csvmerge

Laurence R Taylor

August 8, 2019

Abstract

The package `csvmerge.sty` is designed to take a bit of TEX code such as a letter or a line in a table and produce multiple copies of it with parts of each copy filled in from a data file. It is invoked in the usual manner from a master-file, that is a standard LATEX file with

```
\documentclass{whatever}
\usepackage{csvmerge}
          ⋮
```

The data source is a standard csv file. The bit of TEX code resides in a blueprint file. The `\usepackage{csvmerge}` can go anywhere in the preamble.

Contents

1 Introduction

The `csvmerge.sty` merges TEX code with a data file.

- The TEX code comes from a blueprint file, which looks like a standard TEX file except it contains no code which should go into the preamble of a normal TEX file. The blueprint file should also not contain a `\begin{document}` nor an `\end{document}`. This stuff goes in the master-file.
- The data file is a standard csv file.

One difference between `csvmerge.sty` and other packages which merge `csv` data into a TEX file is that the user does not need to do much with the `csv`-file. As long as the first row contains reasonable entries describing the contents of its column you are good to go. Extra columns are no problem; subsequent rows can be too short or too long; characters which are special to TEX such as `$` and `&` can appear in the data; ... More details concerning the `csv`-file can be found in subsection 2.1.

The `tex`-file consists of standard TEX code. Each row of the `csv`-file is processed and you have access to the values of all the entries on that row as well as the row number. You also can tell if the entry in a particular column is empty. More details concerning the `tex`-file can be found in subsection 2.2.

The main macro, `\mergeFields`, supports two modes. In one mode, *stream mode*, the TEX code from the `tex`-file is added to the input stream at the location from which `\mergeFields` is invoked. In the second mode, *storage mode*, the code is written to a temporary file. You can then use the standard LATEX `\input` to read the temporary file back into the master-file.

Good models for the two modes are the following. Most mail-merges can be done in streaming mode. Because TEX is doing rather complicated things behind the scenes when it is setting a table or a matrix, you are usually better off using storage mode with your blueprint file containing the code for one line of your table/matrix. After the data is generated you can read it back into your table. See here for a typical example.

2 Usage

`\mergeFields` The main macro is `\mergeFields` which takes one argument. The format is

```
\mergeFields{
   tex={<tex-file>}
   csv = {<csv-file>}
   tmp = {<tmp-file>}
  }
```

The `tmp` entry is not required. If there is none, `mergeFields` operates in *streaming* mode, otherwise it acts in *storage* mode. The `<prefix-file>` entry should be a path to the corresponding sort of file. If the file name has spaces you will need to quote it: life is easier if there are no spaces in file names. The format `prefix = {<prefix-file>}` is quite forgiving: all leading and trailing spaces are stripped out so `prefix   =   { <prefix-file>    }` works. The three lines can be in any order and do not even need to be on separate lines. The extensions of the three files are irrelevant: the `tex`-file contains TeX code but it can have an extension of `.txt` (or anything else or none at all) if you prefer. All that matters is that when TeX reads the file the contents have the proper content. If you have a simple project you may be able to complete it just by looking at the examples in appendix B.

2.1 The data file

The data file is a `csv` spreadsheet file. By default the field delimiter is the comma and the text delimiter is the double-quotes character. This is default behavior for most spreadsheets when asked to produce a `cvs` file. If there are no `"`'s in your `csv`-file there should be no problems. In some cases you may need a different set of delimiters and this is supported. See `\setDelimitersTabQuote` below for more details.

The entries in the first row should consist solely of *alphanumeric characters, punctuation and spaces*. Each entry is a *label* for that column. Labels need not be unique although they usually are except sometimes there are several columns with blank labels.

A *field-name* is extracted from a label by deleting all spaces and making all characters lowercase: e.g. a column labeled `First Name` has a field-name `firstname`. Commas are also deleted so a column label of `Last, First Name` becomes a field-name of `lastfirstname`. Distinct labels can have the same field-name although again this seems unlikely except for blank labels.

Associated to each field-name is a *field-name-macro*. This macro is private: its name is found by prepending `@LRT@@` to the field-name. When `\mergeFields` is invoked, the data file is read one row at a time. All rows after the first are data. When a row is read, each entry is peeled off and placed in the field-name-macro whose field-name is the one associated to the label for that column with `@LRT@` prepended. For example, if row one of some column contains `First Name`, the field-name is `firstname` and the field-name-macro is `\@LRT@@firstname`. When a row after the first is read, `\@LRT@@firstname` will expand to the entry in the `First Name` column on that row.

When `\mergeFields` has read a row and set the field-name-macros the `tex`-file is read and the TeX code evaluated. In streaming mode the result is added to the input stream at the current location whereas in storage mode the result is written to the `tmp`-file.

`End of csv-file` This process continues until the last line in the `csv`-file is processed OR until an entirely blank line is encountered.

The request for only alphanumeric, punctuation and blanks for entries in row one is not actually enforced but the macro `caselower` from `stringstrings` is used to produce lowercase letters and may do weird things to characters not on the requested list. Without studying the `stringstrings` manual carefully you may not be able to identify the field-names if you insist on weird characters.

It is quite possible that several columns can have the same field-name, and hence the same field-name-macro. In that case the macro will expand to the contents of the column with that field-name which is furthest to the right.

Because of the prepended `@LRT@@` you can not use these field-name-macros directly in your `tex`-file. The solution is to use `\Field{field-name}` which will expand correctly. See the examples in appendix B.

2.1.1 Entry issues

Except for issues discussed here, you should be able to use any standard `csv`-file.

The data file is read with the usual TeX issues turned off: `#`, `$`, `&`, `@`, `%` , `^`, `~`, and `|` can all be in your data file.

- In streaming mode, only `#`, `$`, `&`, `@`, `%`, `^`, `~` print correctly. The characters \$, #, %, @ and & often occur in data and they print correctly. TeX code will not be rendered correctly so for example, `\$` in the data file produces “`$` when TeXed. As long as you are not too creative in your entries you should get the results you want.

- In storage mode the characters above end up printing correctly in the `tmp`-file BUT when you `\input` the `tmp`-file back into your master-file these characters will resume their usual TeX meaning and wreak havoc with your code. If you absolutely must have them, writing `\&`, `\$`, `\#`, etc. in the `csv`-file may save you.

`Delimiters` There is one more issue. For most csv files, the field separator is the comma. When a field has a comma in it, the entire field is delimited by double quotes (`"`). When the field has a double quote it is converted to a pair of double quotes (`""`) and so on. If you have both commas and quotes in a field the best case scenario is that you will get `""`’s where you were hoping for `"`’s. This probably is not what you want. Even worse, a `",` in a data entry will end the entry prematurely at this point and things deteriorate from there.

There is a mechanism to deal with this. Most spreadsheets can save csv files with other field (and text) delimiters. The second most common version of csv has a tab as the field delimiter and quotes for fields which have a tab in a field.

These two cases are supported out of the box. By default you get the comma-quotes version. If you add `\setDelimitersTabQuote` before calling `\mergeFields` you will get the tab-quotes version. If you have a second call to `\mergeFields` and the data file is comma-quotes, adding `\setDelimitersCommaQuote` before `\mergeFields` will restore comma-quotes. See setDelimters... for a more thorough discussion.

2.2 The blueprint file

The blueprint file is a normal TeX file with macros defined as above from the `csv`-file. It should not contain a `\begin{document}` or an `\end{document}` nor anything that needs to be expanded in the preamble. Such things belong in the master file.

For mail-merges, the `tex`-file should end with a `\newpage` to guarantee that each copy of the message appears on a different page. If you need custom sizes or headers or you have page numbers that you want to reset for each copy, you should do this in the `tex`-file.

To use the data from the `csv`-file you need to write `\Field{some-field-name}` wherever you want the data in your `tex`-file. Some field-names may be rather long so if you are going to use them a lot in your `tex`-file, feel free to write something like `\def\sfn{\Field{some-field-name}}` at the top of your `tex`-file and then use `\sfn` thereafter.

One additional issue occurs in storage mode. You may very well want TeX code in your `tex`-file. When `mergeFields` reads a line of data, you want the field-name-macros to expand to their current values but you often do not want your TeX code to expand. It is good practice stick a `\noexpand` in front of each TeX macro. Each line is expanded once when the line is written to the `tmp` file, so the `\noexpand` disappears and you are left with just your TeX macros.

If you do define macros like `\sfn` as above, you will need to write
`\noexpand\def\noexpand\sfn{\Field{some-field-name}}`.

3 List of public macros

There are the 6 public macros: links are to descriptions of the `@LRT@`-versions which actually do the work.

`\mergeFields`
`\setDelimitersCommaQuote`
`\Field`
`\setDelimitersTabQuote`
`\ifFieldEmpty`
`\makeMePublic`

- `\mergeFields{tex={tex-file}csv={csv-file}}` or
  `\mergeFields{tex={tex-file}csv={csv-file}{tmp={tmp-file}}`
  sets up to do the merge. Section 2 is devoted to a detailed discussion.

- `\Field{field-name}` expands to the current value of the field-name-macro.
- `\ifFieldEmpty{field-name to test}{do if empty}{do if not}`
- The two `\setDelimiters` set the delimiters to match those used in the `csv` file: comma-quotes is the default.
- `\makeMePublic` must be immediately followed by `\@LRT@` or you will get a `Use of \makeMePublic doesn't match its definition.` error. Then you can write `\makeMePublic \@LRT@foo\becomes\Foo` to make the public macro `\Foo` that you just define equal to the private macro `\@LRT@foo`. You may not write `\makeMePublic{\@LRT@foo}\becomes\Foo` or other variants: the first non-white-space character TEX sees after `\makeMePublic` needs to be the start of `\@LRT@`.

  As a useful example, `\makeMePublic\@LRT@Row\becomes\Row` means you can get your hands on the row number using `\Row`.

4 Implementation

`Header` First we have the standard package header stuff. The line numbers in the typeset csvmerge.dtx file match the line numbers in the csvmerge.sty file generated by TEXing the csvmerge.ins file.

```
16 \NeedsTeXFormat{LaTeX2e}[2005/12/01]
17 \ProvidesPackage{csvmerge}
18       [2019/07/17 v1.0 merges TeX code with csv data]
19
```

`Required Packages` These required packages are part of the standard `TeX Live` and `MiKTEX` distributions.

```
20 \RequirePackage{stringstrings}
21 \RequirePackage{trimspaces}
22 \RequirePackage{etoolbox}
23
```

`Options` `csvmerge` supports an option which is rarely needed. All the private macros (except one) in this package have `@LRT@` or `@lrt@` somewhere in their name so are very unlikely to produce name conflicts. In the very unlikely chance that one or more of the six public macros has a name conflict, this can be alleviated by putting `\usepackage[foo]{csvmerge}` in your master file. Here `foo` can be any string of alphabetical characters. Then the names of all six public macros are invoked with a leading `foo`: `\mergeFields` no longer exists but becomes `\foomergeFields`. The evident change happens with the other five macros.

The next two lines of code merely save the option for use later at the end.

```
24 \DeclareOption*{\xdef\@LRT@CO{\CurrentOption}}
25 \ProcessOptions\relax
26
```

```
\makeatletter

```

`\mergeFields`

`\@LRT@mergeFields` actually initiates the work. It initializes some macros for later use. Then it collects the `prefix={<prefix-file>}` arguments using `\@LRT@getFilePaths` and checks to be sure it has a `csv`-file and a `tex`-file. If not it bails. Then it checks if it has a `tmp` file and sets the file reading subroutine, `\@LRT@Input` to `\@LRT@streaming` is there is no `tmp` file and to `\@LRT@storage` if there is. If there is a `tmp` file, it is opened. Then `\@LRT@mergeData` is called which actually does the merging. Finally, if there is a `tmp` file, it is closed.

```
\newcommand{\@LRT@mergeFields}[1]{%
\gdef\@LRT@csv{}%
\gdef\@LRT@tex{}%
\gdef\@LRT@tmp{}%
\setcounter{\@LRT@macroPrefix Row}{2}\relax%
\gdef\@LRT@csvBlankLine{}%
\@LRT@getFilePaths#1=@%
\if@LRT@empty{\@LRT@csv}{\@LRT@missingArgumentMessage{csv}}{}%
\if@LRT@empty{\@LRT@tex}{\@LRT@missingArgumentMessage{tex}}{}%
\if@LRT@empty{\@LRT@tmp}%
{\global\let\@LRT@Input\@LRT@streaming}%
{\global\let\@LRT@Input\@LRT@storage}%
\if@LRT@empty{\@LRT@tmp}{}{%
\immediate\openout\@lrt@tmpFile=\@LRT@tmp\relax}%
\@LRT@mergeData%
\if@LRT@empty{\@LRT@tmp}{}{\immediate\closeout\@lrt@tmpFile\relax}%
}

```

`\@LRT@macroPrefix`

```
\gdef\@LRT@macroPrefix{@LRT@@}

```

`\Field`

`\@LRT@Field` is just a wrapper for a `\csname ... \endcsname`. Notice it adds the `\@LRT@macroPrefix` prefix to the argument so the input is always the field-name and the output is always the value of the corresponding field-name-macro. Because of the third `@` there is no danger that these macros defined from entries in the `csv`-file will conflict with other macros defined in this package since none of them have a `@@` in their names.

```
\def\@LRT@Field#1{%
\csname\@LRT@macroPrefix#1\endcsname%
}

```

`\ifFieldEmpty`

`\ifFieldEmpty` takes three arguments. The first argument is a field-name and if the content of the corresponding macro is empty then the second argument is executed: otherwise the third argument is executed.

```
\long\gdef\@LRT@ifFieldEmpty#1#2#3{\ifbool{\@LRT@macroPrefix#1}{#2}{#3}}

```

`\setDelimitersCommaQuote`

`\setDelimitersTabQuote`

These two `setDelimiters` are similar. Recall how delimiters work. A row in a `csv`-file with comma delimiters looks like `foo1,foo2,foo3,...` so the delimiter is the comma and if `\@LRT@getFieldDelimited` is put in front of this string it will peal off `foo1`. Since `^^I` is TEX for the tab, a row in a csv file with tab delimiters looks like `foo1^^Ifoo2^^Ifoo3^^I...` and so the first two lines of `\@LRT@setDelimitersCommaQuote` and `\@LRT@setDelimitersTabQuote` look the same after switching `,` and `^^I`.

There is a wrinkle in the tab case. The tab normally looks to TEX like white space and so by the time the line reaches `\@LRT@getFieldDelimited` the line looks like `foo1foo2foo3...` and there is no way to break off `foo1`. The `\catcode‘\^^I12\relax` insures `\@LRT@getFieldDelimited` sees the line as `foo1^^Ifoo2^^Ifoo3^^I...`. This means that the catcode of `^^I` must also be 12 when each line is read and the line and the macro `\gdef\@LRT@catcodes{ \catcode‘\^^I12\relax}` insures this. In the comma delimited version no catcode heroics are needed so `\@LRT@catcodes` is empty.

If some field contains one or more `\@LRT@fieldDelimiterBack`'s then the `csv`-file will use the text-delimiter at the front and back. When quote is the text delimiter and `foo2` contains a comma but `foo1` and `foo3` do not, the comma delimited string looks like `foo1,"foo2",foo3,...` and the tab delimited string looks like `foo1^^I"foo2"^^Ifoo3^^I...`.

The text delimiters have a front one, the double-quotes in both versions, and the back one which is the front text delimiter followed by the back field delimiter.

The macro `\@LRT@blankFieldItem` is used to construct what a blank line looks like. In the comma delimited case, you need a string of commas of the same length as the number of entries on line one of the `csv`-file. In the tab delimited case, the blank line turns out to be a blank macro.

The `\@LRT@catcodes` macro is a collection of catcodes that need to be changed: nothing for the comma delimited case and `^^I` for the tab delimited case.

See Appendix A for more discussion on writing your own delimiters.

```
{{%
\gdef\@LRT@setDelimitersCommaQuote{
\gdef\@LRT@fieldDelimiterBack{,}
\gdef\@LRT@getFieldDelimited##1,{\@LRT@getFirstArg{##1}}
\gdef\@LRT@textDelimiterFront{"}
\gdef\@LRT@getTextDelimited"##1",{\@LRT@getFirstArg{##1}}
\gdef\@LRT@blankFieldItem{,}
\gdef\@LRT@catcodes{}
}%
}}

{{\catcode‘\^^I12\relax%
\gdef\@LRT@setDelimitersTabQuote{
\gdef\@LRT@fieldDelimiterBack{^^I}
\gdef\@LRT@getFieldDelimited##1^^I{\@LRT@getFirstArg{##1}}
\gdef\@LRT@textDelimiterFront{"}
\gdef\@LRT@getTextDelimited"##1"^^I{\@LRT@getFirstArg{##1}}
\gdef\@LRT@blankFieldItem{}
\gdef\@LRT@catcodes{\catcode`\^^I12\relax}
}%
}}

```

`\@LRT@makeMePublic` The next line of code defines the only macro which does not have `@LRT@` in its name. It seems sufficiently weird that there should be no name conflicts. Notice that `\makeatletter` is still in effect so `@` has catcode 11.

Then we switch to `\makeatother` so `@` has catcode 12. The next line expands to `\@LRT@makeMePublic` with the initial `\@LRT@` having its `@`'s of catcode 11 so it is a private macro. In the `\@LRT@` that you see, the `@`'s have catcode 12 so it is really the macro `\@` with extra stuff. Since it should never be expanded it doesn't really matter. Then we drop back into `\makeatletter` to define more private macros.

The net result is that you can write

`\makeMePublic \@LRT@foo\becomes\newMacro`

in your master-file and the private macro `\@LRT@foo` becomes the same as `\newMacro` and you can use `\newMacro` in your `tex`-file.

```
\expandafter\gdef\csname1238LRTLRTsvbneLRT\endcsname{@LRT@}
\makeatother
\expandafter\gdef\csname\csname1238LRTLRTsvbneLRT\endcsname
makeMePublic\endcsname\@LRT@#1\becomes#2{\global\letcs{#2}{@LRT@#1}}%
\makeatletter

```

`private macros`

`\@LRT@Row` The macro `\@LRT@Row` expands to the number of the row of the `csv`-file currently being read. To get the `\xdef` to expand correctly we needed to define (and did) `\@LRT@macroPrefix` first. Recall that the first line of actual data is row 2.

```
\xdef\@LRT@Row{\noexpand\number\noexpand\value{\@LRT@macroPrefix Row}}

```

`\if@LRT@empty` Takes three arguments: if `#1` expands to empty do the code in `#2`; otherwise do the code in `#3`.

```
\newcommand{\if@LRT@empty}[3]{\ifthenelse{\equal{#1}{}}{#2}{#3}}

```

`\@LRT@fieldName` Takes an alphanumeric string, uses `\caselower` from the `stringstrings` package to convert the string to all lower case; then it strips the spaces from the answer using `\convertchar`, also from the `stringstrings` package. Next it strips commas from the answer. The final `\@LRT@TheString` makes the answer global.

```
\gdef\@LRT@fieldName#1{%
\caselower[q]{#1}%
\convertchar[q]{\thestring}{ }{}%
\convertchar[q]{\thestring}{,}{}%
\xdef\@LRT@TheString{\thestring}%
}

```

\@LRT@getFilePaths `\@LRT@getFilePaths#1=#2@` is a recursive macro which works through the `prefix={<prefix-file>}` argument list. The first argument, `#1`, picks up one `prefix={<prefix-file>}` and `#2` contains the rest of the arguments. If `#1` is not empty it is passed to `\@LRT@extractKeyValue` which collects the prefix and file path.

```
\gdef\@LRT@getFilePaths#1=#2@{%
\xdef\@LRT@hashKey{\trim@spaces{#1}}%
\if@LRT@empty{\@LRT@hashKey}{}{
\def\@LRT@test{tex}\ifthenelse{\equal{\@LRT@hashKey}{\@LRT@test}}{}%
{\def\@LRT@test{csv}\ifthenelse{\equal{\@LRT@hashKey}{\@LRT@test}}{}%
{\def\@LRT@test{tmp}\ifthenelse{\equal{\@LRT@hashKey}{\@LRT@test}}{}%
{\@LRT@unknownArgumentMessage{#1}}}}}%
\@LRT@extractKeyValue#2@%
\expandafter\@LRT@getFilePaths\@LRT@doWeStillHaveStuffToDo @}}

```

\@LRT@extractKeyValue `\@LRT@hashKey` contains the prefix and `\@LRT@hashValue` contains the file path. If the prefix is not one of `tex`, `csv` or `tmp` you will be warned.

```
\def\@LRT@extractKeyValue#1#2@{%
\edef\@LRT@hashValue{\trim@spaces{#1}}%
\xdef\@LRT@doWeStillHaveStuffToDo{\trim@spaces{#2}}%
\expandafter\xdef\csname @LRT@\@LRT@hashKey\endcsname{\@LRT@hashValue}%
}

```

\@LRT@SecondifFieldEmpty `\@LRT@SecondifFieldEmpty` checks for an empty macro directly. The previously defined `\ifFieldEmpty` uses a different method so both are needed at various points in the code.

```
\newcommand{\@LRT@SecondifFieldEmpty}[3]{%
\edef\@LRT@fieldValue{#1}%
\ifx\@LRT@fieldValue\@empty\relax{#2}\else{#3}\fi\relax}

```

\@LRT@getCommaFieldDelimited The macros `\@LRT@getCommaFieldDelimited`and `\@LRT@getFieldDelimited` are the same when the field delimiter is the comma. The list of field-names is always comma delimited and so we need a `\@LRT@getCommaFieldDelimited` to iterate through it.

```
\gdef\@LRT@getCommaFieldDelimited#1,{\@LRT@getFirstArg{#1}}

```

New counter The counter containing the row number.

```
\newcounter{\@LRT@macroPrefix Row}

```

New read/write's

```
\newread\@lrt@csvFile%
\newread\@lrt@texFile%
\newwrite\@lrt@tmpFile%

```

Messages

```
\gdef\@LRT@justProcessedMessage{%
\immediate\typeout{}%
\immediate\typeout{Just processed row %
\@LRT@Row\space of the data file.}%
\immediate\typeout{}%
\stepcounter{\@LRT@macroPrefix Row}\relax%
}

\gdef\@LRT@failedToOpenMessage#1{%
\immediate\typeout{}%
\immediate\typeout{File #1 failed to open.}%
\immediate\typeout{}\fi%
\end{document}%
}

\gdef\@LRT@NoMoreDataMessage{%
\immediate\typeout{No more data.}%
\immediate\typeout{}%
}

\gdef\@LRT@missingArgumentMessage#1{
\immediate\typeout{}%
\immediate\typeout{I was never given a path to the #1 file.}%
\immediate\typeout{}%
\end{document}%
}

\gdef\@LRT@unknownArgumentMessage#1{
\immediate\typeout{}%
\immediate\typeout{I do not recognize an argument of type #1.}%
\immediate\typeout{}%
}

```

Inputs When `mergeFields` starts up, it knows immediately whether it is in streaming or in storage mode. It sets `\@LRT@Input` to `\@LRT@streaming` or `\@LRT@storage`. Each time a new line of data in the `csv`-file is read the the macros are updated to reflect that data on the current line. `\@LRT@streaming` uses the LaTeX command `\input` to add the `tex`-file to the stream. `\@LRT@storage` works through the data in the `tex`-file. First it opens the `tex` file; then it reads the `tex`-file one line at a time; expands that line; and writes the answer to a the `tmp`-file. When the `eof` for the `tex`-file is reached, the `tex`-file is closed.

`\@LRT@catcodeMagic` is used in two places to set the catcodes of certain characters to values that won't cause trouble.

`\@LRT@texNotAtEOF{#1}` is used to tighten up the code looping over the lines in the `tex`-file in `\@LRT@storage`.

```
\gdef\@LRT@streaming{\input{\@LRT@tex}}

\gdef\@LRT@catcodeMagic{%
\catcode`\^^M9\relax%
\catcode`\#12\relax
\catcode`\$12\relax
\catcode`\&12\relax
\catcode`\^12\relax
\catcode`\_12\relax
\catcode`\%12\relax
}

\gdef\@LRT@texNotAtEOF#1{%
\ifeof\@lrt@texFile{\global\booltrue{@LRT@texEOF}}\else{#1}\fi%
}

\gdef\@LRT@storage{{%
\@LRT@catcodeMagic%
\openin\@lrt@texFile=\@LRT@tex\relax%
\@LRT@texNotAtEOF{\global\boolfalse{@LRT@texEOF}}%
\unlessboolexpr{bool{@LRT@texEOF}}{%
\read\@lrt@texFile to \@LRT@preItem%
\@LRT@texNotAtEOF{%
\immediate\write\@lrt@tmpFile{\@LRT@preItem}\relax}%
}{}%
\immediate\closein\@lrt@texFile%
}}

```

`\@LRT@dataLineIterator` This macro looks to see if the next character is `\@LRT@textDelimiterFront`. If it is the entry is collected using `\@LRT@getTextDelimited` and if not we collect it using `\@LRT@getFieldDelimited`.

```
\gdef\@LRT@dataLineIterator{%
\expandafter\@ifnextchar\@LRT@textDelimiterFront%
{\@LRT@getTextDelimited}{\@LRT@getFieldDelimited}}

```

`\@LRT@getFirstArg` This is the `\@LRT@getFirstArg` from the `\@LRT@getFieldDelimited` and `\@LRT@getTextDelimited` macros. It collects the first field and then uses `\@LRT@getSecondArg` to collect the rest of the line from the `csv`-file.

```
\gdef\@LRT@getFirstArg#1{\gdef\@LRT@firstArg{#1}\@LRT@getSecondArg}
\gdef\@LRT@getSecondArg#1@{\gdef\@LRT@restArg{#1}}

```

`\@LRT@getLOFirstArg` These next two macros do the same thing with the first line of the `csv`-file. The difference is that these two macros put their answers in a different pair of macros than the previous two so we can iterate along both the first line (field-names) and the current line of the `csv`-file.

```
\gdef\@LRT@getLOFirstArg#1{\gdef\@LRT@ffirstArg{#1}\@LRT@getLOSecondArg}
\gdef\@LRT@getLOSecondArg#1@{\gdef\@LRT@frestArg{#1}}

```

`\@LRT@getLineOne` This is the version of `\@LRT@getFirstArg` which is used on the first line of the `csv`-file.

```
\gdef\@LRT@fieldNameIterator#1,{\@LRT@getLOFirstArg{#1}}

```

`\@LRT@setFirstLineAndBlankLine` This macro collects the first line of the `csv`-file and packages it for later use: `\@lrt@firstline` is the first line when we are done. We also construct `\@LRT@csvBlankLine` which is used to check if a line of the `csv`-file is blank.

```
\providebool{@LRT@iterator}
\gdef\@LRT@setFirstLineAndBlankLine#1{%
\gdef\@lrt@firstline{}%
\gdef\@LRT@csvBlankLine{}
\edef\@LRT@restArg{#1\@LRT@fieldDelimiterBack}
\booltrue{@LRT@iterator}%
\whileboolexpr{bool{@LRT@iterator}}{%
\expandafter\@LRT@dataLineIterator\@LRT@restArg @%
\@LRT@fieldName{\@LRT@firstArg}%
\xdef\@lrt@firstline{\@lrt@firstline\@LRT@TheString,}%
\if@LRT@empty{\@LRT@restArg}%
{\boolfalse{@LRT@iterator}}{\booltrue{@LRT@iterator}}%
\ifboolexpr{bool{@LRT@iterator}}%
{\xdef\@LRT@csvBlankLine{\@LRT@csvBlankLine\@LRT@blankFieldItem}}{}%
}{}%
}

```

`\@LRT@getAndProcessNextLine` This is the recursive macro which processes each line of the `csv`-file until there are no more. The code in the first `{{` ... `}}` pair reads a line from the `csv`-file: the last line here converts the answer from local to global. The code in the second `{{` ... `}}` pair checks if the line is blank or not. `\@LRT@exitGetAndProcessNextLine` exits the recursion and `\@LRT@repeatGetAndProcessNextLine` reloops.

```
\gdef\@LRT@getAndProcessNextLine{%
\unlessboolexpr{bool{@LRT@csvEOF}}{%
{{%
\@LRT@catcodeMagic%
\@LRT@catcodes
\read\@lrt@csvFile to\@LRT@x@nextLine%
\ifeof\@lrt@csvFile\relax{\global\booltrue{@LRT@csvEOF}}\else%
{\global\boolfalse{@LRT@csvEOF}}\fi%
\xdef\@LRT@nextLine{\@LRT@x@nextLine}%
}}%
{{%
\if@LRT@empty{\@LRT@nextLine}%
{\global\let\@LRT@next\@LRT@exitGetAndProcessNextLine\relax}%
{\ifx\@LRT@csvBlankLine\@LRT@nextLine\relax%
{\global\let\@LRT@next\@LRT@exitGetAndProcessNextLine\relax}\else%
{\global\let\@LRT@next\@LRT@repeatGetAndProcessNextLine\relax}\fi}%
\@LRT@next%
}}%
}{}%
}

```

`\@LRT@exitGetAndProcessNextLine` This just exits.

```
\gdef\@LRT@exitGetAndProcessNextLine{%
\closein\@lrt@csvFile\relax\global\booltrue{@LRT@csvEOF}}

```

`LRT@repeatGetAndProcessNextLine` This sets up to process the line we just read.

```
\gdef\@LRT@repeatGetAndProcessNextLine{%
\@LRT@processNextLine{\@LRT@nextLine\@LRT@fieldDelimiterBack}}

```

`\@LRT@setMacrosForOneEntry` This bit of code sets the macro and the if-empty boolean for a single item: #1 is the field-name and #2 is the entry in that (row,column).

```
\gdef\@LRT@setMacrosForOneEntry#1#2{%
\expandafter\xdef\csname\@LRT@macroPrefix#1\endcsname{#2}%
\@LRT@SecondifFieldEmpty{#2}%
{\global\booltrue{\@LRT@macroPrefix#1}}%
{\global\boolfalse{\@LRT@macroPrefix#1}}%
}

```

`\@LRT@processNextLine` This code actually processes the line from the `csv`-file. The first three lines are setup. The two `\edef`'s do the following: `\@LRT@restArg{#1}` has the entire current row; and `\@LRT@frestArg` has a comma separated list of field-names. Setting `\booltrue{@LRT@iterator}` means the iteration will start. The two `\expandafter` lines do the iteration: the current values are in `\ffirstArg` for the field-name and `\firstArg` for the entry in the current row. The line `\@LRT@setMacrosForOneEntry` sets the contents of the field-name-macro. Finally the `\if@LRT@empty` line checks if we are done.

The `\edef` between the two `\expandafter`'s adds a blank entry to the back of `\@LRT@restArg` so if the row is too short in the `csv`-file it will still scan correctly.

The `\@LRT@Input` outputs the result either to the stream or to the `tmp`-file and `\@LRT@justProcessedMessage` writes to the terminal which row was just processed. If something suddenly goes wrong you know which row you were working on when things messed up.

```
\gdef\@LRT@processNextLine#1{%
\edef\@LRT@restArg{#1}%
\edef\@LRT@frestArg{\@lrt@firstline}%
\booltrue{@LRT@iterator}%
\whileboolexpr{bool{@LRT@iterator}}{%
\expandafter\@LRT@fieldNameIterator\@LRT@frestArg @%
\edef\@LRT@restArgA{\@LRT@restArg\@LRT@fieldDelimiterBack}
\expandafter\@LRT@dataLineIterator\@LRT@restArgA @%
\@LRT@setMacrosForOneEntry{\@LRT@ffirstArg}{\@LRT@firstArg}%
\if@LRT@empty{\@LRT@frestArg}%
{\boolfalse{@LRT@iterator}}{\booltrue{@LRT@iterator}}%
}{}%
\@LRT@Input\relax%
\@LRT@justProcessedMessage%
}

```

`\@LRT@mergeData` This macro opens the `csv`-file, reads the first row. The code between the `{{ ... }}` makes a comma separated list of field-names in the order in which they occur in the `csv`-file. It also creates a macro which expands to what a blank line in the `csv`-file will look like. It also defines one boolean for each field-name. This requires an iteration over `\@lrtfirstline`.

Because TEX `\newif`'s are not global, we need to define the ones we need as soon as possible, which is right here.

Then `\@LRT@mergeData` calls `\@LRT@getAndProcessNextLine` which reads and processes the subsequent lines of the `csv` file.

Finally it puts out a no-more-data message to let you know `\mergeFields` is finished.

```
\gdef\@LRT@mergeData{%
\providebool{@LRT@texEOF}%
\immediate\openin\@lrt@csvFile=\@LRT@csv\relax%
\providebool{@LRT@csvEOF}%
\boolfalse{@LRT@csvEOF}%
\ifeof\@lrt@csvFile\@LRT@failedToOpenMessage{\@LRT@csv}\fi%
{{\@LRT@catcodes
\read\@lrt@csvFile to\@lrt@firstlineX%
\@LRT@setFirstLineAndBlankLine{\@lrt@firstlineX}
}}%
\booltrue{@LRT@iterator}
\edef\@LRT@restArg{\@lrt@firstline}%
\whileboolexpr{bool{@LRT@iterator}}{%
\expandafter\@LRT@getCommaFieldDelimited\@LRT@restArg @%
\providebool{\@LRT@macroPrefix\@LRT@firstArg}%
\if@LRT@empty{\@LRT@restArg}%
{\boolfalse{@LRT@iterator}}{\booltrue{@LRT@iterator}}%
}{}%
\@LRT@getAndProcessNextLine%
\@LRT@NoMoreDataMessage%
}

```

Options code Here is the code which examines the option string and maps the public macro names to the private codes which actually do the work.

```
\gdef\@LRT@weirdOption#1{\@ifundefined{@LRT@CO}%
{\global\csletcs{#1}{@LRT@#1}}{{\global\csletcs{\@LRT@CO #1}{@LRT@#1}}}%
}

\@LRT@weirdOption{mergeFields}%
\@LRT@weirdOption{Field}%
\@LRT@weirdOption{ifFieldEmpty}%
\@LRT@weirdOption{setDelimitersCommaQuote}%
\@LRT@weirdOption{setDelimitersTabQuote}%
\@LRT@weirdOption{makeMePublic}%

```

Set the default delimiters Here is where the default delimiters are set.

```
\@LRT@setDelimitersCommaQuote

```

Finish up

```
\makeatother
\endinput
```

A Writing new delimiter macros

If the field delimiter is not something TEX sees as white space, life is fairly easy. In your master-file, copy and paste one of the code examples below. Change to suit your needs.
****

```
{{%
\makeatletter
\gdef\setMyDelimiters{%
\gdef\@LRT@fieldDelimiterBack{,}
\gdef\@LRT@getFieldDelimited##1,{\@LRT@getFirstArg{##1}}
\gdef\@LRT@textDelimiterFront{"}
\gdef\@LRT@textDelimiterBack{",}
\gdef\@LRT@getTextDelimited"##1",{\@LRT@getFirstArg{##1}}
\gdef\@LRT@blankFieldItem{,}
\gdef\@LRT@catcodes{}
}%
\makeatother
}}
```

****

Replace all occurrences of `,` by your field delimiter. Replace all occurrences of `"` with your text deliminator and you should be ready to go. Just add `\setMyDelimiters` in your master-file before using `\mergeFields`.

If the field delimiter is a white space character such as a tab, then copy and paste the next example
****

```
\makeatletter
{{\makeatletter
\catcode‘\^^I12\relax%
\gdef\@LRT@setDelimitersTabQuote{
\gdef\@LRT@fieldDelimiterBack{^^I}
\gdef\@LRT@getFieldDelimited##1^^I{\@LRT@getFirstArg{##1}}
\gdef\@LRT@textDelimiterFront{"}
\gdef\@LRT@textDelimiterBack{"^^I}
\gdef\@LRT@getTextDelimited"##1"^^I{\@LRT@getFirstArg{##1}}
\gdef\@LRT@blankFieldItem{}
\gdef\@LRT@catcodes{\catcode‘\^^I12\relax}
}%
}}
\makeatother
}}
```

****

Use Chapter 8 of the TEX book to find how to write your delimiter and replace all occurrences of `\^^I` with the code for your delimiter. Then replace all occurrences of `"` with your text deliminator and hope for the best. Try leaving `\@LRT@blankFieldItem` blank and hope for the best. Again add `\setMyDelimiters` in your master-file before using `\mergeFields`.

B Some examples

B.1 TEX code in a table

A typical line in a tabular environment looks like `item1 & item2& item3\\`. Suppose you have 4 columns in your `csv`-file with entries `Last Name`, `First name`, `Numerator` and `Denominator` and you want `item1` to be `Last Name`, `item2` to be `First name` and `item3` to be `\frac{Numerator}{Denominator}`. Here is the needed code in your `tex`-file:

```
\Field{lastname}&\Field{firstname}&%
$\noexpand\frac{\Field{numerator}}{\Field{denominator}}$\noexpand\\
```

You might have expected to see a `\noexpand` in front of the $'s but remember $ is one of those characters which goes through unscathed to the `tmp`-file. It certainly doesn't hurt to write `\noexpand$`. When you read the `tmp`-file back in you do want `$` to resume its meaning of making math mode and it does.

The `\noexpand` in front of the `\frac` and the `\\` are needed. You don't want to try to expand either one until the `tmp`-file is read back in.

Assuming that the file `merge.tex` contains the code above and that the file `data.csv` contains your data, the following code typesets the table.

```
\documentclass[12pt]{article}
\RequirePackage{csvmerge}
\begin{document}
\mergeFields{
   tmp={example.tmp}
   tex  ={merge.tex}
   csv =  {data.csv}
   }
\begin{tabular}{lll}
\input{example.tmp}
\end{tabular}
\end{document}
```

The `csv`-file needs to contain the four field-names that appear in the `tex`-file, but it is fine if it contains additional columns.

B.2 TEX code in a mail-merge

Here is a snippet of code to do a header in a mail-merge business letter. The field-names from the `csv`-file should be obvious. The interesting point is the usage of `\ifFieldEmpty`.

```
\Field{firstname} \ifFieldEmpty{middleinitial}{}
{\Field{middleinitial}.~}\Field{lastname}\par
\ifFieldEmpty{streetline1}{}{\Field{streetline1}\par}
\ifFieldEmpty{streetline2}{}{\Field{streetline2}\par}
\ifFieldEmpty{streetline3}{}{\Field{streetline3}\par}
\Field{city}, \Field{stateprovince} \Field{postalcode}
```

The `middleinitial` column is a middle initial with no period. Some recipients will have no middle initial but you don't want just a floating period. Also you don't want an additional space. Knuth also suggests that names should be TEXed with `First M.~Last` [The TEX book, Ch. 6]. The `\ifFieldEmpty` with `streetline1` is probably redundant but often `streetline2` will be empty. If it is not, we want a new line after it, but if it is empty we don't want a blank new line.

In storage mode you need a `\noexpand` before the `~`, as well as before the `\par`'s.

B.3 Some master-files

B.3.1 A table

We are doing a table so we want storage mode. We also want a horizontal line after each row of the table. The master-file:

```
\documentclass[12pt]{amsart}
\RequirePackage{csvmerge}
\begin{document}
\mergeFields{
tmp= { ./table.tmp}
csv =  {./data.csv}
tex  ={./table.tex  }
}
\begin{tabular}{|l|l|l|l|l|}
\hline%
\input{./table.tmp}
\end{tabular}
\end{document}
```

The `tex`-file `table.tmp`:

```
% one table line
\Field{firstname} &\Field{id} & \Field{city}  &\Field{lastname}&
\Field{postalcode}  \noexpand\\
\noexpand\hline
```

B.3.2 A mail-merge

We are doing a mail-merge so we can use streaming mode. Since the `tex`-file name has spaces, we have quoted it. The master-file:

```
\documentclass[12pt]{letter}
\RequirePackage{./csvmerge}
\begin{document}
\mergeFields{ tex  = {"./Second letter.template"  }
csv ={  ./Invites.csv}}
\end{document}
```

The `tex`-file `Second letter.template`:

```
 %First page of letter
 \setcounter{page}{1}

 \Field{firstname} \ifFieldEmpty{middleinitial}{}%
 {\Field{middleinitial}.~}\Field{lastname}\par
 \ifFieldEmpty{streetline1}{}{\Field{streetline1} \par}
 \ifFieldEmpty{streetline2}{}{\Field{streetline2}\par}
 \ifFieldEmpty{streetline3}{}{\Field{streetline3}\par}
 \Field{city}, \Field{stateprovince} \Field{postalcode}

 \vskip.25in
 Dear \Field{preferredfirstname},\par
 \vskip10pt
 How come no RSVP?
 \vskip10pt
 Bilbo Baggins
\newpage
```

Here is an equivalent version using storage mode. The master-file:

```
\documentclass[12pt]{letter}
\RequirePackage{./csvmerge}
\begin{document}
\mergeFields{ tex  = {"./Second letter.template"  }
csv ={  ./Invites.csv}tmp={letters}}
\input{letters}
\end{document}
```

We must adjust the `tex`-file `Second letter.template`:

```
%First page of letter
\noexpand\setcounter{page}{1}

\Field{firstname} \ifFieldEmpty{middleinitial}{}%
```

```
 {\Field{middleinitial}.\noexpand~}\Field{lastname}\noexpand\par
 \ifFieldEmpty{streetline1}{}{\Field{streetline1} \noexpand\par}
 \ifFieldEmpty{streetline2}{}{\Field{streetline2}\noexpand\par}
 \ifFieldEmpty{streetline3}{}{\Field{streetline3}\noexpand\par}
 \Field{city}, \Field{stateprovince} \Field{postalcode}

 \noexpand\vskip.25in
 Dear \Field{preferredfirstname},\par
 \noexpand\vskip10pt
 How come no RSVP?
 \noexpand\vskip10pt
 Bilbo Baggins
\noexpand\newpage
```

Usually using storage mode when you don't need it is wasteful since you have to create and save a `tmp`-file. It can come in handy if your letter code has lots of field macros and you can't get one `tex`-file to not have overfull/underfull boxes and the like after the various expansions. The `tmp`-file contains all the letters and you can fine tune each one and then just do

```
\documentclass[12pt]{letter}
\begin{document}
\input{letters}
\end{document}
```