Text merges in TEX and LATEX*

Mike Piff

January 3, 1995

Abstract

In this article the author explains how to do some standard and not so standard word processor text merges in TEX documents, using no other tools than TEX itself. A common application is to the mail merge or form letter, where names and addresses are stored in a file, together with other bits of information, and a standard letter with variable fields embedded in it is customized for every name from this file. Another application is to the pretty-printing of the contents of a database.

The macros described in `textmerg.sty` work equally in both plain TEX and LATEX. However, this has meant heavy use of `\def` where `\newcommand` would have been preferable.

Contents

1 Introduction

It is often said that although LATEX is good at typesetting mathematics, it is wholly unsuitable for common word processor functions such as mail merges. The latter are easy to achieve in most ordinary word processors, but in its raw state LATEX is incapable of doing a mail merge, or, indeed, of generating the same block of text over and over again but with different parameters in each block, those parameters

*This article originally appeared in TUGboat Vol 13(4), 1992, p518–523.

having been read from a subsidiary merge file. The latter file might possibly be the output from a database or any other program.

This article aims to show the reader that such a repetitive task need not be as difficult as it at first appears. In TEX, it is possible to hide many details of a facility inside a subsidiary style file, so that the user is unaware of what fearful processes are going on in the background. It is then possible to present the end-user with an extremely simple interface, perhaps simpler and more powerful than is available in other systems.

In earlier TUGboat articles [Bel87, Gar87, Lee86, McK87] it was shown how a standard letter could be customized by adding names and addresses from a separate file. I aim to show that it is possible to achieve far more than this with a fairly compact but general set of macros.

2 A simple example

Suppose that we have a list of student names and examination grades, one per student, and that we wish to send a letter to each student giving his/her exam grade. We must decide first what bits of information must be prepared in our subsidiary file, by looking at an example letter and finding out which items change from letter to letter.

Suppose that one instance of our letter is the following, a LATEX example.

```
\begin{letter}{Miss Iusta Mo\\
   34 Winchester Road\\
   Sheffield\\
   England}
   \opening{Dear Miss Mo,}
   This letter is to inform you
   that you obtained grade A in
   your recent examinations.
   \closing{Yours faithfully,}
\end{letter}
```

We can see that we need to know the student's title, forename(s), surname, address and grade to compose such a letter.

One of the simplest ways of achieving this effect is to prepare a file with lines of the form

```
\MyLetter{Mr}...{C}
```

for each student and then simply `\input` it into a LATEX file in which `\MyLetter` has been defined as having five parameters. A problem with this approach is that we may not be able to coax the student database into producing such a file. Another problem is that we need something more subtle if there are fifty parameters. For example, we might want to print out the contents of the student database with one page per student, but it could be that there are fifty information fields per student. Even worse, the number of pieces of information per student might not be a constant number, because, say, we are printing out fields from a related file in which marks on individual examination papers are held.

We shall tackle our simple example in a way that lends itself to more generality later on, and in a form that most database programs should be capable of handling.

We thus prepare a subsidiary file `results.dat` with records of five fields in it. Each student is represented by five lines of this file,

```
⟨*results⟩
Miss
Iusta
Mo
34 Winchester Road\\Sheffield\\England
A
Mr
Arthur
Minit
43 Sheffield Road\\Winchester\\England
C
⟨/results⟩
```

and the student records appear one after another in this file. Thus both the field and record separators are carriage returns.

`\Merge` `\Fields`

TeX itself needs to know three bits of information:

1. the name of the subsidiary file,
2. the fields to read, and
3. the template of the letter.

We pass it this information in the following form

```
⟨*examp⟩
\documentclass[12pt]{letter}
\usepackage{textmerg}
\begin{document}
\Fields{\Title\Forenames\Surname
   \Address\Grade}
\Merge{results.dat}{%
\begin{letter}{\Title\ \Forenames\
   \Surname\\\Address}
   \opening{Dear \Title\ \Surname,}
   This letter is to inform you
   that you obtained grade \Grade\ in
   your recent examinations.
   \closing{Yours faithfully,}
\end{letter}}
\end{document}
⟨/examp⟩
```

LaTeX should open the subsidiary file and, for each set of five parameters, generate a letter in the `dvi` file. When it reaches the end of the merge file, LaTeX should terminate execution of the `\Merge` command and presumably finish the document.

3 A few complications

Looking at the above example in a bit more generality, we see that we are reading records of n fields from the merge file and placing them into a TeX document in such a way that they replace n preassigned control sequences. However, it may happen that the merge file is prepared by humans, who might possibly have inserted some extra blank lines into the file. Again, it could be that certain sorts of fields might be blank, whereas others can never be blank. Perhaps it would be better to build in some degree of error recovery.

\Fields We shall make the assumption that the first field in any record is definitely a non-blank one and that we know beforehand whether each of the others might conceivably be blank. We make a modification to our `\Fields` statement. It can contain not only the field name control sequences but also the tokens `+` and `-`, with the following interpretation. A `+` indicates that all following fields should be re-read until a non-blank result is obtained. A `-` indicates that any following fields could conceivably be blank, subject to the restriction that the very first field is always non-blank.

Thus the command

```
\Fields{\a+\b\c-\d}
```

would indicate that only `\d` is allowed to be blank, because the `+` token has no effect. In

```
\Fields{-\a\b+-\c+\d}
```

the initial `-` token enables blank reading of data tokens, but the very first data token is not permitted to be blank anyway. Thus `\a` is read as a non-blank token and `\b` as a possibly blank token. The sequence `+-` now switches non-blank reading on and off again, so `\c` is read as possibly blank. Finally `\d` is non-blank.

Another complication we allow is that the `\Fields` command can appear several times in our file. The interpretation is that the last occurrence of `\Fields` before we encounter the `\Merge` command will indicate the fields to be read for every record. Any occurrences of `\Fields` within the merged text indicate a new list of fields to be read when that command is encountered. This lets us do some conditional processing, such as[1]

```
   \ifx\Title\Mrs
      \Fields{\MaidenName}
   \fi
```

and also gives us some flexibility about the field order later on.

It should also be stressed that the undefined control sequences appearing in the template need not correspond exactly to the fields in the subsidiary file. An example might be that the subsidiary file contains the text

```
Spriggs, Mr Abraham L
```

and one field read is `\FullName`. TeX would then have to pre-process this name to generate its several components as used in the template. The command `\PreProcess` could be included at the start of the template.

```
\def\parse#1, #2 #3\endparse{%
   \def\Surname{#1}\def\Title{#2}%
   \def\Forenames{#3}}
\def\PreProcess{\expandafter
   \parse\FullName\endparse}
```

[1]It is assumed that `\Mrs` expands to `Mrs`.

An alternative and simpler looking approach to reading fields from a file `\fil` might be to define each such field as follows.

```
\def\Field#1{\def#1{\read\fil to#1#1}}
\Field\Name \Field\Address \Field\Mark
```

The first time `\Name` is encountered, it reads its own expansion from `\fil` and then expands itself. Henceforth, it has acquired its new expansion. The disadvantage is that `\Name` must appear in the text before any subsidiary field such as `\Surname` can be used.

Finally we should consider the possibility that the second parameter of `\Merge` might be too large to fit into memory. We can clearly handle this problem by allowing the second parameter merely to consist of the text `\input template`, so that the root file handles two subsidiary files, one containing the template and the other containing the fields.

4 A complicated example

We will next look at an example in which the template contains a table of indeterminate length, albeit fixed width. So far our macros work in either plain TeX or in LaTeX, but the way in which these two packages handle tables is slightly different. However, the only difference that need concern us is that LaTeX uses `\\` where plain TeX uses `\cr`.

`\MultiRead` The example given here is in LaTeX, but our style will work equally well in plain TeX. In our student letter we wish to insert a table of course codes and marks. Since each student did a different number of courses, we need some way of recognizing the end of the course list in the merge file. The default will be to insert a blank line at the end of such a sub-list. Thus, the following text appears before the close of the letter template.

```
Here are your marks on individual papers.
\begin{center}
   \begin{tabular}{|lr|}\hline
      Code&Mark\\\hline
      \MultiRead{2}\\\hline
   \end{tabular}
\end{center}
```

The merge file now has the following structure.

```
Title
...
Grade
Code
Mark
...
Code
Mark
```

⟨*blank*⟩

```
Title
...
```

\MarkEnd In other applications some of the fields in the table might possibly be blank. We then let the user change the ⟨*blank*⟩ line marking the end of a list to some other string of his own choosing.

```
\MarkEnd{***}
```

There might be multiple tables in the same template, with their data intermingled in the merge file with main fields. The generalized `\Fields` command allows us to order the merge file however we want. Thus we could have main fields, then a table, followed by more main fields, and so on.

\Process A final complication is that the fields appearing in a table are essentially anonymous. By this I mean that they are transferred into the table as they are, without any pre-processing possible through appearing in the template as control sequences. If we wish what appears in the table to be different from what appears in the file, a mechanism is needed to tell TEX that a certain column has to be treated in a certain way. The command

```
\Process{n}{\foo}
```

will replace every field ⟨*f*⟩ read into column n by `\foo{`⟨*f*⟩`}`. It is even possible to do some numerical calculations by this method.

Here is a LATEX example to illustrate the table processing features of `textmerg.sty`.

```
⟨*example⟩
\documentclass[12pt]{article}
\usepackage{textmerg}
\MarkEnd{***}
\Process{2}{\Advance}
\def\Advance#1{#1\addtocounter{page}{#1}}
\Fields{+\Name\Verb}
\begin{document}
\Merge{silly.dat}{%
   Dear \Name,\par
   Here is a table to \Verb\ at:
   \Fields{\Width}%
   \begin{tabular}{*{\Width}c}
      \MultiRead\Width
   \end{tabular}.\par
   \Fields{\Adj}%
   That was \Adj!
   \clearpage}
\end{document}
⟨/example⟩
```

The effect of this file is not apparent until we see `silly.dat`. It is listed here.

```
⟨*silly⟩
Mike
look
3
1
2
3
11
12
```

```
13
***
good
Shelagh
gaze
2
21
22
23
24
***
horrid
⟨/silly⟩
```

The same can be done in plain TEX.

```
⟨*plainexample⟩
\input textmerg
\MarkEnd{***}
\Process{2}{\Advance}
\def\Advance#1{#1\global\advance\count0by#1}
\Fields{+\Name\Verb}
\Merge{silly.dat}{%
   Dear \Name,\par
   Here is a table to \Verb\ at:
   \Fields{\Width}%
   \vbox{\halign{\hfil{} ## {}\hfil&&\hfil{} ## {}\hfil\cr
      \MultiRead\Width\cr
   }}.\par
   \Fields{\Adj}%
   That was \Adj!
   \vfill\eject}
\end

⟨/plainexample⟩
```

5 Identification

This package can only be used with LATEX 2_ε, so an appropriate message is displayed when another format is used[2].

```
⟨*textmerg⟩
\NeedsTeXFormat{LaTeX2e}[1994/01/01]
```

Announce the package name and its version:

```
\ProvidesPackage{textmerg}[\filedate]
```

And display it on the terminal (and the log file):

```
\typeout{Package ‘textmerg’ <\filedate>.}
\typeout{\Copyright}
⟨/textmerg⟩
```

The plain TEX version will simply `\input` this package file. Thus we need to know that it will understand everything in the file.

```
⟨*plain⟩
```

[2]However, some code is inserted to allow its use with plain TEX

```
\def\NeedsTeXFormat#1[#2]{}
\def\ProvidesPackage#1[#2]{}
\def\typeout#1{\immediate\write0{#1}}
\input textmerg.sty
⟨/plain⟩
```

6 Implementation of the simple case

`\glet` For convenience we define a frequently used combination here.

```
⟨*textmerg⟩
\def\glet{\global\let}
```

`\MergeFile` `\InputFile` The subsidiary merge file is defined next. A macro is then defined that attempts to open it for reading. If that is unsuccessful, the file is closed and an error message is issued.

```
\newread\MergeFile
\def\InputFile#1{%
   \openin\MergeFile=#1
   \ifeof\MergeFile
   \errmessage{Empty merge file}%
   \closein\MergeFile
   \long\def\MakeTemplate##1{%
      \def\Template{}}%
   \else\GetInput\fi}
```

The command `\MakeTemplate` will be used later to generate the body of the form into which fields are inserted. We redefine it if the file is empty so that it produces no text.

`\GetInput` Because the conditional `\ifeof` does not return true until after an unsuccessful read operation, a mechanism of looking ahead is used which is similar to that found in Pascal.

```
\def\GetInput{{\endlinechar=-1
   \global\read\MergeFile to\InputBuffer}}
```

`\SeeIfEof` `\LookAgain` We set up a mechanism for deciding whether or not we have exhausted the merge file. It forces `\ifeof` to return true by skipping over blank lines.

```
\def\SeeIfEof{%
   \let\NextLook\relax
   \ifeof\MergeFile
   \else
      \ifx\InputBuffer\empty
         \LookAgain
      \fi
   \fi
   \NextLook}
\def\LookAgain{\GetInput
   \let\NextLook\SeeIfEof}
```

`\ifNonBlank` `\AllowBlank` `\DontAllowBlank` We can now prepare to read actual fields from the merge file. A conditional is used to indicate whether or not the field we are about to read is allowed to be blank. We also set up a mechanism for changing its value.

```
\newif\ifNonBlank \NonBlankfalse
\def\AllowBlank{\global\NonBlankfalse}
\def\DontAllowBlank{\global\NonBlanktrue}
```

\ReadIn
\MissingField

Fields are actually read by means of the following command. Its only parameter is the name of the control sequence into which the field is read.

```
\def\ReadIn#1{%
   \ifNonBlank\SeeIfEof\fi
   \ifeof\MergeFile
      \gdef#1{??}\MissingField
   \else
      \glet#1\InputBuffer
      \GetInput
   \fi}
\def\MissingField{%
   \message{Missing field in file}}
```

\GlobalFields
\Fields

The `\Fields` command places its parameter into a token register called `\GlobalFields`. This command will be redefined by the `\Merge` command.

```
\newtoks\GlobalFields
\def\Fields#1{\GlobalFields{#1}}
```

\ParseFields
\EndParseFields

When a field token list is read, each individual token within it must be either read as a field or interpreted as a blank/nonblank switch. The next token is then read by tail recursion. It is assumed that the final token in the list is `\EndParseFields`. This must be defined to expand to something unlikely to be read as a value of one of the fields, and so we `\let` it to `\ParseFields`.

```
\def\ParseFields#1{%
   \ifx#1\EndParseFields
      \let\NextParse\relax
   \else
      \let\NextParse\ParseFields
      \ifx#1+\DontAllowBlank
      \else
         \ifx#1-\AllowBlank
         \else\ReadIn#1
         \fi
      \fi
   \fi\NextParse}
\let\EndParseFields\ParseFields
```

\ReadFields

We apply this command to our token register after expanding it.

```
\def\ReadFields#1{\expandafter\ParseFields
   \the#1\EndParseFields
   \AllowBlank}
```

\Merge
\MakeTemplate

At long last we are ready to define the `\Merge` command itself. The first parameter is the filename of the subsidiary file and the second is the template or form into which fields are inserted. Since a `\Fields` command within the `\Merge` text is meant to act immediately on the token list that follows it, we redefine it to operate in a different way.

```
\long\def\Merge#1#2{\begingroup%
   \InputFile{#1}%
```

```
   \def\Fields##1{%
      \ParseFields##1\EndParseFields}%
   \MakeTemplate{#2}\Iterate}
\long\def\MakeTemplate#1{\def\Template{#1}}
```

The grouping keeps any changes to the definition of `\MakeTemplate` local to this merge. Thus several consecutive merges can be handled within one document. The `\endgroup` is supplied by the macro `\Iterate` when the merge file has been exhausted.

`\Iteratecounter`
`\Iterate`

`\Iterate` must read the fields which were declared before it was entered, substitute them into its template and repeat itself using tail recursion if the end of the merge file has not been encountered.

```
\countdef\Iteratecounter=2
\Iteratecounter=0
\def\Iterate{%
   \global\advance\Iteratecounter by1
   \ReadFields\GlobalFields
   \Template
   \SeeIfEof
   \ifeof\MergeFile
      \def\NextIteration{%
         \endgroup\closein\MergeFile}%
   \else
      \let\NextIteration\Iterate
   \fi
   \NextIteration}
```

The point of the use of counter 2 in the above is that it is accessible to the print driver for page selection. Anyone who has started printing 150 letters, all with page number 1, only to run out of paper half way, will appreciate the use of this artifice!

7 Implementation of merged tables

`\MultiCount`
`\MaxCount`
`\ifStartOfList`

We set up two counters, one for the column we are reading and the other for the total number of columns in the table. We also need a conditional to mark the start of the table, so that we terminate each row correctly with `\\` or `\cr`, or nothing at all at the beginning of the first row.

```
\newcount\MultiCount \newcount\MaxCount
\newif\ifStartOfList
```

`\MultiRead`

The parameter to `\MultiRead` is the number of columns to read at a time. This command passes control to `\NextRead` after initializing certain parameters.

```
\def\MultiRead#1{%
   \ifnum#1>0
      \SelectCR
      \MakeEmpty{#1}%
      \global\StartOfListtrue
      \glet\NextRead\MRead
      \AllowBlank
      \global\MaxCount=#1
```

```
        \NextRead
    \fi}
```

\Emptyctr \MakeEmpty The command \MakeEmpty is required by the pre-processing of each field. The idea is that the command \csname pr*nn*\endcsname, which we will loosely call \pr*nn*, is executed on each field in column *nn*. However, most of these commands will be undefined, and so we equate each of those that has not been defined to \empty.

```
\newcount\Emptyctr
\def\MakeEmpty#1{\Emptyctr=0
    \loop
        \advance\Emptyctr by1
        \expandafter\ifx\csname
            pr\the\Emptyctr\endcsname\relax
            \expandafter\glet\csname
                pr\the\Emptyctr\endcsname\empty
        \fi
        \ifnum\Emptyctr<#1
    \repeat}
```

Note that, because of the way we are accessing it via \csname, the first time \pr*nn* is encountered it equates to \relax.

\Process The command \Process#1#2 defines \pr#1 to mean #2.

```
\def\Process#1#2{%
    \expandafter\def\csname
        pr#1\endcsname##1{#2{##1}}}
```

\MarkEnd We need to know how the last row is to be recognized. The default is an empty line in the merge file.

```
\def\MarkEnd#1{\gdef\EndMarker{#1}}
\MarkEnd{}
```

\NextLine \NextField We collect each row in a token register. The full row is assembled in \NextLine before being passed back to TeX. Each field is read in \TempField and then placed temporarily into \NextField.

```
\newtoks\NextLine \newtoks\NextField
```

It is not necessary to do things this way; \edef can be used instead, but that approach might expand tokens prematurely.

\AppendNextField After the next field has been read, it is appended to \NextLine.

```
\def\AppendNextField{%
    \global\advance\MultiCount1
    \NextField=\expandafter{\TempField}%
    \edef\Append{\NextLine=
        {\the\NextLine&\csname
            pr\the\MultiCount\endcsname
            {\the\NextField}}}%
    \Append}
```

\EndLine \FinishLine We need to insert the correct end marker after each row of the table. The token \cr must be disguised a little before it is acceptable in a LaTeX document.

```
\def\SelectCR{\glet\EndLine\\}%
⟨/textmerg⟩
⟨plain⟩\def\SelectCR{\gdef\EndLine{\cr}}%
⟨*textmerg⟩
\def\FinishLine{%
   \ifStartOfList
      \global\StartOfListfalse
   \else\EndLine\fi}
```

This makes the assumption that if \array is defined then we must be in LaTeX.

\StopProcessing We need a command to finish off a table. This should reset \NextRead to \AllowBlank to terminate the tail recursion, and also do some error recovery in case the file ends prematurely in the middle of a row.

```
\def\StopProcessing{%
   \global\MultiCount\MaxCount
   \glet\NextRead\AllowBlank}
```

\MRead The command \MRead prepares to read a row of a table. It reads a field from the merge file and checks to see whether the table has been exhausted.

```
\def\MRead{%
   \global\MultiCount=1
   \ReadIn\TempField
   \ifx\TempField\EndMarker
      \StopProcessing
   \else
      \FinishLine
      \NextField=\expandafter{\TempField}%
      \edef\StartLine{\NextLine={\csname
         pr1\endcsname{\the\NextField}}}%
      \StartLine
      \ConstructNextRow
   \fi
   \NextRead}
```

\ConstructNextRow Command \ConstructNextRow does most of the work of assembling a row of the table. It assembles \MaxCount fields at a time into \NextLine unless an error is encountered.

```
\def\ConstructNextRow{%
   \ifnum\MultiCount<\MaxCount
      \loop
         \ReadIn\TempField
         \ifx\TempField\EndMarker
            \glet\TempField\empty
            \StopProcessing
            \MissingField
         \else
            \ifeof\MergeFile
               \glet\TempField\empty
               \StopProcessing
               \MissingField
```

```
            \fi
          \fi
          \AppendNextField
          \ifnum\MultiCount<\MaxCount
      \repeat
    \fi
  \the\NextLine}
⟨/textmerg⟩
```

8 The documentation driver file

This is the driver file that produces this documentation. We use the document class provided by the LaTeX 2ε distribution for producing the documentation.

```
⟨*driver⟩
\documentclass{ltxdoc}
\RecordChanges
\begin{document}
  \DocInput{textmerg.dtx}
  \PrintIndex
  \PrintChanges
\end{document}
⟨/driver⟩
```

References

[Bel87] Edwin V. Bell, II. AutoLetter: A TeX form letter procedure. *TUGBoat*, 8(1):54, April 1987.

[Gar87] John S. Garavelli. Form letter macros. *TUGBoat*, 8(1):53, April 1987.

[Lee86] John Lee. Form letters. *TUGBoat*, 7(3):187, October 1986.

[McK87] Graeme McKinstry. Form letters. *TUGBoat*, 8(1):60, April 1987.

Change History

2.01
  "General": First version for LaTeX2e 1