟩}

This makes ⟨*option-name*⟩ a 'declared option' of the class or package in which it is put.

The ⟨*code*⟩ argument contains the code to be executed if that option is specified for the class or package; it can contain any valid LaTeX 2_ε construct.

Example:

```
\DeclareOption{twoside}{\@twosidetrue}
```

`\DeclareOption*` {⟨*code*⟩}

This declares the ⟨*code*⟩ to be executed for every option which is specified for, but otherwise not explicitly declared by, the class or package; this code is called the 'default option code' and it can contain any valid LaTeX 2_ε construct.

If a class file contains no `\DeclareOption*` then, by default, all specified but undeclared options for that class will be silently passed to all packages (as will the specified and declared options for that class).

If a package file contains no `\DeclareOption*` then, by default, each specified but undeclared option for that package will produce an error.

5.1 Within option code commands

These two commands can be used only within the ⟨*code*⟩ argument of either `\DeclareOption` or `\DeclareOption*`. Other commands commonly used within these arguments can be found in the next few subsections.

`\CurrentOption`

This expands to the name of the current option.

`\OptionNotUsed`

This causes the current option to be added to the list of 'unused options'.

You can now include hash marks (`#`) within these ⟨*code*⟩ arguments without special treatment (formerly, it had been necessary to double them). New feature 1995/06/01

5.2 Moving options around

These two commands are also very useful within the ⟨*code*⟩ argument of `\DeclareOption` or `\DeclareOption*`:

`\PassOptionsToPackage` {⟨*options-list*⟩} {⟨*package-name*⟩}
`\PassOptionsToClass` {⟨*options-list*⟩} {⟨*class-name*⟩}

The command `\PassOptionsToPackage` passes the options in ⟨*options-list*⟩ to package ⟨*package-name*⟩. This means that it adds the ⟨*option-list*⟩ to the list of options used by any future `\RequirePackage` or `\usepackage` command for package ⟨*package-name*⟩.

Example:

```
\PassOptionsToPackage{foo,bar}{fred}
\RequirePackage[baz]{fred}
```

is the same as:

```
\RequirePackage[foo,bar,baz]{fred}
```

Similarly, `\PassOptionsToClass` may be used in a class file to pass options to another class to be loaded with `\LoadClass`.

New description 1995/12/01

The effects and use of these two commands should be contrasted with those of the following two (documented above, in 4.2):

```
\LoadClassWithOptions
\RequirePackageWithOptions
```

The command`\RequirePackageWithOptions` is similar to `\RequirePackage`, but it always loads the required package with exactly the same option list as that being used by the current class or package, rather than with any option explicitly supplied or passed on by `\PassOptionsToPackage`.

The main purpose of `\LoadClassWithOptions` is to allow one class to simply build on another, for example:

```
\LoadClassWithOptions{article}
```

This should be compared with the slightly different construction

```
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax
\LoadClass{article}
```

As used above, the effects are more or less the same, but the first is a lot less to type; also the `\LoadClassWithOptions` method runs slightly quicker.

If, however, the class declares options of its own then the two constructions are different. Compare, for example:

```
\DeclareOption{landscape}{\@landscapetrue}
\ProcessOptions\relax
\LoadClassWithOptions{article}
```

with:

```
\DeclareOption{landscape}{\@landscapetrue}
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax
\LoadClass{article}
```

In the first example, the article class will be loaded with option `landscape` precisely when the current class is called with this option. By contrast, in the second example it will never be called with option `landscape` as in that case article is passed options only by the default option handler, but this handler is not used for `landscape` because that option is explicitly declared.

5.3 Delaying code

These first two commands are also intended primarily for use within the ⟨*code*⟩ argument of `\DeclareOption` or `\DeclareOption*`.

`\AtEndOfClass` {⟨*code*⟩}
`\AtEndOfPackage` {⟨*code*⟩}

These commands declare ⟨*code*⟩ that is saved away internally and then executed after processing the whole of the current class or package file.

Repeated use of these commands is permitted: the code in the arguments is stored (and later executed) in the order of their declarations.

`\AtBeginDocument` {⟨*code*⟩}
`\AtEndDocument` {⟨*code*⟩}

These commands declare ⟨*code*⟩ to be saved internally and executed while LaTeX is executing `\begin{document}` or `\end{document}`.

The ⟨*code*⟩ specified in the argument to `\AtBeginDocument` is executed near the end of the `\begin{document}` code, *after* the font selection tables have been set up. It is therefore a useful place to put code which needs to be executed after

everything has been prepared for typesetting and when the normal font for the document is the current font.

The `\AtBeginDocumnt` hook should not be used for code that does any typesetting since the typeset result would be unpredictable.

New description 1995/12/01

The ⟨*code*⟩ specified in the argument to `\AtEndDocument` is executed at the beginning of the `\end{document}` code, *before* the final page is finished and before any leftover floating environments are processed. If some of the ⟨*code*⟩ is to be executed after these two processes, you should include a `\clearpage` at the appropriate point in ⟨*code*⟩.

Repeated use of these commands is permitted: the code in the arguments is stored (and later executed) in the order of their declarations.

`\AtBeginDvi{`⟨*specials*⟩`}`

New feature 1994/12/01

These commands save in a box register things which are written to the `.dvi` file at the beginning of the 'shipout' of the first page of the document.

This should not be used for anything that will add any typeset material to the `.dvi` file.

Repeated use of this command is permitted.

5.4 Option processing

`\ProcessOptions`

This command executes the ⟨*code*⟩ for each selected option.

We shall first describe how `\ProcessOptions` works in a package file, and then how this differs in a class file.

To understand in detail what `\ProcessOptions` does in a package file, you have to know the difference between *local* and *global* options.

- **Local options** are those which have been explicitly specified for this particular package in the ⟨*options*⟩ argument of any of these:

  `\PassOptionsToPackage{`⟨*options*⟩`}` `\usepackage[`⟨*options*⟩`]`
  `\RequirePackage[`⟨*options*⟩`]`

- **Global options** are any other options that are specified by the author in the ⟨*options*⟩ argument of `\documentclass[`⟨*options*⟩`]`.

For example, suppose that a document begins:

```
\documentclass[german,twocolumn]{article}
\usepackage{gerhardt}
```

whilst package `gerhardt` calls package `fred` with:

```
\PassOptionsToPackage{german,dvips,a4paper}{fred}
\RequirePackage[errorshow]{fred}
```

then:

- `fred`'s local options are `german`, `dvips`, `a4paper` and `errorshow`;
- `fred`'s only global option is `twocolumn`.

When `\ProcessOptions` is called, the following happen.

- *First*, for each option so far declared in `fred.sty` by `\DeclareOption`, it looks to see if that option is either a global or a local option for `fred`: if it is then the corresponding code is executed.

  This is done in the order in which these options were declared in `fred.sty`.

- *Then*, for each remaining *local* option, the command `\ds@`⟨*option*⟩ is executed if it has been defined somewhere (other than by a `\DeclareOption`); otherwise, the 'default option code' is executed. If no default option code has been declared then an error message is produced.

  This is done in the order in which these options were specified.

Throughout this process, the system ensures that the code declared for an option is executed at most once.

Returning to the example, if `fred.sty` contains:

```
\DeclareOption{dvips}{\typeout{DVIPS}}
\DeclareOption{german}{\typeout{GERMAN}}
\DeclareOption{french}{\typeout{FRENCH}}
\DeclareOption*{\PackageWarning{fred}{Unknown `\CurrentOption'}}
\ProcessOptions\relax
```

then the result of processing this document will be:

```
DVIPS
GERMAN
Package fred Warning: Unknown `a4paper'.
Package fred Warning: Unknown `errorshow'.
```

Note the following:

- the code for the `dvips` option is executed before that for the `german` option, because that is the order in which they are declared in `fred.sty`;
- the code for the `german` option is executed only once, when the declared options are being processed;
- the `a4paper` and `errorshow` options produce the warning from the code declared by `\DeclareOption*` (in the order in which they were specified), whilst the `twocolumn` option does not: this is because `twocolumn` is a global option.

In a class file, `\ProcessOptions` works in the same way, except that: *all* options are local; and the default value for `\DeclareOption*` is `\OptionNotUsed` rather than an error.

Note that, because `\ProcessOptions` has a `*`-form, it is wise to follow the non-star form with `\relax`, as in the previous examples, since this prevents unnecessary look ahead and possibly misleading error messages being issued.

New description 1995/12/01

`\ProcessOptions* \@options`

This is like `\ProcessOptions` but it executes the options in the order specified in the calling commands, rather than in the order of declaration in the class or package. For a package this means that the global options are processed first.

The `\@options` command from LaTeX 2.09 has been made equivalent to this in order to ease the task of updating old document styles to LaTeX 2_ε class files.

`\ExecuteOptions {`⟨*options-list*⟩`}`

For each option in the ⟨*options-list*⟩, in order, this command simply executes the command `\ds@`⟨*option*⟩ (if this command is not defined, then that option is silently ignored).

It can be used to provide a 'default option list' just before `\ProcessOptions`. For example, suppose that in a class file you want to set up the default design to be: two-sided printing; 11pt fonts; in two columns. Then it could specify:

```
\ExecuteOptions{11pt,twoside,twocolumn}
```

5.5 Safe File Commands

These commands deal with file input; they ensure that the non-existence of a requested file can be handled in a user-friendly way.

`\IfFileExists` {⟨*file-name*⟩} {⟨*true*⟩} {⟨*false*⟩}

If the file exists then the code specified in ⟨*true*⟩ is executed.

If the file does not exist then the code specified in ⟨*false*⟩ is executed.

This command does *not* input the file.

`\InputIfFileExists` {⟨*file-name*⟩} {⟨*true*⟩} {⟨*false*⟩}

This inputs the file ⟨*file-name*⟩ if it exists and, immediately before the input, the code specified in ⟨*true*⟩ is executed.

If the file does not exist then the code specified in ⟨*false*⟩ is executed.

It is implemented using `\IfFileExists`.

5.6 Reporting errors, etc

These commands should be used by third party classes and packages to report errors, or to provide information to authors.

`\ClassError` {⟨*class-name*⟩} {⟨*error-text*⟩} {⟨*help-text*⟩}
`\PackageError` {⟨*package-name*⟩} {⟨*error-text*⟩} {⟨*help-text*⟩}

These produce an error message. The ⟨*error-text*⟩ is displayed and the `?` error prompt is shown. If the user types `h`, they will be shown the ⟨*help-text*⟩.

Within the ⟨*error-text*⟩ and ⟨*help-text*⟩: `\protect` can be used to stop a command from expanding; `\MessageBreak` causes a line-break; and `\space` prints a space.

Note that the ⟨*error-text*⟩ will have a full stop added to it, so do not put one into the argument.

For example:

```
\newcommand{\foo}{FOO}
\PackageError{ethel}{%
   Your hovercraft is full of eels,\MessageBreak
   and \protect\foo\space is \foo
}{%
   Oh dear! Something's gone wrong.\MessageBreak
   \space \space Try typing \space <return>
   \space to proceed, ignoring \protect\foo.
}
```

produces this display:

```
! Package ethel Error: Your hovercraft is full of eels,
(ethel)               and \foo is FOO.

See the ethel package documentation for explanation.
```

If the user types `h`, this will be shown:

```
Oh dear! Something's gone wrong.
  Try typing  <return>  to proceed, ignoring \foo.
```

`\ClassWarning` {⟨*class-name*⟩} {⟨*warning-text*⟩}
`\PackageWarning` {⟨*package-name*⟩} {⟨*warning-text*⟩}
`\ClassWarningNoLine` {⟨*class-name*⟩} {⟨*warning-text*⟩}
`\PackageWarningNoLine` {⟨*package-name*⟩} {⟨*warning-text*⟩}
`\ClassInfo` {⟨*class-name*⟩} {⟨*info-text*⟩}
`\PackageInfo` {⟨*package-name*⟩} {⟨*info-text*⟩}

The four `Warning` commands are similar to the error commands, except that they produce only a warning on the screen, with no error prompt.

The first two, `Warning` versions, also show the line number where the warning occurred, whilst the second two, `WarningNoLine` versions, do not.

The two `Info` commands are similar except that they log the information only in the transcript file, including the line number. There are no `NoLine` versions of these two.

Within the ⟨*warning-text*⟩ and ⟨*info-text*⟩: `\protect` can be used to stop a command from expanding; `\MessageBreak` causes a line-break; and `\space` prints a space. Also, these should not end with a full stop as one is automatically added.

5.7 Defining commands

LaTeX 2ε provides some extra methods of (re)defining commands that are intended for use in class and package files.

The `*`-forms of these commands should be used to define commands that are not, in TeX terms, long. This can be useful for error-trapping with commands whose arguments are not intended to contain whole paragraphs of text.

New feature 1994/12/01

`\DeclareRobustCommand` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}
`\DeclareRobustCommand*` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}

This command takes the same arguments as `\newcommand` but it declares a robust command, even if some code within the⟨*definition*⟩ is fragile. You can use this command to define new robust commands, or to redefine existing commands

and make them robust. A log is put into the transcript file if a command is redefined.

For example, if `\seq` is defined as follows:

```
\DeclareRobustCommand{\seq}[2][n]{%
  \ifmmode
    #1_{1}\ldots#1_{#2}%
  \else
    \PackageWarning{fred}{You can't use \protect\seq\space in text}%
  \fi
}
```

Then the command `\seq` can be used in moving arguments, even though `\ifmmode` cannot, for example:

```
\section{Stuff about sequences $\seq{x}$}
```

Note also that there is no need to put a `\relax` before the `\ifmmode` at the beginning of the definition; this is because the protection given by this `\relax` against expansion at the wrong time will be provided internally.

`\CheckCommand` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}
`\CheckCommand*` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}

This takes the same arguments as `\newcommand` but, rather than define ⟨*cmd*⟩, it just checks that the current definition of ⟨*cmd*⟩ is exactly as given by ⟨*definition*⟩. An error is raised if these definitions differ.

This command is useful for checking the state of the system before your package starts altering the definitions of commands. It allows you to check, in particular, that no other package has redefined the same command.

5.8 Moving arguments

New description 1994/12/01

The setting of protect whilst processing (i.e. moving) moving arguments has been reimplemented, as has the method of writing information from the `.aux` file to other files such as the `.toc` file. Details can be found in the file `ltdefns.dtx`.

We hope that these changes will not affect many packages.

5.9 Layout parameters

`\paperheight`
`\paperwidth`

These two parameters are usually set by the class to be the size of the paper being used. This should be actual paper size, unlike `\textwidth` and `\textheight`

which are the size of the main text body within the margins.

5.10 Case changing

`\MakeUppercase` {⟨*text*⟩}
`\MakeLowercase` {⟨*text*⟩}

TEX provides two primitives `\uppercase` and `\lowercase` for changing the case of text. These are sometimes used in document classes, for example to set information in running heads in all capitals.

New feature 1995/06/01

Unfortunately, these TEX primitives do not change the case of characters accessed by commands like `\ae` or `\aa`. To overcome this problem, LATEX provides two new commands `\MakeUppercase` and `\MakeLowercase` to do this.

For example:

`\uppercase{aBcD\ae\AA\ss\OE}`	ABCDæÅßŒ
`\lowercase{aBcD\ae\AA\ss\OE}`	abcdæÅßŒ
`\MakeUppercase{aBcD\ae\AA\ss\OE}`	ABCDÆÅSSŒ
`\MakeLowercase{aBcD\ae\AA\ss\OE}`	abcdæåßœ

The commands `\MakeUppercase` and `\MakeLowercase` themselves are robust, but they have moving arguments.

The commands use the TEX primitives `\uppercase` and `\lowercase`, and so have a number of unexpected 'features'. In particular, they change the case of everything (except characters in the names of control-sequences) in their text argument: this includes mathematics, environment names, and label names.

For example:

```
\MakeUppercase{$x+y$ in \ref{foo}}
```

produces $X + Y$ and the warning:

```
LaTeX Warning: Reference `FOO' on page ... undefined on ...
```

In the long run, we would like to use all-caps fonts rather than any command like `\MakeUppercase` but this is not possible at the moment because such fonts do not exist.

In order that upper/lower-casing will work reasonably well, and in order to provide any correct hyphenation, LATEX 2$_\varepsilon$ *must* use, throughout a document, the same fixed table for changing case. The table used is designed for the font encoding `T1`; this works well with the standard TEX fonts for all Latin alphabets but will cause problems when using other alphabets.

New description 1995/12/01

6 Upgrading LaTeX 2.09 classes and packages

This section describes the changes you may need to make when you upgrade an existing LaTeX style to a package or class but we shall start in optimistic mode.

Many existing style files will run with LaTeX 2$_\varepsilon$ without any modification to the file itself. When everything is running OK, please put a note in the newly created package or class file to record that it runs with the new standard LaTeX; then distribute it to your users.

6.1 Try it first!

The first thing you should do is to test your style in 'compatibility mode'. The only change you need to make in order to do this is, possibly, to change the extension of the file to `.cls`: you should make this change only if your file was used as a main document style. Now, without any other modifications, run LaTeX 2$_\varepsilon$ on a document that uses your file. This assumes that you have a suitable collection of files that tests all the functionality provided by your style file. (If you haven't, now is the time to make one!)

You now need to change the test document files so that they are LaTeX 2$_\varepsilon$ documents: see *LaTeX 2$_\varepsilon$ for Authors* for details of how to do this and then try them again. You have now tried the test documents in both LaTeX 2$_\varepsilon$ native mode and LaTeX 2.09 compatibility mode.

6.2 Troubleshooting

If your file does not work with LaTeX 2$_\varepsilon$, there are two likely reasons.

- LaTeX now has a robust, well-defined designer's interface for selecting fonts, which is very different from the LaTeX 2.09 internals.
- Your style file may have used some LaTeX 2.09 internal commands which have changed, or which have been removed.

When you are debugging your file, you will probably need more information than is normally displayed by LaTeX 2$_\varepsilon$. This is achieved by resetting the counter `errorcontextlines` from its default value of -1 to a much higher value, e.g. 999.

6.3 Font commands

Some font and size commands are now defined by the document class rather than by the LaTeX kernel. If you are upgrading a LaTeX 2.09 document style to

a class that does not load one of the standard classes, then you will probably need to add definitions for these commands.

`\rm \sf \tt \bf \it \sl \sc`

None of these short-form font selection commands are defined in the LaTeX 2_ε kernel. They are defined by all the standard class files.

If you want to define them in your class file, there are several reasonable ways to do this.

One possible definition is:

```
\newcommand{\rm}{\rmfamily}
...
\newcommand{\sc}{\scshape}
```

This would make the font commands orthogonal; for example `{\bf\it text}` would produce bold italic, thus: ***text***. It will also make them produce an error if used in math mode.

Another possible definition is:

```
\DeclareOldFontCommand{\rm}{\rmfamily}{\mathrm}
...
\DeclareOldFontCommand{\sc}{\scshape}{\mathsc}
```

This will make `\rm` act like `\rmfamily` in text mode (see above) and it will make `\rm` select the `\mathrm` math alphabet in math mode.

Thus `${\rm math} = X + 1$` will produce 'math $= X + 1$'.

If you do not want font selection to be orthogonal then you can follow the standard classes and define:

```
\DeclareOldFontCommand{\rm}{\normalfont\rmfamily}{\mathrm}
...
\DeclareOldFontCommand{\sc}{\normalfont\scshape}{\mathsc}
```

This means, for example, that `{\bf\it text}` will produce medium weight (rather than bold) italic, thus: *text*.

`\normalsize`
`\@normalsize`

The command `\@normalsize` is retained for compatibility with LaTeX 2.09 packages which may have used its value; but redefining it in a class file will have no effect since it is always reset to have the same meaning as `\normalsize`.

This means that classes *must* now redefine `\normalsize` rather than redefining `\@normalsize`; for example (a rather incomplete one):

```
\renewcommand{\normalsize}{\fontsize{10}{12}\selectfont}
```

Note that `\normalsize` is defined by the LaTeX kernel to be an error message.

```
\tiny \footnotesize \small \large
\Large \LARGE \huge \Huge
```

None of these other 'standard' size-changing commands are defined in the kernel: each needs to be defined in a class file if you need it. They are all defined by the standard classes.

This means you should use `\renewcommand` for `\normalsize` and `\newcommand` for the other size-changing commands.

6.4 Obsolete commands

Some packages will not work with LaTeX 2_ε, normally because they relied on an internal LaTeX command which was never supported and has now changed, or been removed.

In many cases there will now be a robust, high-level means of achieving what previously required low-level commands. Please consult Section 4 to see if you can now use the LaTeX 2_ε class and package writers commands.

Also, of course, if your package or class redefined any of the kernel commands (i.e. those defined in the files `latex.tex`, `slitex.tex`, `lfonts.tex`, `sfonts.tex`) then you will need to check it very carefully against the new kernel in case the implementation has changed or the command no longer exists in the LaTeX2e kernel.

Too many of the internal commands of LaTeX 2.09 have been re-implemented or removed to be able to list them all here. You must check any that you have used or changed.

We shall, however, list some of the more important commands which are no longer supported.

```
\tenrm \elvrm \twlrm ...
\tenbf \elvbf \twlbf ...
\tensf \elvsf \twlsf ...
    ⋮
```

The (approximately) seventy internal commands of this form are no longer defined. If your class or package uses them then *please* replace them with new font commands described in *LaTeX 2_ε Font Selection*.

For example, the command `\twlsf` should be replaced by:

```
\fontsize{12}{14}\normalfont\sffamily\selectfont
```

Another possibility is to use the `rawfonts` package, described in *LaTeX 2ε for Authors*.

Also, remember that many of the fonts preloaded by LaTeX 2.09 are no longer preloaded.

`\vpt` `\vipt` `\viipt` ...

These were the internal size-selecting commands in LaTeX 2.09. (They can still be used in LaTeX 2.09 compatibility mode.) Please use the command `\fontsize` instead: see *LaTeX 2ε Font Selection* for details.

For example, `\vpt` should be replaced by:

```
\fontsize{5}{6}\normalfont\selectfont
```

`\prm`, `\pbf`, `\ppounds`, `\pLaTeX` ...

LaTeX 2.09 used several commands beginning with `\p` in order to provide 'protected' commands. For example, `\LaTeX` was defined to be `\protect\pLaTeX`, and `\pLaTeX` was defined to produce the LaTeX logo. This made `\LaTeX` robust, even though `\pLaTeX` was not.

These commands have now been reimplemented using `\DeclareRobustCommand` (described in Section 5.7). If your package redefined one of the `\p`-commands then you must remove the redefinition and use `\DeclareRobustCommand` to redefine the non-`\p` command.

`\footheight`
`\@maxsep`
`\@dblmaxsep`

These parameters are not used by LaTeX 2ε so they have been removed, except in LaTeX 2.09 compatibility mode. Classes should no longer set them.

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[2] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1986. Revised to cover TeX3, 1991.

[3] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

LaTeX 2ε Summary sheet: updating old styles

Section references below are to *LaTeX 2ε for Class and Package Writers.*

1. Should it become a class or a package? See Section 2.3 for how to answer this question.

2. If it uses another style file, then you will need to obtain an updated version of this other file. Look at Section 2.7.1 for information on how to load other class and package files.

3. Try it: see Section 6.1.

4. It worked? Excellent, but there are probably still some things you should change in order to make your file into a well-structured LaTeX 2ε file that is both robust and portable. So you should now read Section 2, especially 2.7. You will also find some useful examples in Section 3.

   If your file sets up new fonts, font-changing commands or symbols, you should also read *LaTeX 2ε Font Selection.*

5. It did not work? There are three possibilities here:

   - error messages are produced whilst reading your file;
   - error messages are produced whilst processing test documents;
   - there are no errors but the output is not as it should be.

   Don't forget to check carefully for this last possibility.

   If you have got to this stage then you will need to read Section 6 to find the solutions that will make your file work.