Bijlage 3

TEX in 2003: Part II

Proposal for a `\special` standard

NTG TEX future working group
P.O. Box 394,
1740 AJ Schagen,
The Netherlands
ntg-toekomsttex@ntg.nl
http://www.ntg.nl

abstract

The text of this article is a proposal for an "endorsed" `\special` specification, to be voted on by the assembly of the TUG98 meeting. Portions of this text are reworks of an original article by Nelson Beebe, and indeed large portions of the proposal itself are also based on original work done by Beebe.

Introduction

Most existing drivers have chosen an arbitrary syntax for the `\special` strings they support. This is undesirable, for at least these reasons:

- The chosen syntax is usually unique to a particular driver, and therefore seriously compromises document portability.
- The syntax is usually not extensible in an easy way.
- The syntax cannot always be unambiguously parsed.
- The output device, or driver, to which the `\special` applies is not determinable.
- The capabilities are weak, and fail to address many of the potential uses of the `\special` command.

The `\special` syntax that we have developed, which is really an extension and modification on the work done by Nelson Beebe, resolves these objections. It has the following features:

- The `\special` string is defined to contain a program written in a small language that consists of an identification string and a command, followed by sequences of assignment statements, possibly with embedded comments.
- The `\special` language is *rigorously defined* by a programming language grammar (available on request).
- The language is *extensible*. An assignment statement consists of a keyword/value(s) pair. Several keywords are already defined, and new ones can be added without invalidating existing uses of the language.
- Keywords are typed, and constant values assigned to them must be of the correct type. The supported types are names, strings, numbers, and dimensions.
- Value string concatenation is supported in the style of ANSI C, avoiding the often severe line length limitations of text editors, operating systems, and file systems.
- Provision is made for encoding all 8-bit characters in the host character set, so that, e.g. binary printer control sequences can be incorporated as *printable*, and *portable*, text in TEX documents.
- A particular keyword, `language`, is provided to permit the user to specify the output device language, or the DVI driver, to which the `\special` command is directed.

By suitable abstractions, it is possible to create a recursive-descent parser for the language in which commands, keywords and value storage locations are provided in a table passed to the parser. The parser code is therefore completely portable, and independent of the commands and keywords in the language it parses.

We will write a table-driven parser that will accept all the commands and keywords we have defined, and this parser, written in the C language, will be included in the DVIview program that will serve as a reference implementation. The parser itself will be available in the public domain soon, and patches will be made to at least `dvips` and `xdvi` to support this proposed standard.

A proposed syntax for the `\special` command

What does the language look like? Some examples will give the general flavor before we describe the details of the grammar. Here are some fragments of hypothetical TEX input which show some of the `\special` commands:

```
% Display a picture with the upper-left
% corner at the current point
\special{**include pict.eps}

% Display a picture at its original
% absolute page position
\special{**overlay "pict.001",
         filetype metapost}

% Display a figure at half size
\special{**include "pict.eps",
           scale 0.5 0.5}

% Switch to a different colour
\special{**colour .09 .06 .6,
           model rgb}
```

Naturally, the details of a `\special` command invocation should be hidden away in suitable macros that are easy to use.

The language grammar

In the original article, Beebe gave a formal grammar for his language. In the interest of keeping this article as short and readable as possible to a non-programmer, that grammar has been deleted and we have not inserted our own. If you are interested in it, it is available upon request. We will suffice here with giving a textual explanation.

We will start by defining the various primitive types that are supported:

Spaces

Whitespace is ignored except as delimiting characters, so the specification can be formatted for readability, or for compactness. Token may not contain embedded blanks (except strings of course).

Comments

Comments are from percent to end-of-line, like in TEX. Comments cannot occur inside of strings or keywords, so this is not a comment:

```
\special{**message "Here % is some text"}
```

and this is in fact illegal:

```
\special{**mes% neat eh?
    sage "Here % is some text"}
```

Names

The grammar states that an `extended letter` is a digit, letter, hyphen, dot, or underscore. These are the characters that are allowed in commands, keywords, alternative values and unquoted strings. Letter case is not significant in these cases.

The characters permitted are chosen such that for instance simple filenames can be used without surrounding quotes (see below for more info on strings and alternative values).

An “alternative value” is actually a string with some predefined values.

Numbers

Numeric constants are parsed by the ANSI C library routine, `strtod()`, which expects to see numbers in the form:

```
[whitespace][sign][digits][. digits]
  [{e|E}[sign]digits]
```

Dimensions

Dimensions can be given in any absolute unit known to TEX (`bp cc cm dd in mm pc pt sp`). Note that the font-specific `em` and `ex` are not allowed. Since tokens may not contain embedded blanks, 210mm is legal input, but 210mm is not.

Any keyword that accepts dimensions as arguments will also accept numbers. In the absence of a dimensional unit, a default value will be used. This default can be defined with a separate `\special` (see below under `defaultunits` for important usage information), or, in the absence of that `\special`, the driver will presume scaled points (`sp`).

Strings

The grammar supports unquoted strings and two kinds of quoted strings.

An unquoted string has to be one word only (since there are no spaces allowed), and can only use the characters that are legal `extended letter`s as defined above.

The *normal* kind of quoted string is delimited by double quotes, and inside it are recognized all the escape sequences supported by the C language. The *raw* kind is delimited by single quotes; only escape-single-quote pairs are recognized inside it. This is more convenient when it is necessary to have strings with several backslashes, since it then avoids having to double all of them. Once normal and raw strings are parsed, they are stored identically.

Backslashes in literal strings and filenames pose a small problem for the user, because TEX will ordinarily try to interpret control sequences triggered by backslashes in the argument of the `\special` command. Although it would have been possible to choose another escape character than backslash for such strings, this would likely prove confusing to those users who are used to C and UNIX, where the backslash escape character is firmly entrenched.

Fortunately, the solution is not difficult, because TEX does not have backslash hardcoded as a control sequence prefix; you can change it by altering TEX's catcodes.

In the descriptions of the \specials below, the character n and m used to indicate a value from a fixed set of alternatives, s is used to indicate all sorts of strings, x, y and z (possibly with numeric tags) are used for dimensions, and a through j are used for numbers.

Now let us move to the portions of a \special that actually define things. The structure of a \special command is as follows:

ID bytes

The first 2 characters in every \special are to be the two tokens **. The rationale behind this is that a convention like this makes it easier to adjust programs that have to remain backward-compatible with their old private syntax. As far as we know, this particular sequence of tokens is never used in current \specials.

Command

The next word is the principal command for this \special. Depending on the command itself, it may have arguments or it may be a single command.

Assignments

Optionally, the command can be followed by a series of keywords that supply extra information. Keywords follow the same syntax as commands, so there can be zero or more arguments to a keyword.

In a series of assignment statements, the order of the keywords is not significant, except that if duplicate keywords are specified, the value of the last one is used.

Every keyword-value group needs to be separated from the previous one by a separator, which may be either a semicolon or a comma. This is correct:

```
\special{**include "pict.eps";
          scale 0.5 0.5}
```

And this is not:

```
\special{**include "pict.eps"
          scale 0.5 0.5}
```

Separating items

Finally, the assignment statement may use either the equals or colon operator, or the operator may be omitted altogether. This supports the common forms:

```
\special{**include=pict.eps}
\special{**include:pict.eps}
\special{**include pict.eps}
```

Because the values have very limited syntactical possibilities, there is no ambiguity created by this.

The \special language

The preceding section defined the grammar for the \special language. We now need to define what commands and keywords will be recognized. As emphasized above, the language is *extensible*, and the parser that we will implement for it makes it easy to add new commands and keywords *without touching a single line of the parser code itself*.

However, we presume that there will be a maintainer or maintenance group assigned to take care of this specification, and this person has the right to refuse to accept extensions that do not fit in.

Generic keywords

The full set of commands and keywords that are recognized is given below, but we will start off with some general keywords. These keywords can be used within any \special, and also be used as a command. They will not be mentioned separately in the descriptions of the other \specials:

Keyword	Value	Action
message	s	Supply an operator message to be sent to the terminal and log file.
id	n	Supplies a name that uniquely identifies this \special.
use	n	Supplies a name that identifies a previously defined \special.

The message string provides a means for operator communication; for example,

```
message "Thesis bond paper for this job"
```

The message is sent verbatim to the terminal and the log file.

The id allows identification of the \special it occurs in. The command and the keywords and values associated with this \special, are saved and available for later reuse through use. The current location in the file is also saved, for later retrieval by one of the cross-link \specials.

The usage of use is as follows: first, all of the data from the \special it refers minus the id value are inserted in the current \special, and other any values that occur in the current \special are used to override the inherited options. An example is probably the best way to show this.

After

```
\special{**include "pic1.eps";
          scale 0.5 0.5;
          id mypic}
```

The following command re-does precisely the same in a later portion of the document:

```
\special{**use mypic}
```

and

```
\special{**use mypic;
          scale 1 1;
          id mypic2}
```

inserts the same figure, but at a different scale. It also assigns a new `id` to this current `\special`. The following is also allowed

```
\special{**include "pic2.eps";
          use mypic;
          id mypic2}
```

but it is *not* legal to switch to an entirely different command, like `overlay`.

Drivers are allowed to set an upper limit to the number of *distinct* `id`s that can be used in a document, but this limit should not be lower than 256. There is never a point to limit the total amount of `id`s, since later definitions will just overwrite the previous one with the same name.

There is at the moment exactly one command that affects the `\special` parser itself:

Keyword	Value	Action
`defaultunits`	n	Sets the default units to one of the defined dimension types instead of `sp`.

Commands for graphics inclusion

There are three possible ways of including a graphic figure file from disc:

Keyword	Value	Action
`include`	s	Insert file contents with relative page positioning.
`overlay`	s	Insert file contents with absolute page positioning.
`underlay`	s	Insert file contents with absolute page positioning.

The filename string can be used for normal local files, but it can also be used for URLs, following the normal rules for URL specification. If no explicit protocol (like `http` or `ftp`) is given, the name is assumed to be a local file. Even non-networked drivers are required to correctly handle one protocol: `file://`.

In all these three cases, drivers can opt to give a default search path for figure files with relative path names, but this is not required nor encouraged. The driver is not required to include any file type except `dvi`.

`overlay` and `underlay` are supposed to start from the lower-left corner of the physical page, with coordinates as in PostScript: up and right are positive values for x and y. In cases where there is no obvious lower-left corner (as may be the case for on-line backends), the lower-left corner is defined to be at the end of the output medium.

`include` places the top-left corner of the image at TEX's current point. Here coordinates are as in DVI: down and right are positive values for x and y.

The difference between `overlay` and `underlay` should be clear: overlays can actually obstruct other images and text on the page (depending on where precisely on the page the `\special` was given), underlays can never do this, but a second underlay might be on top of the previous one.

If the file cannot be opened, or for relative positioning, the bounding box cannot be determined, a warning message is issued and the `\special` command is ignored.

There is also a `\special` command available for the inclusion of literal drawing commands:

Keyword	Value	Action
`graphics`	s	Execute the graphics primitives in string (defined below).

The `graphics` keyword value is used to insert simple generic graphics commands in one of the existing (mini-)languages for graphics. These are properly handled by using the `graphics` and `type` keywords together.

```
\special{**graphics = "...",
         type = tpic }
```

The driver will issue an error if there is a `graphics` command without a `type` specified as well, and the corresponding `\special` will be ignored. The driver is not required to execute `graphics` except if the `type` is `dvi`.

All four graphics `\special`s accept the following options:

Keyword	Value	Action
`boundingbox`	x1 y1 x2 y2	Defines the four dimensions

		of the lower-left and upper-right corners of the box which bounds the figure.
clipbox	x1 y1 x2 y2	If present, clipping to the specified four dimensions should occur.
position	n m	Specify the reference point on an inserted figure which is to be mapped to the current page position.
size	x y z	three values that are absolute dimensions for the size of the figure.
type	s	gives a way to specify the type for files with non-standard extensions.

boundingbox also applies to graphics, since it can be used to decide whether and where clipping should occur. Note that this is essentially the same value as the PostScript BoundingBox for (E)PS figures. For clipping purposes, this statement overrules the in-file version of such a BoundingBox. In the absence of a boundingbox keyword, (E)PS and similar file formats where it is legal to draw outside the box should *always* be clipped to the in-file values.

The position keyword specifies two values. The first should be one of top, middle, or bottom, and the second should be one of left, center, or right. These words may be abbreviated to a single letter if desired. Together, they select on the bounding box one of nine points (four corners, four edge centers, and the box center) which is to be placed at the TEX current point. If this keyword is not given, the default is

```
position = top left
```

The point selected by this keyword (or by default) will be the *reference point* for the insertion of the graphic file.

In the values of size, negative dimensions means that size in that direction should be ignored.

The string argument to type is used to give information about the type of file or graphics. This value should be either the 'normal' three-letter extension for this type of file or the name of a graphics description language. The following language names are predefined: dv, dvi (ordinary binary dvi commands), epic, encapsulated postscript (also eps), eepic, emtex, fig, metapost (also mp), pcl, pdf, postscript (also ps), tektronics, tpic, xpic.

Generic graphics keywords

There are three keywords that define transformations. Actually these belong to the graphics language, but they can also appear inside figure \specials, which is why they are explained here.

Keyword	Value	Action
translate	x y	Defines two dimensions that shift the figure's reference point from the default value.
scale	a b	one or two numbers that are relative to the 'normal' size of the figure.
rotate	a	rotation angle in degrees. Counterclockwise is positive.

These three keywords can be used as stand-alone commands, in which case they apply until explicitly stopped by means of one of the commands we will define below, or they can be included inside one of the four \specials for figure inclusion, in which case they only apply to the subject of that \special.

The keyword size is processed before taking any transformation commands within the same \special into account.

Rotations etc. that were in force at the time the figure \special was encountered, *are* taken into account before the calculations for inclusions are done. Here is a small example that demonstrates possible usage:

```
\special{**gsave}
\special{**scale=2 2}
     Some large text here
\special{**rotate=45}
     Large and rotated text
     \special{**include test.eps,
              rotate = 45}
     This figure is rotated 90 deg CCW
     and twice as large.
\special{**grestore}
     Back to normal
```

Command for colour specifications

There is only one command defined for colour specification (well, actually two, since the American spellling "color" is also accepted), and one optional keyword:

Keyword	Value	Action
colo(u)r	?	The value should be the numbers or tokens that specify the color in the defined colour model
model	s	The value should be a recognizable color model name

Every driver is required to recognize the following six named values for the option string of `model`. These are the ones that define the four most commonly used colour models: `rgb`, `cmyk`, `gray`, (also known as `grey`) and `mono` (`bitmap`).

For all these predefined colour models, a colour is defined as one or more real numbers between 0 and 1. In the absence of a `model` keyword, drivers should take the following guess as default action: if there is one number in `colour`'s value, the colour model is `grey`. If there are three numbers, the model is `rgb`, and if four, the model is `cmyk`. All other non-qualified values signify a syntax error.

Commands for the in-line graphics language

First there are the commands that change the state of the graphics system's default values:

Keyword	Value	Action
setlinejoin	n	Select method of joining lines.
setlinecap	n	Selects the line ending method. One of `butt`, `round`, `square`
setdash	offset values	Select the dashing pattern for drawing lines.
setlinewidth	x	Selects the line-width.
setmiterlimit	a	Sets the miter limit for drawing.
setoverprint	n	Value is `yes` or `no`
setvisible	n	Value is `yes` or `no`

Note that the commands `scale`, `translate`, `rotate` and `colour` also belong to this category.

`setvisible` and `setoverprint` are supposed to compensate for overlays and underlays as well as for the background colour of the page (defined below in the section on paper settings).

Then there are commands that draw stuff:

Keyword	Value	Action
moveto	x y	Moves the cursor position to (x,y).
lineto	x y	Draws a line to (x,y).
curveto	x1 y1 x2 y2 x3 y3	Draws a Bézier curve where (x1,y1) and (x2,y2) are the control points and (x3,y3) is the end-point.

All three commands draw relative to the current point, and in fact, they even move the driver's idea of 'current point' just like the regular DVI commands do. If this side-effect is undesirable, the commands should be part of an explicit drawing, which is defined and drawn with one of the following commands:

Keyword	Value	Action
startgraphic		Indicates the beginning of a graphic.
stopgraphic		Analogously ends a graphic.

Inside one of those explicit figures, the drawing commands do not actually draw anything. Instead, one of the following commands should be used:

Keyword	Value	Action
newpath		Discards any present paths and start a new one.
closepath		Closes the current path.
stroke		Draws all the lines with the current selected pen.
clip		Selects the current path as the clipping path.
fill		Fills the current path with the current selected color.

Of course you are allowed to use the other commands too, and there might be intermixed text. Page breaks are not allowed though, since the entire graphics state will be restored to it's default state at the beginning of the page. Usage of these commands is analogous to PostScript.

Alternatively, the graphics state can be saved and restored explicitly, again as in PostScript:

Keyword	Value	Action
gsave		Saves the graphics state. Position, current color, current

	path, current clipping path, current transformation matrix, and the current pen-type is saved.
`grestore`	Restores the graphics state.

Commands for hyper-referencing

There are not that many keywords explicitly involved with hyperlinks, since they can use the keyword `id` to mark either pages or locations. in the document. The link specification decides whether the specific `id` indicates a location marker or a page marker.

Linking re-uses the option keywords `position`, `size`, `filename` and `type` that are defined elsewhere in this paper.

Keyword	Value	Action
`linktopage`	n	The name has to be defined though an `id` elsewhere
`linktoloc`	n	The name has to be defined though an `id` elsewhere
`linkend`		Ends an HTML style link
`position`	n m	Specify the reference point of the link area.
`size`	x y z	three dimensions that are width, height and depth of the link area.
`filename`	s	This is the url in case an external file is linked to
`type`	s	gives a way to specify the type for files with non-standard extensions. The value should be the 'normal' three-letter extension for this type (like `pdf` or `dvi`).

The value of `size`, if available, gives the borders of the 'clickable area'. An example:

```
\special{**id=1}This is a
\special{**linktoloc=1,
   size=16pt 6pt 1pt}link.
```

If `size` is not explicitly given, `linkto...` functions analogous to the HTML style syntax, and `linkend` is used to stop the area. Here is an example of the this approach:

```
\special{**id=1}This is a
    \special{**linktoloc 1}link%
    \special{**linkend}.
```

It is a syntax error to end a link with `linkend` if that link was started with an explicit `size`, and the entire link specification will be ignored by the driver.

It is *not* an error if there is a line or even line break in the case that is supposed to end with `linkend`. These cases have to be handled correctly by the driver (the clickable area will probably have to be split in separate parts).

Commands for meta-information

A number of keywords is available to pass information to the processing application. This information can be used to fill <*meta*> tags or for debugging purposes.

Keyword	Value	Action
`info`	n	Value can be either `meta`, `debug`, or `comment`
`title`	s	Name of the current document
`subject`	s	Subject of the document
`author`	s	The (probably human) author
`creator`	s	The generating program
`version`	s	Version information
`keywords`	s	Keywords for this document
`abstract`	s	Short abstract for this document
`filename`	s	Original filename
`lineno`	a	Records original line number in source
`charno`	a	Records character location in line
`byteno`	a	Records location in file
`date`	a b c	Date in a fixed format (`dd mm yyyy`)
`time`	a b	Generation time in fixed (`hh mm`) format, assumed to be GMT

The meanings should be clear from the names. These commands can all be used inside of any other `\special` in this same group, and they can be used in the optional part of the three figure file inclusion `\specials` and as part of the `linktoloc` and `linktopage` commands if they refer to an external file, where they can be used to request a specific version of a file. (The driver does not have to honour these latter cases in order to comply, but it is required to give the usual warning about failing to process the `\special` entirely).

Handling paper.

Device initialization can be a complicated business, so it will usually require the `language` keyword as well (see below), but some of the more common keywords can be

defined without problems. Paper is fairly simple. There are two commands available, `paper` and `screen`.

Keyword	Value	Action
`paper`	s	paper form name
`screen`	s	screen form name
`height`	x	paper or screen height
`width`	y	paper or screen width
`colo(u)r`	?	The value should be the numbers or tokens that specify the color in the defined colour model

The `paper` and `screen` keywords defines a name that is used to tag the collected parameters. If the form name already exists, assignments will replace previous values. Otherwise, a new form is created. `screen` is intended for on-line formats, and is a synonym for `paper` that feels more natural in this case.

The use of `colour` here defines the background colour of the paper or screen. Printer drivers (or any other driver where execution of this command might lead to very expensive output) are supposed to ask confirmation from the user before executing this `\special`.

Other processing options
are

Keyword	Value	Action
`imaging_type`	n	The type of imaging that is applied
`resolution`	x	Gives the required resolution for device where there are more possible resolutions
`tray`	n	Tray number for devices with more then one input tray
`duplex`	n	Either on or off

The `imaging_type` can be one of the words `normal`, `negative`, `mirror` or `mirrornegative`.

The commands are used for for instance typesetter output, and they always apply to at least one full page (the page the `\special` appeared one)

Other device options
Certain drivers might require certain extra commands that only they understand. There is one command reserved to handle these things.

Keyword	Value	Action
`language`	n	Name the output-device language for which this `\special` is intended.
`literal`	s	Insert literal output device code.
`options`	s	Insert driver option.

The `language` keyword determines whether the DVI driver will process this `\special`, or ignore it.

Drivers are not required to understand *any* kind of `language` special, and are free to ignore that `\special` right after it has seen the `language` command. However, any driver that is willing to support this `\special`, even in a very minor way, *must* recognize a generic language choice relevant to its output device, such as `PostScript` or `Epson`. Also, each driver that tries to handle this `\special` *must* recognize its own name as a language value.

`literal` is allowed to occur *only* in combination with `language`, and is used to insert literal portions of the command language used by the `language` in question.

The `options` keyword can be used to supply device-dependent information to the driver; this is only allowed if the `language` is the name of a driver.

Correct driver behaviour

Drivers are supposed to correctly interpret and execute all of the `\special`s defined in this document, except were we specifically indicated that this is not needed.

If the program that processes the DVI file *does not* know how to handle a specified `\special` (other than those defined in this document), it is allowed to issue *at most one* warning to the user per unrecognized `\special` type.

Since there is a reasonable chance that this DVI file at hand should have been processed by another program alltogether, one warning seems prudent, but that should be enough. This rule prevents the appearance of miriads of "unknown special" warnings in documents that have parallel `\special`s for various drivers.

If the program that processes the DVI file *does* know how to handle a certain `\special`, it is allowed to issue messages, warnings or errors as it sees fit. It is always *required* to give warnings in the case of a `\special` that can only be partly obeyed. It is also *required* to give user errors for all `\special`s that have syntax errors (assuming the driver is aware of the right syntax, which may not always be the case, but is definately the case for the `\special`s defined here)