A LaTeX style file generator and editor

Jon Stenerson
TCI Software Research, Las Cruces, New Mexico
Jon_Stenerson@tcisoft.com

Abstract

This article presents a program that facilitates the creation of customized LaTeX style files. The user provides a style specification and the style editor writes all the macros. Editing takes place in a graphical user interface composed of windows, menus, and dialog boxes. While the editor may be used in any LaTeX environment, it is intended primarily for use with TCI Software Research's word processor Scientific Word.

The current style editor runs under any Windows 3.1 system. The performance is acceptable on a 386-based machine and naturally improves on 486's and Pentiums. As Scientific Word is ported to other systems so will the style editor be ported.

Introduction

The style editor is a program that facilitates the creation and modification of *styles*. It represents a style as a list of *generic markup tags*, and thinks of a tag as a list of *parameters* which determine its typesetting properties. It performs the basic operations of creating a new tag, modifying a tag's parameters, and deleting a tag. A tag's formatting instructions are not explicitly displayed. That is to say you do not see any TeX on the screen. Instead you see dialog boxes containing icons, menus, radio buttons, check boxes, and so forth. These prompt you to specify the style by *filling in* parameters and selecting options. There are some screen shots at the end of this article to give an idea of the style editor's general appearance.

Styles, generic markup tags, and Scientific Word

A *generic markup tag* is a device by which an author specifies a document's logical structure without specifying its visual format. For instance, the LaTeX tag `\section` conveys the information that a new section is beginning and that the tagged text is its title. By itself this has no implications for the *appearance* of the section heading. It does not tell us the heading's font, justification, or vertical spacing. A *style file*, external to the document, contains associations between the tag names and specific typesetting instructions. The style file says what tags exist and how text marked with those tags should be typeset. We see that the use of generic markup tags provides a certain division of labor. I write the article, someone else writes the style, and TeX and LaTeX do the typesetting. The only style information I need as an author is a list of tag names and instructions for their use.

These days most word processors do not make use of generic markup tags. The reason is that they want to be WYSIWYG (what you see is what you get). This means that they display on the video monitor exactly what you will get when you print the final copy. Files produced by WYSIWYG word processors are filled with explicit typesetting instructions like "put a 14pt Helvetica A at coordinates (100, 112)." Compare this approach with the generic markup approach. First, the division of labor mentioned above is lost and the author is now responsible for all typesetting decisions. Of course this is also the main attraction of such systems. Second, stylistic information is now duplicated throughout the document. If subsection headings have to be left justified rather than centered the author will have to track them all down and change them one by one.

At TCI Software Research we are trying the generic markup approach to word processing. Our word processor, Scientific Word, is not a WYSIWYG word processor in the usual sense. Instead it displays a document's text plus markup. The markup is graphical, rather than textual, in nature. Whereas in LaTeX you will see `\section{Introduction}`, in Scientific Word you will see the word Introduction in large blue letters on the video monitor. Ideally the document's text plus markup tags represents the entire content of the document. In practice there are some important exceptions where visual formatting carries a lot of information. For example, in mathematical equations and in tables the precise positioning of text contributes enormously to its meaning. Scientific Word is WYSIWYG to the extent that if the appearance of an object carries meaning, as in the case of an equation or table, then that object is displayed in an approximation to its printed form. When Scientific Word saves a document on a disk it is saved in LaTeX form, and from there it can be typeset and printed.

Being "LaTeX-oriented" leaves Scientific Word open to some of the same criticisms leveled at LaTeX. In particular, authors do not always appreciate the division of labor I mentioned above. Some of them need or want more control over the style and cannot accept that someone else just hands them a style. At TCI we receive hundreds of requests for style modifications each year. Most of them are quite straightforward but many are not. It frustrates our customers, used to WYSIWYG systems, that some apparently trivial operations are not trivial for the casual LaTeX user. This suggests the need for another division of labor: style designer versus style writer. Our authors do not actually want to write styles, they want to specify styles. I was assigned the task of developing tools to alleviate this problem. The style editor represents the current state of that research.

For further discussion of markup tags and LaTeX see the first couple chapters of Goossens, Mittelbach, and Samarin 1994. For a discussion of generic markup in a non-LaTeX environment read about SGML (Standard Generalized Markup Language) (Bryan 1988).

The development process

Before continuing with the style editor itself I'd like to talk a little about the process of designing and implementing the editor. I was trained as an algebraic geometer in graduate school, had previously worked as a math professor, and this was my first professional programming experience. The process of programming is still novel enough to me that I feel like writing about it.

The first part of my research was to work with our customers in the capacity of style writer. I did this for four months to learn TeX, to learn how to think about style issues, and to find out what our customers wanted in the way of style modification. When I had enough experience to contemplate writing a program I e-mailed 500 customers and asked if anyone was interested in the design of a style editor. About 45 people responded and provided numerous comments and suggestions.

Still not knowing what a style editor should look like I decided to make a prototype, learn from my mistakes, and then build a release version.The prototype was implemented in three months between December 1993 and February 1994. It was complete enough to handle some realistic design issues even though it did not have a nice user interface. I wrote several styles with it including a style for one chapter of the new Scientific Word User's Guide.

In retrospect, I think that I spent the wrong amount of time on the prototype. The last few weeks of work on the prototype were spent getting it ready for testers - adding minor features, fixing bugs and writing documentation. As it turned out the testers paid little attention to the prototype editor. It was too primitive and too scary and I didn't get the feedback I'd hoped for. I either should have either gone ahead and made a nicer and more polished interface for the prototype, or I should have quit earlier and started on the release version editor sooner.

I learned many things from the prototype:

- Most importantly I learned that it is possible to develop a useful style editor. This was not obvious to me at first, but much of what I did worked better than I thought it would. I am now confident that TCI can and will develop a style editor that allows the casual user with no LaTeX knowledge to make basic style changes, and allows the advanced user to create any style at all.
- I learned that a lot more attention had to be paid to the user interface. I did not spend much time on the prototype's user interface because I had to first concentrate on getting the right model for the styles and getting the right basic functionality. For the style editor release version we added another programmer, Chris Gorman, to concentrate on getting the user interface in shape. He is responsible for much of the slick look and feel of the final program.
- Using the completed prototype to write some actual styles uncovered a number of flaws in the model I was using to represent styles.
- Writing the code for the prototype uncovered a number of flaws in my programming technique. Actually, many of these flaws were uncovered by John Mackendrick, one of our inhouse testers. I am a better programmer than I was six months ago. While the prototype always seemed a little flaky and buggy, the new program seems much more robust just by virtue of being better written.

Overall design

The style editor consists of the following components:

1. A GUI (Graphical User Interface). This manages interaction with the user and with the platform. The only platform Chris and I have worked on so far is Windows 3.1. We used Microsoft's Visual C++ and their MFC (Microsoft Foundation Classes) application framework. My understanding is that MFC code is supposed to eventually be portable to other platforms (Apple's Macintosh and Unix). So when Microsoft finishes MFC on those platforms we should be able to port the style editor.
2. A data structure called the Style. The program actually represents the style in two different

forms: internal and external. The internal form is a set of C++ classes suitable for editing. The nature of these classes lies outside the scope of this article. The external form is textual, suitable for interpretation as a LaTeX style and for human perusal. I frequently lump the internal and external data structures together into one abstract concept that I call the style *model*.

3. A set of functions called the "core". These functions do the following;
   (a) convert between the two style representations. In other words they read and write style files.
   (b) perform error checking. For example, before saving a style it makes sure that every item referred to in the style has been defined in the style.
   (c) constructs other files needed by Scientific Word. Besides the style file there is also a *shell* file and a *screen appearance* file. Each style file has a shell file that is used as a template whenever Scientific Word creates a new document of that style. The screen appearance file tells Scientific Word what tags are in the style and determines how they will appear on the video display.
4. A set of *macro writers*. These are TeX macros that interpret the style editor output as an actual style. They accomplish this by reading the style file and writing macros to implement the tags described in the style. This is all done on the fly. You will not normally see the macros written by the macro writers. They are constructed in the computer's memory and do not assume any printed appearance without inserting a `\show` command.

The key to the style editor is the last item so I'll talk about it some more. The *macro writers* are contained in a file called `sebase.cls`. This file is used as the document class for any style editor style. This is a misuse of the `.cls` extension because `sebase.cls` does not define any document class. Nor does it define any macros that may be used to markup a document. Rather it is a toolbox. The tools in `sebase.cls` are used to automatically write the macros that will be used in document markup. Eventually I will make a format file out of `sebase` but for now it depends on using the LaTeX format. Style files generated by the editor are read in with a `\usepackage` command.

Here is an example. In my scheme the definition of a section tag would look something like this:

```
\Division{
  \Name{section}
  \Level{1}
  \Heading{SectionHeading}
  \EnterTOC{true}
  \StartsOn{NextPage}
  \SetRightMark{true}
}
```

This is somewhat simplified but it gives the basic idea of what the style editor output might look like. In the file `sebase` there is a macro writer called `\Division` that writes a document division[1] macro on the basis of its parameters. In this case it writes a macro named `\section`. You see parameters describing the division's behavior with regard to the table of contents and running header and whether it must start on a new page, but you do not see any formatting instructions for a heading. This is because I distinguish between the division and its heading. There is just a reference to a heading. The heading itself is defined like this:

```
\DisplayElement{
  \Name{SectionHeading}
  \SkipBefore{20pt plus 4pt minus 2pt}
  \SkipAfter{12pt plus 2 pt minus 1pt}
  \ParagraphType{HeadingParagraph}
  \Font{MajorHeadingFont}
  \Components{
     Section \sectionCount .\Space{2mm}
     \CurrentHeading}
}
```

I have around 20 macro writers. Each of these is responsible for writing a certain *category* of macro. Thus I have a Division category, a Display Element category, a List category, a Font category, and so forth. These are discussed in more detail in the next section.

To get a feeling for how an editing session proceeds look at the screen shots at the end of this article. The first shows the start-up screen. You can see various controls for adjusting margins and page sizes. At the top of the screen is a menu labeled Category. The second screen shot shows the category menu pulled down and the division category about to be selected. You can see all of the categories. The third screen shot shows the screen after selecting the division category. Look at the split screen window. The left part of the window lists all the instances of the category that have been defined so far. In this case it lists all of the style's divisions: chapter, section, subsection and appendix. This list may be added to or deleted from. The figure also shows that "section" has been selected from the list of all divisions. The information for the section division is displayed in a dialog box contained in the right pane of the split window. This dialog changes radically depending on the category. One uses the controls found in that pane to inspect or alter the displayed

[1] I started using the term "division" because I found it awkward to continually refer to sections, subsections, and chapters as "sections".

parameters. When the style is saved the information is written in a form similar to that shown above.

Parametrized macro writing is not a new idea. For example, in the code for the LaTeX format there is a macro called `\@startsection`. This macro is used to define sectioning macros. It has a number of parameters and by specifying various values for these parameters one defines a wide variety of sectioning commands. Here is a typical definition of the `\section` tag from a human authored style file:

```
\def\section{
  \@startsection {section}{1}{\z@}
       {3.5ex plus 1ex minus .2ex}
       {2.3ex plus .2ex}{\large\bf}}
```

Only a dedicated person can remember what those parameters do, or that if one is negative it has a different meaning than if it is non-negative. On the other hand I have noticed that many styles override `\@startsection` itself, suggesting that it may not have enough parameters! In addition to borrowing ideas from LaTeX I have found that Bechtolsheim's *TeX in Practice* (Bechtolsheim 1993) is an excellent source of ideas for parametrized macros.

The idea of macro-writing macros is also not new. A trivial example is the `\title` macro found in LaTeX styles. It is defined like this:

```
\def\title#1{
  \def\@title{#1}
}
```

It takes a parameter and uses it to write another macro.

Victor Eijkhout's Lollipop format (Eijkhout 1992) is an example of a complete system of macro writing tools. I have not had an opportunity to use Lollipop but from the article I suspect that it would be possible to put a user interface on it similar to the one used with sebase. I thank the anonymous reviewer of this article for pointing out the existence of Eijkhout's work. I am a relative newcomer to TeX and was not aware of Lollipop but it is clearly related to what I am doing. Since I don't know Lollipop I will quote verbatim an example from the reference showing how a subsection heading might be created in that system:

```
\DefineHeading:SubSection counter:i
    whitebefore:18pt  whiteafter:15pt
    Pointsize:14  Style:bold
    block:start  SectionCounter literal:,
      SubsectionCounter literal:.
      fillupto:levelindent title
    external:Contents title external:stop
    Stop
```

You can see that this uses the idea of defining macros by specifying parameters in the form of keyword plus value.

A model for styles

Preliminaries. I think that a good piece of software must be based on a clean and straightforward model. In the case of the style editor this means finding an abstract representation of a style. My initial reaction after learning TeX and thinking about styles for a while was that it was not possible to write a style editor. There seemed to be so much disorganized "stuff" that I had no idea where to start. Had I started programming at this point I probably would have picked for my model a particular style file, say `article.sty`, and my program would have been an expert at editing all of the parameters and options found in this file. Instead I had a few "modelling" talks with Roger Hunter (TCI's president) and Andy Canham (development team leader). The model that came out of those meetings was implemented in the prototype and was subsequently modified for the release version based on that experience.

I said before that the model has two concrete representations: one as a C++ class, the other as a style file. The latter is probably more familiar to the reader so we will identify the style file with the style model. The remainder of this section talks about style files written by the style editor. The main idea behind style editor style files is that they contain no algorithmic information. There are no sequences of instructions, no branches, and no loops. They consist only of a long list of declarative information. Style editor style files use a very uniform syntax for this declarative data and therefore look different from other style files.

The style file consists of a list of declarations. The syntax for a declaration is always the same:

```
\CategoryName{
   \Parameter1{value 1}
   \Parameter2{value 2}
   ...etc...
}
```

Every category requires a fixed number and type of parameters. Parameters are discussed in the following subsection, and categories in the subsection after that.

The samples shown below are simplified. Actual style editor files contain information related to the operation of the style editor program. They also contain multiple versions of style data related to features described in the section on the user interface. I will suppress these kinds of data in the following discussion.

There is nothing proprietary about style editor style files. Anyone can go in with an ASCII editor and make changes to them without the style editor. For that matter anyone can write an entire style editor style file without using the style editor.

Parameters. Every parameter expects a value of a particular type. I've found the following types of parameters to be adequate:

1. A word parameter requires a string of letters (A–Z, a–z). These are usually references to macros.
2. A text parameter requires a string of characters. Any characters that are not among TeX's special characters (like braces and dollar signs) are allowed.
3. A boolean parameter requires one of the two words "true" or "false".
4. A numeric parameter requires a signed decimal number.
5. A dimension parameter requires a dimension in the TeX sense of a number plus a unit. The style editor knows all of the TeX units and can convert between them.
6. A glue parameter requires a glue value in the TeX sense of a natural dimension with a stretch dimension and a shrink dimension.
7. A component list parameter requires a list of *components*. Each component is either text in the sense given above, or a control sequence which is called a reference component in the style editor.

Some of these were demonstrated in the previous section's example of a `\Division`: `\Heading` is a word parameter, `\EnterInTOC` is a boolean parameter, and `\Level` is a numeric parameter. Next look at the `\DisplayElement` example also in the previous section. `\SkipBefore` and `\SkipAfter` are glue parameters and `\Components` is a component list parameter. The value of `\Components` in the example consists of five components: two text components "Section " and ".", and three reference components.

Categories. Now we'll take a look at some of the other categories that the style editor knows about. There are more categories than I can describe even briefly so I'm just going to try get across a few ideas about how it all fits together. In particular we will not see categories that define Lists, Table of Contents, Index, Bibliography, or Math. These perform fairly specialized functions and after reading what follows you may be able to imagine their nature.

Document Variables. These are macros that the document uses to pass information back to the style. A typical example is a macro to handle the document's title:

```
\DocumentVariable{
  \Name{Title}
  ...
}
```

A document variable's most important parameter is its name. It actually has a couple more parameters that have to do with Scientific Word's handling of the variable. The macro writer, `\DocumentVariable`, writes a macro called `\SetTitle`. The `\SetTitle` macro is used in the document like this:

```
\SetTitle{My TUG paper}
```

This in turn defines a macro `\Title` whose replacement text is `My TUG paper`. Thus `\SetTitle` and `\Title` have the same relation to each other as `\title` and `\@title` have in LaTeX.

The style editor also knows about several built-in macros that get information from the document. These include `\PageNum`, and `\CurrentHeading`. These keep track of the current page number and the title of the most recently encountered division.

Fonts. The font category provides an interface to NFSS. Here is a sample style file entry:

```
\FontNFSS{
  \Name{BodyTextFont}
  \Family{Serif}
  \Shape{Upright}
  \Series{Medium}
  \Size{normalsize}
}
```

`\FontNFSS` will write a macro, `\BodyTextFont`, which performs the indicated font switch. The precise nature of the various families, shapes, series, and sizes are determined by selecting a "Font Scheme" elsewhere in the style.

Paragraphs and Environments. The paragraph category provides an interface to a number of TeX parameters related to paragraph typesetting: font, baseline-to-baseline distance, indentations and so forth. By setting these properly you can create tags like the `\quote` and `\center` found in LaTeX. Here is an example:

```
\Paragraph{
  \Name{Center}
  \Font{BodyTextFont}
  \ParIndent{0pt}
  \LeftIndent{0pt plus 1fil}
  \RightIndent{0pt plus 1fil}
  \ParFillSkip{0pt}
  \ParSkip{0pt}
  \PageBreakPenalty{100}
  \HyphenationPenalty{100}
}
```

When used in conjunction with an environment category item this will make available in the document an environment `\begin{Center} ... \end{Center}` that typesets a prefix, such as a vertical skip, then switches to the centering paragraph, and then has a suffix.

In-line and display elements. An in-line element is just a component list plus a font. It is intended to typeset text which is part of a surrounding paragraph. Here is an example:

```
\InlineElement{
  \Name{AbstractLeadin}
  \Font{SmallCapFont}
  \Components{Abstract.\Space{1pc}}
}
```

This creates a macro, \AbstractLeadin, which typesets the word "Abstract" followed by a period and a space. It uses a font called \SmallCapFont which must be defined via the Font category. Component lists may use the names of in-line elements so this \AbstractLeadin item may be reused throughout the style.

A display element is intended to be typeset in its own paragraph and set off from the surrounding text. We have already seen an example of this earlier in the section.

In-line and display elements are frequently used in conjunction with a document variable. For example, consider generating a macro to typeset the title of the document. We would first declare a document variable to hold the title

```
\DocumentVariable{
  \Name{Title}
}
```

and then declare a display element that uses the document variable

```
\DisplayElement{
  \Name{TITLE}
  \SkipBefore{0pt}
  \SkipAfter{0pt}
  \Paragraph{CenterHeading}
  \Font{MajorHeadingFont}
  \Components{\Title}
}
```

This produces a macro called \TITLE that typesets the value of the variable \Title with the given paragraph and font settings. The \TITLE macro may be used in the document but will probably be used in a title page macro (see below).

Page Setup. This category provides an interface to many TEX parameters involved in page style: page size, trim size, margins, headers and footers, footnotes and margin notes. Most styles will need to create only one item in the page setup category.

Exceptional Pages. An exceptional page is one that deviates from the surrounding pages in that it has some special formatting requirements. A typical example is a title page. A title page has some specially typeset material and usually has special headers and footers. Here is an example:

```
\Exception{
  \Name{TitlePage}
  \VerticalMaterial{
     \Space{2cm}
     \TITLE
     \Space{1cm}
     \AUTHOR
     \DATE
     \Space{1cm}
  }
  \ContinueTextOn{ThisPage}
  \SpecialLeftHead{}
  \SpecialMiddleHead{}
  \SpecialRightHead{}
  ...etc...
}
```

This writes a macro called \TitlePage which in turn causes a new page to begin, typesets the vertical material, and then allows text to continue on this page. The vertical material consists of built-in macros such as \Space or names of elements defined elsewhere in the style such as \TITLE, \AUTHOR, and \DATE.

The user interface

The prototype editor had a simple interface. In essence there were dialog boxes in one-to-one correspondence with the macro writers and in each dialog box there were controls in one-to-one correspondence with the macro writer's parameters. To a TEX programmer this interface would probably seem pretty friendly. If you saw an edit control labeled "Par. Skip" you'd probably have a good idea of the sort of thing you might enter. Editing with the prototype was not that far removed from editing the style file with an ASCII editor. The major step forward was the ease with which you could move around the style. I'm sure that all TEX programmers have had the experience of searching style files for a macro definition. The prototype style editor could find any piece of data instantly.

Most of our customers however do not want to fill in parameters. They do not want to know what glue is. They do not even want to see the word "skip" on the screen. They want to use the mouse to click on a picture of what they want, check a few boxes or radio buttons, and have the program do the right thing. On the other hand I liked the prototype's powerful interface and was not willing to give it up. So I opted for a hybrid scheme. A category item can now have two different interfaces: a "quick screen" in which a few simple options are presented, and a "custom screen" which presents all the category's parameters. The quick screen for the Paragraph category has several sets of icons. By selecting an icon from each set you determine certain characteristics of the paragraph. For example, one set is labeled "Paragraph spacing" and it contains two icons. One icon suggests tight spacing, the other suggests loose spacing. The custom screen by contrast has several places where actual dimension and glue values must be given. To prevent casual users from stumbling into dialogs they don't understand the program has two modes. In the first mode many features including all the custom screens are disabled.

As described so far the quick screen seems limited. It has two icons for paragraph spacing, but what glue values should correspond to these two icons? It clearly depends on the style. I therefore decided to let the icons themselves be programmable. By selecting an icon and pressing the F2 key you get a dialog box where a specific glue can be given. This value is saved in the style file. Finally, if you can't get the effect you want from the quick screen, the quick screen F2 modifications, or the custom screen, you can tell the style editor that you want to write this macro yourself. You will then have to do so in another macro file.

Conclusion

The style editor as it now stands is a useful program but there is still a lot of work to be done before it is a complete program. What I anticipate in the near future is that a style writer will prepare a style using the style editor together with a little straight TeX to fill in the gaps. The resulting style, at least those parts that do not rely on the plain TeX additions, can be customized by the author without any TeX knowhow. As time goes by I will manage to get more and more TeX into the editor's quick screens and there will be fewer and fewer gaps.

I have more basic functionality planned. For instance, I want to include a fancy "cut and paste" feature that will facilitate moving tag definitions from one style to another. The editor will resolve internal naming conflicts and make sure that auxiliary definitions needed for the tags being moved are moved at the same time. Having an abstract style representation should make it possible to move features from style to style. This in turn will make it possible to "change styles". A frequent customer request is to change a document from one style to another. If the two styles have the same set of markup tags this is pretty easy. If they do not this is pretty hard. If the style editor can reliably move tags from one style to another then this problem will be solved.

Shortly before the TUG meeting I received preprints of two other papers, Baxter 1994 and Ogawa 1994, that are found elsewhere in these proceedings. These talk are about using the object-oriented paradigm in TeX programming and in document markup. In some ways the style editor is also part of this discussion on the object-oriented approach. In fact the style editor directly represents the style as a C++ class in which each generic markup tag acts as a "style object" that can be acted upon by an object-oriented interface. I think that combining a style editor of the sort I've described here with a markup scheme such as described in the above references would lead to quite a powerful typesetting system.

References

Baxter, William Erik. "An Object-Oriented Programming System in TeX." These proceedings.

Bechtolsheim, Stephan von. *TeX in Practice*. Springer-Verlag, New York, NY, USA, 1993.

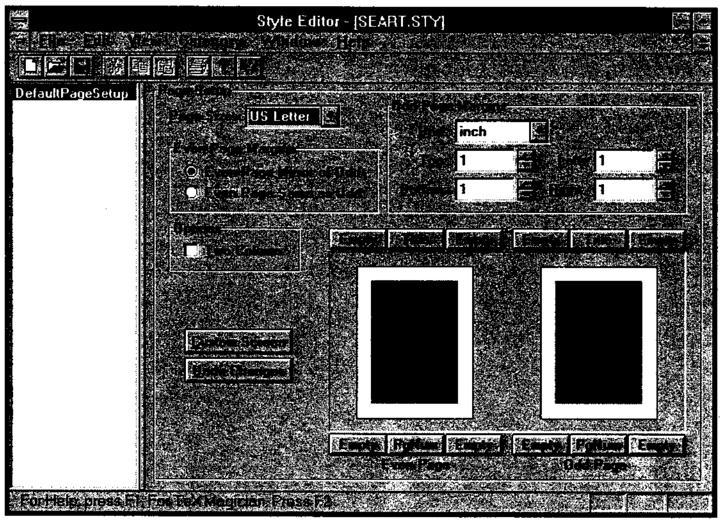
Eijkhout, Victor "Just give me a lollipop (it makes my heart go giddy-up)." *TUGboat* **13** (3), pages 341-346, 1992.

Goossens, Michel, Frank Mittelbach and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, MA, USA, 1994.

Mittelbach, Frank. "An extension of the LaTeX theorem environment." *TUGboat* **10** (3), pages 416-426, 1989.

Ogawa, Arthur. "Object-Oriented Programming, Descriptive Markup, and TeX." These proceedings.

Bryan, Martin. *SGML: an Author's Guide*. Addison-Wesley, Reading, MA, USA, 1988.



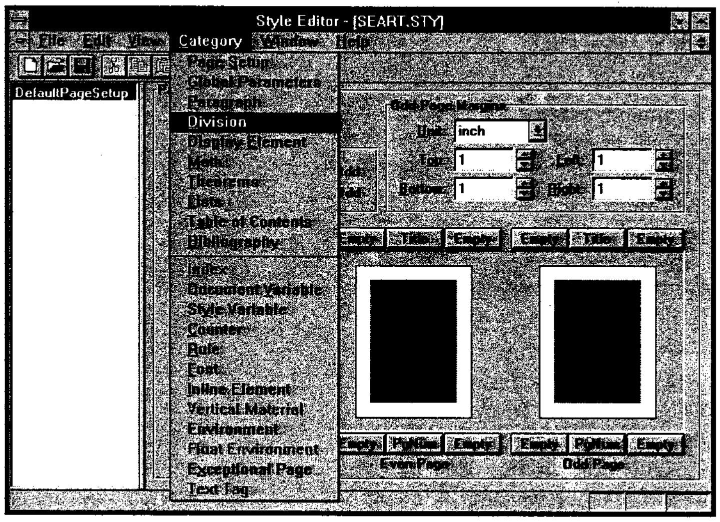
Page Size: US Letter

Unit: inch

Top: 1 Left: 1

Bottom: 1 Right: 1

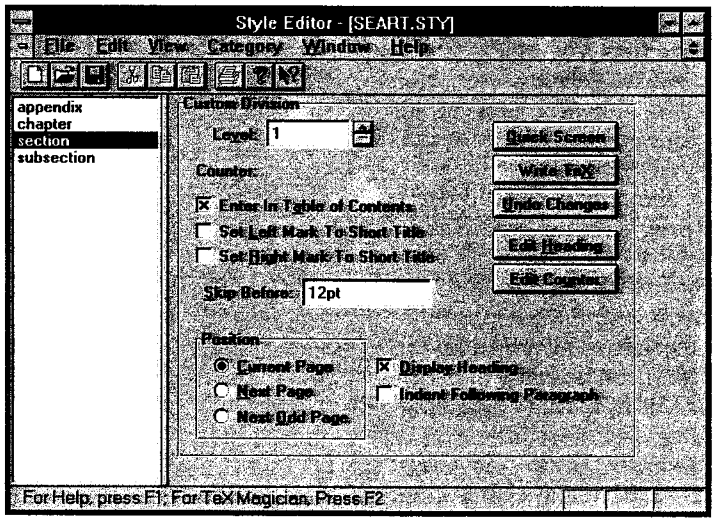
For Help, press F1; For TeX Magician, Press F2



Unit: inch

Top: 1 Left: 1

Bottom: 1 Right: 1



Custom Division

Level: 1

Counter:

[✓] Enter In Table of Contents

[ ] Set Left Mark To Short Title

[ ] Set Right Mark To Short Title

Skip Before: 12pt

Position

[✓ Current Page]

( ) Next Page

( ) Next Odd Page

[✓] Display Heading

[ ] Indent Following Paragraph

For Help, press F1; For TeX Magician, Press F2

Examples of user interface screens.