Here is some test input:

```
\storecat\%
\catcode`\%=3
    \storecat\%
    \catcode`\%=4
        \storecat\%
        \catcode`\%=5
        \restorecat\%
    \showthe\catcode`\%
    \restorecat%
\showthe\catcode`\%
\restorecat\%
\showthe\catcode`\%
```

As a whimsical aside, the double assignment to `\toks0` can also be done in one statement:

```
\toks0\xp\xp\xp\xp\xp\xp\xp
   \xp\xp\xp\xp\xp\xp\xp\xp
  {\xp\xp\xp\csname\xp\string\xp
   \restorecat\string#1\endcsname}
```

using 15 consecutive `\expandafter`s.

◇ Victor Eijkhout
Department of Computer Science
University of Tennessee
104 Ayres Hall
Knoxville, Tennessee 37996, USA
eijkhout@cs.utk.edu

ZzTEX: A Macro Package for Books

Paul C. Anagnostopoulos

Introduction

ZzTEX is a macro package for producing books, journals, and technical documentation. The primary advantage of ZzTEX is its design flexibility, which makes it well-suited to typesetting books according to the specifications of a professional book designer. During the past three years, I and my associates have used the package to produce approximately 25 books, ranging from the 100-page journal, *System Dynamics Review*, published by John Wiley & Sons, to the 1400-page book, *VMS Internals and Data Structures, Version 5.2*, published by Digital Press. In this article I hope to give you a taste of some of ZzTEX's more interesting technical aspects. Future articles will delve deeper into the details of the macros themselves.

I was initiated into the composition and typesetting business when I agreed to compose my own book for Digital Press. I had written the book using LATEX, and continued to use it for the composition. As a software engineer, I found it impossible not to fall in love with book production: finally, an endeavor that produces a concrete work of art as its end product, rather than some ethereal software. However, I needed more design and page-makeup flexibility than LATEX had to offer, so I undertook to write my own macro package, which I subsequently named ZzTEX, after a rock group from Texas. Design flexibility is of paramount importance when producing books according to typographers' designs; neither they nor the publishers like to hear "I'm sorry, that element is too difficult to typeset."

The first book produced with ZzTEX was a real struggle. It took about two weeks to create the design file and produce sample pages. With time, I refined the package and added many new features. Each enhancement was a direct result of a design requirement in a book I had produced, so I believe ZzTEX is a practical, realistic macro package. Furthermore, my knowledge of the publishing business grew with each book. Now when I receive a design specification I can produce sample pages in less than a day. The package includes approximately 7,200 lines of TEX code and various utilities written in AWK.

The ZzTEX macro package and manual are available from the author for a nominal fee. The macro package may be freely distributed, but the manual is copyrighted and must be ordered from the author. ZzTEX uses significant amounts of

TEX memory, so the author recommends a TEX implementation that has memory areas at least doubled in size.

The Block

In general, each of ZzTEX's typographic elements is assembled from a fundamental construct called the *block*. A block "contains" the text of the element, and separates its text from that of surrounding blocks. Here is an example of a block that produces a bulleted list:

```
\list{bullet}
\item This is the first item of the
bulleted list.
\item This is the second item of
the list.  It can contain many
lines of text, and even multiple
paragraphs.
\item This is the last item.
\endlist
```

The `\list` command begins a list block. The argument in braces is called the *block type*, which allows you to specify arbitrarily many list designs. The text of the list begins after the `\list` command and ends at the `\endlist` command. Each item in the list is initiated with the `\item` command.

The list block implementation in the ZzTEX macro package provides a "generic list formatter" that has the capability to produce almost any style of list. In order to format a particular type of list for a particular book design, the block accepts a set of *design parameters* that directs its formatting of that list (e.g., the `\leftindent` parameter determines the indentation of the list items). You are responsible for providing the design parameters for each type of list, placing them in your book's *design file* (see next section). The design file contains a *design macro* for each type of block element in the book; the design macro encloses the specifications of the block's design parameters.

ZzTEX provides approximately 35 kinds of blocks, many of which accept a type argument to allow an unlimited number of variations.

The power of the block lies in the steps that ZzTEX takes when it begins and ends a block. When a block is started, ZzTEX performs the following steps:

1. Automatically closes any blocks that are terminated by the new block. For example, the section block terminates a preceding section block.
2. Opens the block scope by starting a group. This hides any parameter changes made inside the block, allowing parameters to revert to their previous values when the block terminates.
3. Stores the `\baselineskip`, `\parskip`, and `\parindent` of the enclosing block in three special parameters, thus making the surrounding values available within the new block (after all, these parameters may be changed within the block).
4. Increments a block-specific depth counter.
5. Invokes the design macro for the block. Design macros are contained in the design file loaded by ZzTEX at the beginning of the run (see next section). In the case of lists, there is a separate design macro for each type of list. The design macro establishes a set of parameters that controls further processing of the block.
6. Increments a block-specific sequence counter. If appropriate, this counter can be used to number the instances of the block, as might be done with sections or footnotes.
7. Resets the sequence counter of any subordinate blocks. ZzTEX assumes the existence of certain block relationships, such as the standard section, subsection, and subsubsection hierarchy, and resets counters accordingly. Furthermore, ZzTEX provides the `\resetnumber` design parameter on most blocks, with which you can explicitly specify other sequence counters to be reset.
8. Invokes a command (defined in the design macro) that formats the *composite number* for the block. For example, the composite number for a subsection might be '3.2.5', for a table '3-8'.
9. Performs any formatting required at the beginning of the block, including vertical space above and perhaps a title.

Once the contents of the block are typeset and ZzTEX encounters the ending command, it performs these steps:

1. Checks to make sure that the ending command had a matching starting command.
2. Recursively closes any subordinate blocks that are still open (e.g., the `\endsection` command automatically closes any subsection in progress).
3. Performs any formatting required at the end of the block, including vertical space below.
4. Decrements the depth counter.
5. Closes the block scope, discarding any parameter changes made in the block.

The Design File

A book's design specification is embodied in the ZzTEX design file, which includes a design macro for each element in the book. The design macro for a particular element specifies values for various *design parameters* that determine the formatting of the element and control behind-the-scenes activities such as the generation of marks. In addition, the design file contains commands that establish the *font table*, a matrix that correlates type sizes and styles. The design file for a typical book might comprise 50 design macros and 200 font table commands, with a total of 800 lines.

Most of the elements in a book are realized in ZzTEX with a block. Your design file contains the design macros for all the blocks used in your book. There are no block designs embedded in ZzTEX, only generalized macros that can format a block given your design parameters. Therefore, as far as design is concerned, ZzTEX is a tabula rasa waiting for your book's design. You must always include a design macro for a special block called the *document block*. The document block design includes parameters that specify the overall design of the book, plus parameters that control the look of the main text paragraphs (e.g., their paragraph skip and indentation). Figure 1 illustrates the document block design macro for a book I recently completed. In addition, the design file might contain design macros for bulleted lists, numbered lists, and plain lists. It might contain a macro for computer program examples in running text, and another for program examples in figures. And it might contain macros for chapters, appendixes, sections, and subsections.

Figure 2 illustrates the bulleted list design macros from the same book. The name of the main list macro is `\listbulletidesign`: '`list`' is the name of the block, '`bullet`' is the block type, and '`i`' is the depth. Similarly, the name of the sublist macro is `\listbulletiidesign`. This naming scheme accommodates many types, or flavors, of the same block, and also different designs for first-, second-, and third-level nested lists. The first thing the sublist macro does is invoke the main list macro; this establishes all of the main list design parameters. Then the sublist macro redefines only those parameters that are different in the sublist design. Design macros can be arbitrarily interdefined in this manner. Another common reason for defining one block in terms of another occurs when you want a design for numbered lists. The first-level numbered list macro

```
\def \documentdesign {%
  \setflag\cropmarks = \true
  \eveninnermargin = 1in
  \evenlefttextmargin = 4pc
  \evenrighttextmargin = 0pt
  \footerheight = 26pt
  \headerheight = 17.5pt
  \headmargin = .5625in
  \hoffset = -.25in
  \maxbottomcolumnfloats = 3
  \maxtopcolumnfloats = 3
  \oddinnermargin = .646in
  \oddlefttextmargin = 4pc
  \oddrighttextmargin = 0pt
  \parindent = 10pt
  \parskip = 0pt
  \setflag\PostScriptoutput = \true
  \textareaheight = 526.5pt
  \textareawidth = 28pc
  \topskip = 13.5pt
  \trimheight = 9.25in
  \trimwidth = 7in
  \voffset = -.125in}
```

Figure 1

can invoke the first-level bulleted list macro, thereby sharing common parameters such as `\aboveskip`, `\belowskip`, and `\bodyfont`.

Careful page composition often requires that individual blocks be adjusted. The `\with` command is used as a prefix on a block command to alter one or more design parameters for that particular instance of the block. This is how you can change the space above and below a list:

```
\with{\aboveskip=18pt plus 1.2pt
      \belowskip=\aboveskip}
\list{bullet}
```

ZzTEX performs `\with` assignments *after* it invokes the block's design macro.

The Font Table

One of my primary goals in creating ZzTEX was to allow complete flexibility in selecting fonts. In fact, I have never typeset a book using Computer Modern Roman. (I have used Computer Modern Typewriter, because many book designers realize it is better than other available monospaced fonts.) ZzTEX employs a *font table* to select fonts. You can think of the font table as a matrix with rows corresponding to type sizes and columns to type styles. Figure 3 illustrates a simple font table.

Three steps are required to set up the font table:

```
\def \listbulletidesign {%
  \aboveskip = 21pt plus 1.6pt
                    minus 3.1pt
  \belowskip = \aboveskip
  \bodyfont = {}% Same as surrounding.
  \interitemskip = 15pt
  \def \labelformat ##1{##1\hfil}%
  \labelshift = -\enclosingparindent
  \def \labeltext {%
    \centeronxheight{\bul .}}%
  \labelwidth = \enclosingparindent
  \leftindent = \labelwidth
  \parindent = 0pt
  \parskip = 6pt
  \rightindent = 0pt
  \width = \naturalwidth}

\def \listbulletiidesign {%
  \listbulletidesign
  \aboveskip = 18.25pt plus 1pt
                       minus 2.7pt
  \belowskip = \aboveskip
  \labelshift = -8pt
  \def \labeltext {--}%
  \labelwidth = 8pt
  \leftindent = \labelwidth}
```

Figure 2

	Type Style		
Type size	\rm	\it	\dbf
\chapsize	–	–	24' Optima Bold
\aheadsize	–	–	12' Optima Bold
\textsize	10' Nofret Regular	10' Nofret Italic	–
\ftntsize	7' Nofret Regular	7' Nofret Italic	–

Figure 3

1. Define any type styles you need in addition to the built-in ones (Roman, math italic, math symbol, math extended symbol, PostScript symbol, italic, bold, bold italic, and typewriter). The definition includes a specification of the character set encoding used by the style (e.g., Roman vs. italic vs. monospace). Knowing the encoding, ZzTEX can, for example, automatically insert an italic correction after italic text.
2. Define all the fonts you need.
3. Define the logical type sizes you need (there are no built-in ones). The size definition includes the baseline-to-baseline distance. After each size definition, set the fonts for those styles that appear in that size. Styles that are used in math require three fonts (text, script, and scriptscript); other styles require one font.

```
\definetypestyle{\sb}{\encoderoman}
\definetypestyle{\sbi}{\encodeitalic}
\definetypestyle{\bul}{\encoderoman}
\definetypestyle{\drm}{\encoderoman}
\definetypestyle{\dbf}{\encoderoman}

\definefont{\twentyfourdrm}{ss at 24pt}
\definefont{\twentyfourdbf}{ssb at 24pt}
\definefont{\eighteendrm}{ss at 18pt}
\definefont{\ninetqtt}{cmtt10 at 9.75pt}
\definefont{\nineqrm}{sr at 9.25pt}
\definefont{\nineqmit}{cmmi10 at 9.25pt}
\definefont{\nineqmsy}{cmsy10 at 9.25pt}
\definefont{\nineqmex}{cmex10 at 9.25pt}
\definefont{\nineqit}{sri at 9.25pt}
\definefont{\nineqsb}{srsb at 9.25pt}
\definefont{\nineqsbi}{srsbi at 9.25pt}
\definefont{\sevenrm}{sr at 7pt}
\definefont{\sevenmit}{cmmi10 at 7pt}
\definefont{\sevenmsy}{cmsy10 at 7pt}
\definefont{\sevenit}{sri at 7pt}

\definetypesize{\chaptersize}{24/28}
\setfont{\chaptersize}{\drm}
        {\twentyfourdrm}
\setfont{\chaptersize}{\dbf}
        {\twentyfourdbf}

\definetypesize{\textsize}{9.25/13.5}
\setfontmath{\textsize}{\rm}{\nineqrm}
            {\sevenrm}{\fiverm}
\setfontmath{\textsize}{\mit}{\nineqmit}
            {\sevenmit}{\fivemit}
\setfontmath{\textsize}{\msy}{\nineqmsy}
            {\sevenmsy}{\fivemsy}
\setfontmath{\textsize}{\mex}{\nineqmex}
            {\nineqmex}{\nineqmex}
\setfontmath{\textsize}{\it}{\nineqit}
            {\sevenit}{\fiveit}
\setfont{\textsize}{\tt}{\ninetqtt}
\setfont{\textsize}{\sb}{\nineqsb}
\setfont{\textsize}{\sbi}{\nineqsbi}
\setfont{\textsize}{\bul}{\eighteendrm}
```

Figure 4

Figure 4 shows a portion of the font table for a book.

```
\input zztex

\selectdesign{merusi}

\document
\copyident{\sevenrm Merusi first pages}
\printart{\true}

\setdivisions{chap1,chap2}

\division{front}
\setpagenumber{1}
\setfoliostyle{\decimal}
\division{chap1}
\division{chap2}
\division{chap3}
\division{chap4}
\division{chap5}
\division{appa}
\division{appb}
\division{index}

\enddocument
```

Figure 5

In addition to the main font table, ZzTEX provides a second table that specifies *style relationships*. There is a built-in relationship called `\emph` that you use to produce emphasis. The relationship table specifies that Roman is emphasized with italic, and vice versa. Furthermore, bold is emphasized with bold italic, and vice versa. You can add additional entries for emphasis, and invent your own relationships such as `\smallcaps`, `\newterm`, or `\varname`.

The Division

Any book longer than about 20 pages is best broken into *divisions*, each of which typically contains the material in one chapter. The entire book is represented by a *root file*, and the root file incorporates each division with a `\division` command. Figure 5 shows a small root file. You can use the `\setdivisions` command to select specific divisions for processing.

Associated with the root file and each division file is a *division cross-reference file* that contains the following information:

- An entry for each title that should appear in a table of contents, list of figures, list of tables, or a similar listing of other floating elements. ZzTEX allows you to define additional types of floating elements (e.g., computer program code) and produce a listing of those elements.
- An entry for each symbolic tag that is referred to elsewhere in the book.
- An entry for each endnote.
- A single *snapshot* at the end. The snapshot contains the division's final page number, chapter number, section number, and so forth.

At the end of a run, ZzTEX combines all the root and division cross-reference files into one *composite cross-reference file*. The composite file is considered the master list of cross-reference information.

At the beginning of a run, ZzTEX:

1. Loads the composite cross-reference file to obtain the symbolic cross-reference tags. (All tags in a book must be unique, so if you are producing a book from multiple independent authors, you may have to alter tags during conversion to make them unique.)
2. Starts a cross-reference file for the root file.

For each division, ZzTEX:

1. Adds an entry to the root cross-reference file that names the division cross-reference file.
2. Creates the division cross-reference file.
3. Writes table of contents entries, cross-reference tags, and endnotes to the division file.
4. Finishes the division file with the snapshot of that division.

At the end of a *successful* run, ZzTEX:

1. Closes the root cross-reference file.
2. Combines the root and division cross-reference files into one composite cross-reference file.

Whenever ZzTEX needs cross-reference information, it consults the composite cross-reference file. Thus, when you send a book to someone else for processing, you send only the TEX source files and the single composite file. If TEX encounters an error processing your book, and you terminate the run, that division's cross-reference file is invalid, but the composite file still accurately reflects the previous run. So when you reprocess the book to correct the error, the cross-reference information is still intact. The second reprocessing run behaves just like the first run.

If the `\setdivisions` command excludes a division, ZzTEX does not process it. Instead, it searches the composite cross-reference file for the division's snapshot and updates important counters such as the page number and chapter number so they reflect the state of affairs *at the end* of the division. In this way, the next division processed will appear to be in the correct place in the book.

ZzTEX writes "moving arguments", such as titles, into the cross-reference files without expansion, so that no "protect" mechanism is needed. A special command, `\adjusttitle`, allows you to customize the format of a title where it appears in the main text, a table of contents, a running head, or a textual cross-reference. An adjustment can be as simple as a line break or as complicated as a footnote.

Vertical Spacing

One of the most difficult tasks I encountered in creating ZzTEX was to ensure consistent vertical space between elements. If the book designer requests 24 points base-to-base above an A-head and 16 points below, then ZzTEX must produce that much space, regardless of the element above the A-head, the size of the A-head, or whether there is text or a B-head immediately below it. Allowing some stretch and shrink above a heading does not diminish the need for consistent nominal space. I solved the problem with the *vertical spacing environment* (not to be confused with a LATEX environment).

ZzTEX provides six commands to produce vertical space:

- `\bbskipabove`. This command specifies a certain base-to-base space between the previous element and the next. If two or more of these commands appear in a row, the space from the *first* one prevails.
- `\bbskipbelow`. This command specifies a certain base-to-base space between the previous element and the next. This command is not used to produce vertical space at the end of a block (see next item). If two or more of these commands appear in a row, the space from the *last* one prevails.
- `\bbskipbelowblock`. This command is equivalent to `\bbskipbelow`, but must be used to produce vertical space at the end of a block. It compensates for any change in the `\baselineskip` and `\parskip` values that might occur after the block terminates.
- `\bbskipbelowblockpar`. This command performs the same functions as `\bbskipbelowblock`, but also checks whether the next element begins a new paragraph. If not, it ensures that the paragraph continuation is not indented.
- `\vsink`. This command specifies a certain base-to-base distance between the top of the type page and the next element. You can use it more than once on a single page, usually in front matter to format the half title or full title page.
- `\vspace`. This command adds additional vertical space between two elements that is independent of any other explicit or implicit base-to-base space between those elements. Thus it replaces `\vskip`.

The first four commands also accept an argument that specifies the page-break penalty inserted above the vertical space. If a `\bbskipabove` command follows one of the `\bbskipbelow` commands, the maximum of the two spaces is used. If the reverse occurs, an error is signaled.

In order for vertical space to be consistent, you must use these six commands wherever you request explicit vertical space. However, most vertical space is produced implicitly at the beginning and end of a block. Any block that produces vertical space accepts the `\aboveskip` and `\belowskip` design parameters, which specify the space above and below the block, respectively. The macros that implement the block use `\bbskipabove` to produce the space above the block, and `\bbskipbelowblock` or `\bbskipbelowblockpar` to produce the space below. Therefore, you usually only need `\vsink` to format pages such as title pages, or `\vspace` to force more or less space between certain elements for aesthetic purposes. You occasionally have to use `\bbskipabove` or `\bbskipbelow` to obtain the correct base-to-base distance between two elements that are not blocks (e.g., a title and subtitle).

The vertical spacing environment is maintained as a stack of structures, for reasons explained below. The structure at each level of the stack includes the following data items:

- The type of the previous vertical space: none; inter-paragraph space specified by `\parskip`; space above, produced by `\bbskipabove`; or space below, produced by `\bbskipbelow` and friends.
- The page-break penalty associated with the previous vertical space.
- The base-to-base space requested for the previous vertical space.
- The actual glue ZzTEX inserted for the previous vertical space.

By carefully inspecting and maintaining these items, the vertical spacing commands consistently produce the correct amount of space. The environment is on a stack because vertical spacing within some blocks, such as floating figures, is independent of the vertical spacing in progress in the surrounding text. When a

floating figure begins, it pushes the current vertical spacing environment on the stack, and reinitializes the environment. The vertical spacing within the figure thus begins afresh, unaffected by the space above the figure. When the figure block terminates, it pops the stack. Each level of the stack is simply implemented as a numbered definition containing the saved values of the environment items. The items for the top level are in global variables.

The Index

The only difficulty about typesetting an index from a prepared file of entries is producing carry-over lines when a first- or second-level entry continues onto a new page. However, creating that file of entries from indexing commands in the book is a challenge. I am rarely asked to do it, because most authors do not index their books, and many publishers prefer not to use authors' indexes. Nevertheless, to support those publishers and authors who want to generate an index automatically, I developed facilities to produce an index entry file from commands in the text.

To produce an index you must first define the required *index locators*. An index locator is a particular item of information associated with an entry. The most common index locator is a page number or page range. Another common locator is the "see also" locator that refers to another index entry. When you define a locator, you specify the following information:

- The name of the locator.
- A set of attributes. The `\page` attribute specifies that a page or range of pages is associated with the locator. The `\text` attribute specifies that arbitrary text is associated with it (e.g., another index heading).
- The sorting precedence of the locator. For example, "see also" locators have a lower precedence than page locators, so they appear after the page numbers in an entry.
- The prefix text. The prefix text for a "see also" entry in English is "*See also*".
- A template that specifies how to format a single page number or the text of the locator.
- A template that specifies how to format a page range.

ZzTEX predefines the following locators, in order of precedence: the null locator (for entries without any locators), the "see" locator, the page number/range locator, a locator for each type of floating object (for referring to tables, figures, etc.), the "see also" locator, and the "consult" locator (for referring to other books).

A locator definition creates one or more macros you can use throughout a book to produce index entries. If the locator is named `command` and has the `\page` attribute, then `\xcommand` produces a locator with the current page number. The `\xcommandbegin` and `\xcommandend` macros produce locators that begin and end a page range, respectively. Each macro accepts one, two, or three arguments designating up to three levels of index headings.

To prepare an index entry file, you include the `\prepareindex` command in the root file. This command specifies an index type, thereby allowing you to have more than one index in a book (e.g., a main index, an index of commands, and an index of variables). It also specifies the name of a *index preparation file* that receives all of the control information necessary to prepare an index: root file name, index type, list of locators included in this index, and the definitions of those locators. The `\prepareindex` command also activates indexing so that ZzTEX attends to indexing commands and writes the entries into the raw index files. Without a `\prepareindex` command, index entries in source files are ignored. There is one raw index file for the root and one file for each division (as with cross-reference files).

After a ZzTEX run, three steps are required to produce the index described by one particular index preparation file:

1. An AWK program consolidates all the locators in the raw index files into one composite index file. Prefixed to each record is a six-part key that includes the headings (canonicalized for sorting), precedence, segment number, and page number. The segment number is used to separate front matter pages from main body pages. This step discards any locators that should not be included in the index.
2. The composite index file is sorted.
3. Another AWK program processes the sorted index file and creates the TEX source file for the index. All of the locators for one entry are merged into a paragraph, using the prefix text, single-page template, and page-range template specified with each locator definition. The resulting file is suitable for inclusion in an `\index` block anywhere in the book.

Conversion To and From ZzTEX

My colleagues and I have typeset books from authors who used many different document preparation

systems, including Microsoft Word, troff, VAX DOCUMENT, DECwrite, TEX, and LATEX. Not only are there many document systems, but the degree to which an author uses the features of any given system varies greatly, as does the author's level of consistency. To facilitate the conversion of these books to ZzTEX, I developed a table-driven translator called ZzTran. Because ZzTran is based on AWK, you can specify regular expression patterns that match tags in the source language. For each pattern you specify how to generate the corresponding tag in the target language (usually ZzTEX), so that a translation file consists of a table of patterns and their replacements, along with auxiliary AWK functions you write to help with the more complicated tag translations.

To translate files, ZzTran runs an AWK program that "compiles" the translation table into a pure AWK program. The compiler incorporates both driver functions that control the translation process and a standard library of utility functions (e.g., `replace()` to replace a tag, `keep()` to maintain a tag as is). ZzTran then runs the resulting AWK program, which reads a source file and produces the corresponding target file. ZzTran can usually accomplish about 95% of the translation automatically, so that very little must be done by hand.

Occasionally, after typesetting, I am asked to translate the final ZzTEX files back to the author's original document preparation system. This is usually simple because the tags in ZzTEX files are more specific and complete than those in files acceptable to the author's system. The reverse translation is a matter of converting some ZzTEX tags to the original coding system and discarding the rest.

Production Methodology

A commercial book is created by a team of people. When my colleagues and I produce a book with ZzTEX we usually break the work down as follows. I am responsible for converting the author's files to ZzTEX, because this often involves some programming with ZzTran. I am also responsible for creating the design file according to the book designer's specifications, and producing sample pages that illustrate the design. Once the design is accepted by the publisher and author, I deliver the design file and the ZzTEX source files to a person who will be responsible for the composition of the book. The compositor enters copyediting changes, produces a rough set of first pages (galleys), enters proofreading changes, produces an almost-complete set of the second pages, and produces the final pages for reproduction proof or film.

If you are writing and producing your own book, you may be responsible for all of the steps outlined above. In either case, care is required to keep the master files up-to-date, and to ensure that the notations on paper manuscripts marked by copyeditors and proofreaders are incorporated back into the files completely and accurately. The biggest dilemma concerns automatic indexing: how can the compositor work on the files while the indexer "owns" them to enter index entries? In a confined computer environment, some type of source control system can be used. But, in the real world of far-and-wide subcontractors, I have not devised an acceptable solution. We usually have the indexer put the entries in a text file, convert it to ZzTEX, and typeset it independently of the rest of the book.

Future Work

Although I have been developing ZzTEX for three years, there is still significant work to be done. My to-do list includes the following:

- The multicolumn support must be redesigned. At present, ZzTEX produces multicolumn pages by treating each column as a separate logical page. This allows single-column footnotes and floating figures, but makes it difficult to balance the columns on the final page, particularly when switching back to single column format on the same page. A hybrid scheme is required, where each column is a separate logical page, but, when necessary, all columns can be collected and treated as one page.
- While the current scheme for horizontal placement of elements is flexible, its use is not intuitive. It is particularly difficult to produce atypical kinds of centering, such as centering text in an area other than the normal text measure. I want to implement a new technique involving four parameters: `\hshift`, to control horizontal shift; `\measure`, to control the text measure; `\flush`, to control whether text is flush left, flush right, or centered; and `\width`, to specify the width on which the flushing is performed.
- I want to complete the work necessary to make ZzTEX independent of Plain TEX, yet allow them to coexist. You will be able to load ZzTEX into TEX with or without the Plain TEX macros.

- The ZzTEX manual is incomplete. It is currently about 220 pages, and will expand to approximately 400 pages.

No software is ever finished, and I will continue to enhance ZzTEX. Nonetheless, I know that Donald Knuth's TEX is capable of producing truly beautiful books.

◇ Paul C. Anagnostopoulos
Windfall Software
433 Rutland Street
Carlisle, MA 01741
greek@genome.wi.edu

The \noname macros — a technical report

Jonathan Fine

Abstract

The \noname package provides a powerful environment for writing TEX macros. Its use makes macros easier to read, easier to write, and easier to document. It allows ready access to powerful control macros. It allows diagnostic and other code to be tagged for conditional inclusion. The \noname package is fully compatible with existing macros.

Here are two major features. It allows easy access to arbitary character tokens. Lines that do not begin with a white space character are comments, and are ignored.

The intention has been to provide the productive features that users of other programming languages take for granted. This article provides an outline of the history, design and implementation of the \noname package.

Acknowledgements

I would like to thank Nelson Beebe, and particularly Michael Downes, for their careful comments on an earlier version of this article.

1 Introduction

The \noname package grew out of work the author was doing two years ago. The goal was to write macros, for setting verbatim code, that would set source in a \tt font, and comments in a proportional font. This effect was to be achieved without additional mark up of the input file. Other refinements over the usual verbatim listing for source code were also desired.

In the course of this programming, extensive access to characters with \catcodes other than those usually given was desired. This proved to be a stumbling block for this project, which still awaits completion. Various programming tricks were introduced. The result of systematically developing these devices is the \noname macros. Although they have now reached the stage of being useful, there are developments being considered that will further increase their power and usefulness.

The physical activity of erecting a building commences with the digging of a hole, that will become the foundations that support the planned structure. The larger the building, the deeper the hole. The \noname package is intended to provide secure foundations for large scale collections of TEX macros.

2 Examples

Here is a line of code from plain.tex. It supports the \newif construction. It creates a control sequence \if@ that must be followed by *other* characters 'i' and 'f' with catcode *other*, i.e. 12. Such funny letters arise from use of \string.

```
{\uccode`1=`i \uccode`2=`f
  \uppercase{\gdef\if@12{}}}
```

(The purpose of \if@ is to extract the string foo from \iffoo, which is then used to construct \foofalse and \footrue. The macro \if@ is also intended to give an error if the argument to \newif is *not* of the form \if...).

Here is the same macro defined using \noname.

```
\def \if@ 'i'f {}
```

The right quote symbol '`' is an *escape character* that serves to produce a character with catcode *other*, whose character code is given by the following alphabetic constant.

Here is another example. The \noname macro definition

```
\def\spaces{ ~~~~~ }
```

defines \spaces to be a macro whose replacement text is five ordinary space tokens. (Ordinarily, special tricks are required to get a space after a control word or another space). Finally,

```
\def !\^^M { \par }
```

will in \noname define active carriage return to expand to \par.