Driver Support for Color in TeX: Proposal and Implementation

Tomas Gerhard Rokicki
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
Internet: `rokicki@cs.stanford.edu`

Abstract

The advent of inexpensive medium-resolution color printing devices is creating an increasing demand for flexible and powerful color support in TeX. In this paper we discuss a new implementation of color support and propose an initial standard for color and color-like specials. We first discuss the difficulties that are presented to the driver writer in supporting color, and other features, by presenting a number of hard examples. Second, we present an implementation of a driver that provides a solution to many of the problems discussed. Best of all, this solution includes modular C code that is easily integrated into other drivers, automatically translating the higher-level special commands into existing low-level special commands.

Introduction

This paper has two parts: a collection of difficulties, and a proposed partial solution. The collection of difficulties is by far the easier part to write and to read; it is always easier to criticize than to originate. Nonetheless, it includes some subtle conclusions. The proposed solution does not come near to solving all of the problems raised in the first section, but it attempts to solve at least one, as one step towards a more general solution for the remaining ones.

Our perspective is that of a `dvi` driver writer. We care not for the user; let the macro programmers provide a convenient interface. Rather, we attempt to provide the primitive functionality from which specific effects can be accomplished.

For driver writers, on the other hand, we have untold sympathy. We will even do much of the work for them, by providing a set of C routines that implement the new functionality.

In order to understand why each problem is difficult, and what conclusions we can draw from each problem, we need to understand the limitations of TeX and of the various device drivers. While there is only one TeX, there are many different types of device drivers, each with its own requirements and capabilities. We can divide the drivers into four categories according to their style of operation.

The first kind is a driver that scans the entire `dvi` file (or at least up to the last required page) before generating any output. This prescan phase usually determines what fonts and what characters from each font need to be downloaded. This type of driver is typically necessary for laser printers.

The second type of driver does not perform this prescan phase, usually because the output device does not support downloaded fonts; this is typically the case for dot-matrix printers or FAX machines. This type of driver must render the entire page before shipping even the first row of pixels; it too buffers information, but at the page level instead of the document level.

Both of these types of drivers typically process the pages in the order they are given in the `dvi` file. A previewer, our third type of driver, does no such thing; instead, the pages are processed in some random order, and quick access to each page is desired.

The fourth and last type of driver we recognize is the driver that generates a `dvi` file as output. These include programs that do pagination tricks, like `dvidvi` and `dviselect`, and programs that expand virtual fonts, like `dvicopy`, and even the `dvicolorsep` program that does color separation.

Because TeX does not support color directly, we conclude that any such support must come through specials. Thus, the task of the device driver writer is two-fold: to recognize and parse the specials that direct his rendering, and to perform the rendering appropriately. This paper is primarily concerned with the first task. Color rendering and imaging is incredibly complex, so other than a few minor points, we shall not yet concern ourselves with these issues. Instead, we adopt the current solution, as described in Dr. Hafner's paper in these proceedings.

Part One: The Problems

Now we are ready to present some sample difficulties and draw some conclusions from each.

Colored text and rules. Our first example is the most basic; we want to specify that some text or rules in our document be red. Because TeX does not allow us to attach color information directly to text or rules, this must be implemented as a change of state for our abstract rendering engine. Since we are using specials to implement colors, this change of state must occur at the point in the `dvi` file that the special is emitted. Therefore, *specials that indicate state changes must be used to implement colors.*

Even at this early stage, problems arise. It is not always obvious to the user where a special will be emitted. In general, it occurs in the same place in the linear stream of text that the user types, but occasionally this is not the case. Consider, for example, LaTeX 2.09's `list` environment. Placing a special immediately after an `\item` command causes the special to occur in the `dvi` file before the bullet, coloring the bullet; this is not the intuitive result. (Technically, this happens because the special does not cause a switch to horizontal mode and is thus simply attached to the current vertical list; the bullet is inserted at the head of each paragraph, which starts with the switch to horizontal mode.) On the other hand, this can be considered simply a side-effect of the way the list environment is implemented; adding a `\leavevmode` command before the special command works around this difficulty. LaTeX2$_\varepsilon$ solves this particular problem using color nesting, but similar problems can arise in other situations and with other macro packages.

If the state change occurs at the point at which the special occurs, then how shall we define the range of the color command? One alternative is to define the range to be that sequence of `dvi` commands enclosed between two specials. A second is to define it to be until the end of the enclosing TeX box. A third is to define it to be until the end of the enclosing TeX group. A fourth is to use some combination of these.

Unfortunately, the box solution fails in a number of ways. First, there is no real notion of boxes at the `dvi` level. Indeed, this can make it difficult to color a paragraph red—that paragraph might be split across several pages, and thus several boxes, with no overall enclosing box.

The first solution subsumes the third. Groups are not visible at the `dvi` level, but TeX's aftergroup command can be used to make specific groups visible. Therefore, *the range of color commands must be from special to special.*

Nested colors. The next question is whether to nest colors. In other words, should we be able to color a word red, without having to figure out and restore the color of the enclosing paragraph? Somehow it seems more consistent with TeX to allow nesting of colors, and in many situations, nesting colors solves some important problems. For instance, nesting is used in the previous version of color support in `dvips` and in the current version of LaTeX2$_\varepsilon$ to allow headlines to work correctly. Certainly it is not hard to implement. Thus, *we should allow the nesting of colors.*

Should the driver be responsible for maintaining the color stack, or should the TeX macros? Either is easily implemented, and since the color stack should never nest deeply, the resources consumed by either should be negligible. If we use TeX, we can always make the current color available to the user of the macro package, provided that we standardize on some representation. On the other hand, we might not want to require that the color stack be provided by the macro package—and implementing a color stack is easy enough that we might as well provide one at the `dvi` driver level. Providing one at the driver level does not require the TeX macros to use it. In any case, backwards compatibility with the current color implementation requires a color stack. *The driver should implement color stacking, and some macro packages might also maintain the color stack for their own purposes.*

Should we also include a command to set the current color, independent of state changes? If we are using a set of simple macros that just set the color and ignore the stacking capability of the driver, this might cause the stack to get increasingly deep. And just issuing a pop stack command before each color command fails with the first color. Since it is a pretty easy feature to provide, we might as well. *The driver should implement non-stacked color changing.*

Colored text split across pages. Now imagine the word "example," in red, split across two pages. At the `dvi` level, the "begin red" special will occur near the end of one page, and the "end red" special will occur near the beginning of the next. Thus, `dvi` *drivers must maintain the color stack information across pages.*

In the context of page reversal, page selection, and random page access, this requires that the dvi driver store the contents of the color stack for each page it might need to revisit, and set up the output device state appropriately. This is not hard to implement once the requirement is understood.

Page break in colored region with black headline. There is a danger that a color region split across pages might also cause some headlines or footers to become contaminated with color. There is nothing in the dvi file indicating that some text is a headline or footline, so a straightforward nested color implementation will have this problem. The only real solution to this is to have the output routine put that information in the dvi file. Similar problems arise with footnotes, figures, and marginal notes. *The TeX output routine must indicate the origin of text in order for the color to be maintained correctly.*

Alternatively, the output routine can simply reset the color to black in regions such as headlines, footlines, marginal note, and floats; this is the solution currently adopted in LaTeX 2_ε.

Split footnote with colored regions. It might be desired to color headlines or marginal notes. Indeed, footnotes might have colored regions that are split across pages. A single page break might split both a pagebody colored region and a footnote colored region. Therefore, *the driver should actually maintain separate and independent color contexts, each with its own color stack, and the output routine should issue the necessary commands to switch among them.*

In the case of marginal notes, it may not be clear what the enclosing color context is. A marginal note might occur inside of a float or inside of a normal pagebody paragraph. Therefore, *the driver should maintain a stack of color contexts.*

Such contexts make it easy to do things like color all headers red; simply invoke the header context, push or set the color red, and then return to the previous context.

It is not clear how many different sources of text there might be, so the color stacks should be dynamically allocated by name inside the driver.

Footnotes within a colored region. Floats pose an interesting problem. If an entire section of a document is colored, should the included footnotes be colored as well? What happens if the floats move into the next section? As a logical consequence of the color context idea, they should (by default) not be colored, since they are from a different stream of text. On the other hand, to provide just this functionality if it is desired, it is easy to provide a global context that is always used for attributes not set in the current context. This global context will provide functionality backwards compatible with the current FoilTeX color model, and it will allow setting the color of entire regions of a document. On the other hand, it will not allow floats or footnotes that have portions on pages after the end of the color region to have the appropriate color; the color contexts must be used to obtain that effect. *A special "global" color context should be used as a default for parameters not set in the current context.*

To summarize, all stack push and pop commands affect the context on the top of the context stack; this is the current context. Colors (and other items) are always searched for first in the current context and then, if not found, in the global context.

An alternative, and perhaps preferable, implementation is to search in the current context, and then in the next context on the context stack, etc. This may be more natural, but it undoes the "defaulting" that we currently get if we set the pagebody to red and draw a marginal note. We believe this defaulting is more important, so we have implemented evaluation to only search the current and the global context, rather than all of the ones on the context stack.

Everything we have described so far is easy to implement. At the beginning of each page, we have a particular stack of contexts, which we save away in case we ever need to render that page again. In order to generate that data structure for a particular page, we must scan the dvi file from the front to that page. In other words, in the presence of color, it is no longer possible to read the dvi postamble and skip backwards on the previous page pointers in order to quickly find a page. On the other hand, the processing required to skip pages is negligible. *In order to properly render any page, all previous pages must be scanned.*

Because it is trivial to write out specials to set the stacks to any desired state, page reversal is also implementable. Indeed, it is easy to eliminate the stacks altogether using a dvi to dvi translator, thus allowing the use of simpler drivers, or translating the specials to a form recognized by a particular driver. The only trick is to use a syntax that allows the dvi to dvi program to easily distinguish those specials it must manipulate from those that it must leave alone.

Changebars. The color mechanism we have described will also help with tasks other than color.

For instance, changebars are also complicated by the asynchronous nature of TeX's output routine. Defining changebar on and changebar off to be color-type commands gives us the full nesting and state saving capabilities we used for color. Indeed, we can use the context switching commands to give us a vertical reference position, and define some changebar parameter to give a horizontal offset from that position, allowing dual-column changebars. This solves the problem of having changebars span inappropriate figures and not span appropriate ones.

The current implementation does not yet support changebars, but the author feels that the changes should be straightforward. Indeed, as with color, it is possible for a `dvi` to `dvi` program to convert a `dvi` file that specifies changebars into one that uses explicit rules. *Color and color context specials are appropriate for tasks other than color.*

Colored backgrounds. Another use of color, especially for slides, is in setting the current background color. Instead of modifying characters and rules between specials, this affects the entire page background before anything is drawn. There is no reason not to allow this to nest just like other color commands do, even though the primitives are at a different level. Thus, *we must be able to specify the background color.*

Colored background with headline on first page. Because of the way specials are sent out, headline text is emitted before any specials attached to the page contents. Thus, if the first page has a headline, that headline will occur in the `dvi` file before any page content such as specials. Therefore, the page global attribute values in effect at the beginning of a page, or before the first character or rule in the `dvi` file, might not be what is intended.

To solve this problem, we define that the page globals in effect at the *end* of the page are what define the values for the page background, orientation, or other page globals. This has two effects. The simple one is that page globals must be syntactically distinguishable from non-page-global color information. Indeed, this last requirement also allows us to distinguish a page-global rotation from a local rotation. *Page globals must be syntactically different from local attributes.*

A more important effect is that either pages must be fully prescanned before rendering can begin, or the driver must be prepared to restart the rendering of a page if a page global is encountered with different values from those currently in effect. Currently, many drivers prescan anyway. For those that do not, they cannot send out the first row of pixels until the entire page has been scanned anyway (a character at the top of the page might be the last character rendered in the `dvi` file), so rerendering when necessary is not terribly inconvenient. Thus, *to support page globals, pages must be prescanned or possibly rerendered.*

Paper size specification. One important page global is the specification of the paper size. Indeed, the lack of a standard for this information makes the driver's job much more difficult; knowing the job is intended for A4 paper can allow the driver to either request the appropriate paper, or shift or scale the page to fit. Certainly paper size is a typesetting-level and not a print-level option. Paper size should be specified as a page global on the first page. *The desired paper size should be specified in the* `dvi` *file.*

Imposition of pages with colored backgrounds or varying paper sizes. One function of `dvi` to `dvi` programs is page imposition—where pages are laid out in a specific order and orientation so that the folded signatures contain them in the proper order. When pages are imposed, the semantics of the page global options such as paper size and background color change slightly; this is simply a complexity that must be dealt with by the `dvi` to `dvi` program. It is possible to approximate some of these combinations using the appropriate `dvi` commands; for instance, page background commands can be converted into commands to draw a large background rule in the appropriate color.

Envelope/media selection. Page globals, such as paper size, might change in a particular job. For instance, many modern printers include an envelope tray; it would be convenient to have a media-selection page global that would allow a standard letter style to properly print the envelope, or select a sheet of letterhead for the first page of a long letter. *Drivers should support different paper sizes within a single document.*

Coloring the backgrounds of boxes. Occasionally a user might want to color the background of a particular TeX box. There are several problems with this. The first is that the box information simply is not available at the `dvi` level. The second is that the box dimensions tightly enclose the contents; does the user really intend to have the italic "f" protruding from the colored region? Finally, this is something that is easy to do at the TeX macro level by simply drawing a rule of the appropriate size and color before setting the box. *Many things still should be implemented at the TEX level.*

Colored table backgrounds. One of the more common uses of color is to decorate the backgrounds of tables—each column gets a distinct shade or color. This is quite difficult to implement, although Timothy Van Zandt has had success with his `colortab.sty`. The primary difficulty is obtaining the column dimensions—height and width—before rendering the text of the columns. *Many common requirements still defy easy solution.*

Included graphics and other objects. It should also be possible to include graphics and do other rendering with specials, in the way they were intended. The main requirement is that *these types of specials be syntactically different from the color specials, so that* `dvi` *to* `dvi` *programs know which specials to manipulate and which to leave alone.*

As an aside, it is important that the mechanism for including graphics respect the `dvi` magnification and any rotation and scaling commands, so that imposition and scaling work correctly. In addition, it would be convenient to be able to easily calculate the size of the enclosing rectangle from just the special arguments so that, if nothing else, an outline can be drawn. *The* `dvi` *magnification should be respected in scaling graphics, and some standard for sizing/scaling included objects should be defined.*

And while we are off the topic, there is no excuse for not rendering PostScript graphics and fonts with previewers and non-PostScript drivers. The fine freely-available programs GhostScript and ps2pk do all the hard work of rendering for virtually any platform; a few dozen or hundred lines of interface code is usually all that is necessary for a fail-safe interface. *If you can't fully use PostScript in your TEX environment, it is time to complain.*

White on black. It is not necessary to wait for a color device to support color. Even black and white printers should support the two colors black and white, including being able to render white text on a black background. This is useful in itself and for color separations. *Even black and white devices need "color" support.*

Dithered text. When approximating gray text on the screen or to a low-resolution printer using dither patterns, the resulting image is often impossible to read. This is because the dither pattern sacrifices the high resolution needed to render characters for the ability to approximate gray levels.

In professional printing, spot colors, rather than four-color separations, are used to render small text and other single-color highlights. It is important to be able to specify what colors are intended to be spot colors. *We need a standard for specifying and using spot colors.*

For previewing or rendering on low-resolution printers, it is often useful to disable dithering for small fonts in order to end up with something that is readable.

Fountains. Another comment request is fountains. These are smooth graduations of color over an area. For instance, many slides are rendered with a background that is deep blue at the bottom and lighter blue at the top. A rainbow can also be considered a fountain. Fountains are normally approximated by drawing hundreds or thousands of narrow rules, each of a color midway between its neighbors. Whatever color model is chosen for TEX, it would be extremely nice to be able to render fountains.

There are many more examples, including clipping paths, character fountains, chokes, spreads, and the complexities of color vision, color rendering, and color models, that we will not address here.

Part Two: Some Solutions

This second part proposes a solution and implementation for some of the problems listed above. This implementation is used in both `dvips` and `dvidvi`, and the code is freely available to be used in any manner whatsoever.

First we will discuss a categorization of specials. Next, we will define a syntax, and finally, we will describe some keywords and what they mean.

Before we delve into the technical details, let us dispose of one objection: why not just introduce a little language for specials? In essence, that is essentially what we are doing; some might ask why not give it variables, types, and control flow as well. Of course, TEX is already a language; any processing that can be done at the special level is probably better and more portably done using TEX. Also, we would rather people spend their time learning a more practical language, such as PostScript. Indeed, some may consider what we are proposing as already unnecessarily complex—and they may be right.

With even a very simple language, implementing things such as change bars, colored table backgrounds, and much more would be straightforward.

We are not ready yet to define such a language, but we see it as an extension of what we propose here.

Syntax and parsing. Specials are case-sensitive. Words are defined as sequences of characters delimited by any of tabs, spaces, commas, equal signs, or open or close parentheses. If one of the delimiting characters is an equals sign, then the word on the left of the equals sign is associated with the word on the right.

The first word of the special is the keyword. The remainder of the special are its optional arguments.

If a double quote occurs, everything up until the next double quote is considered a single argument.

If a left quote occurs, the following argument is treated as a string without the left quote. If such an argument is opened as a file name, the argument is treated as a command to be executed, and the output from that command is read as the input from the file.

The types of words are string, number, and dimension. Strings or keywords are sequences of numbers, digits, or any character other than delimiters. Numbers consist of an optional negative sign followed by a sequence of digits, optional decimal point and additional sequence of digits. Dimensions are numbers, followed by an optional `true`, followed by one of `in`, `pt`, `bp`, `dd`, `cm`, or `mm`. They are interpreted exactly as in TEX.

Categories of specials. We divide specials into five categories: context switches, foreground state changes, background state changes, document globals, and objects.

1. Context switches push and pop contexts onto the context stack by name. If the context named does not exist, it is created. The default context at startup is `global`.

   ```
   context <push/pop> <name>
   ```

2. Foreground state changes set, push, or pop a foreground state item, such as a color.

   ```
   attribute <push/pop/set> <name>
      [<value>]*
   ```

3. Background state changes set, push, or pop a background state item, such as a background color or paper type.

   ```
   attribute <push/pop/set> page
      <name> [<value>]*
   ```

4. Document globals set some resource requirement or provide some other information. These specials must always occur somewhere on the first page.

   ```
   attribute <push/pop/set> document
      <name> [<value>]*
   ```

5. Objects are everything else, including snippets of PostScript code and included graphics.

   ```
   psfile=foo.ps llx=72 lly=72
      urx=452 ury=930 rwi=500
   ```

With the above syntax, it is easy to syntactically identify the type of a special without needing to understand the specific instances.

Interpretation. We have introduced the idea of a `dvi` color context that can be saved and restored in a non-nested fashion. We allocate contexts dynamically as they are encountered; a macro package might define one for each of footnotes, pagebody, figures, headers, marginal notes, and global. The output routine will then issue the appropriate 'switch context' commands at each point.

```
context push header
<header stuff>
context pop
context push pagebody
<pagebody>
context push figure
<figure>
context pop
<more pagebody>
context push margin-note
<margin note>
context pop
<more pagebody>
context pop
```

Default values for attributes are more troublesome. Consider a document that, on page ten, sets a specific special attribute woomp to the value there-it-is, and this value remains set for the rest of the document. If this document is reversed, the set would then occur at the beginning of the new document—but something must be done to undo the special at the place where page ten now occurs.

The solution is straightforward. If a context stack does not have an entry for a particular attribute when a set occurs, the set is interpreted as a push; otherwise, the set is interpreted as a pop followed by a push. Thus, for a flat sequence of sets, the first will allocate an entry on the context stack for the attribute, and all others will modify that attribute. If it becomes necessary to reset an attribute to its default value, a pop will suffice.

The following implementation effectively flattens all contexts into a simple sequence of set attributes and pops. Pops are only issued to reset attributes to their defaults; there are no corresponding pushes except the implicit ones introduced by the sets.

Thus, with the provided C code, it is trivial to integrate color contexts into an existing driver.

Implementation. Implementing these specials is straightforward. The key idea is that we need to maintain the stack states for each page and restore them appropriately. In addition, an implementation can choose between always prescanning the first time a page is encountered, either on a page or document basis, or possibly re-rendering the page if it should turn out to be necessary. Our implementation supports both possibilities.

Essentially, the code provided flattens all context specials and attribute settings to a simple sequence of attribute sets and pops. All page specials are moved to the very beginning of a page, and all document specials are moved to the very beginning of a document. The `dvidvi` program provided does this from the provided `dvi` file; for all other drivers, this special translation and movement happens dynamically.

When a new `dvi` file is started, the driver is responsible for calling `initcontexts()` to initialize the various data structures. At the beginning and end of each page, the driver should call `bopcontexts()` and `eopcontexts()`. These need not come in matched pairs; if page rendering is interrupted for any reason (such as the user selecting the next page before rendering is completed) the driver must not call `eopcontexts()` but should instead simply call `bopcontexts()` for the next page.

The exception to this, of course, is that each page must be fully scanned at least once, and `eopcontexts()` called, before any subsequent page can be rendered.

The driver must provide the subroutine `dospecial()` that is responsible for parsing and understanding specials in the normal manner. Typically this already exists in almost all drivers. But rather than calling this subroutine every time a special is encountered, the driver should instead call the supplied routine `contextspecial()`. This subroutine will check if the special is one of the context specials described here, and if so, translate it to the appropriate flat specials, calling `dospecial()` for each one. If the special is not a context special, then the driver's `dospecial()` routine is invoked.

If the special was a page special or a document special, and this is the first time this page has been encountered, `contextspecial()` will return the special value `RERENDER` indicating that the driver should consider rerendering the page from the beginning after performing (finishing) a prescan. If the driver has not yet rendered any characters or rules, or if the driver is scanning rather than rendering, this return code can be ignored.

To identify pages, the driver should also provide a routine called `dviloc()` that returns a long value indicating the byte position in the `dvi` file.

The call to `bopcontexts()` at the beginning of a page may cause the driver's `dospecial()` routine to be invoked many times, once for every outstanding page attribute and local attribute.

To handle document global specials, the entire first page must always be fully prescanned.

The way the code works is as follows. At the beginning of each page that has not been previously encountered, the full stack contents of each context are saved and associated with the `dvi` file location for that page. If the page has been encountered, then the stack contents are restored, issuing any necessary set attribute specials for current attributes in the global context. In addition, any page attribute values are set. The context stack is set to hold just the global context.

When a push context special is encountered, the context associated with that name is found. If none exists, one is allocated. If the context stack has more than just the global context, then the attribute values from the context on top of the context stack are hidden. In any case, the attribute values for the context being pushed are made visible.

Attribute values are hidden by searching for the same attribute in the global context. If one exists, then its value is emitted with a flat set attribute special. Otherwise, the value is reset with a pop attribute special.

Attribute values are made visible by simply executing a flat set attribute special for each value.

When a pop context special is encountered, the context stack is checked to make sure it has at least two entries. If not, an error routine is called. Otherwise, the top context is popped, and all attribute values in that context are hidden. If the resulting context stack has more than just one context, then the attributes in that context are made visible.

When an attribute push special is encountered, then the attribute name and value pair are added to the current context, and the new value is made visible.

When an attribute set special is encountered, if the context on top of the context stack has such an attribute, than that attribute is changed and the new value made visible. Otherwise, the set attribute special is treated precisely as though it were a push attribute special.

When an attribute pop special is encountered, the context on top of the context stack is searched for that attribute. If the context has no such attribute, an error is reported. Otherwise, the attribute is hidden, and the attribute/value pair is popped from the context. If the same attribute exists in the current context (further down on the stack), then that attribute value is made visible.

Note that attributes do not need to nest "correctly"; the following sequence is legal:

```
attribute push color red
   <text>
attribute push changebar on
   <text>
attribute pop color
   <text>
attribute pop changebar
```

In addition, pushing and popping contexts simply makes them visible and hidden; it does not affect their values. Thus, assuming that the global context is on the context stack, after the following sequence, the color in the global context will be green:

```
attribute push color red
context push header
context push global
attribute set color green
context pop
context pop
```

Backwards compatibility. For backwards compatibility, existing `dvips` specials are fully supported. Most specials fall into the object category and are automatically passed through to `dospecial()`. These specials include those for EPSF inclusion and literal PostScript code.

The existing color macros are trivially supported by translation. The existing color macros never change contexts (they always use the implicit global context), so the semantics are unchanged with one exception. The explicit color set macro is now legal even when there are colors on the color stack; only the topmost entry on the stack is affected.

The four specials `header`, `papersize`, `landscape`, and `!` are document global specials and are translated as such. The next release of `dvips` will also allow `papersize` and `landscape` specials to apply on a page basis.

The code implementing these color specials, along with documentation describing how to use the code, is available in both `dvips` and `dvidvi` on `labrea.stanford.edu`.

Future work. We plan to continue the development of special capabilities using this form of interface. In particular, we hope to add support for colored box backgrounds, changebars, and similar things through a simple language. As we or others enhance the released code, any drivers that use this will automatically get the new capabilities. And, the `dvidvi` program will provide full support for these specials for those drivers that don't use the code.

Acknowledgments. The ideas in this paper are primarily derived from discussions with James Hafner, David Carlisle, Leslie Lamport, Frank Mittelbach, Sebastian Rahtz, and Tim Van Zandt. The confusion and complexity is attributable to me. I can only hope that this code will evolve quickly and stabilize into a useful and powerful base for using color in TEX.