rtf2troff
An RTF to troff Translator

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 10 November 1993

1. Introduction

rtf2troff is a document translator that takes an input file in RTF format and writes output suitable for processing by *troff*. It has a number of features, including:

- Production of output that, when run through *troff*, on rare occasions possesses a mild resemblance to the original document.
- Voluminous, inefficient and largely incomprehensible source code (available for free and overpriced at that).
- Often-incomprehensible output, especially for tables.
- Complete lack of support for formulas.
- Support for underlining and strikethrough that generates prize-winning amounts of output. Besides its incredible bulk, this has the additional security feature of being impossible to make sense of for editing. The reckless user is, however, given the option of disabling this valuable form of protection.
- Inability to write *nroff*-specific output, or output specific to the −*me*, −*mm* or −*ms* macro packages.
- Blissful ignorance of the fact that there are any fonts other than the default, except for purposes of boldface and italics.
- Ability to completely lose any unrecognized input, or, for variety, core dump.
- Merges text of footnotes right into the main body of the document. You didn’t really want ’em at the bottom of the page anyway, did you?

Perhaps over time this list will shrink and this section can be removed. But don’t hold your breath.

The intended audience of this document is not those who might use *rtf2troff* for daily work, but programmers who want to know why it’s written the way it is. As such, it discusses aspects of the implementation. The source code should be consulted for further reference. Actually, I suspect you’re reading this because you’ve already looked at the source and couldn’t make any sense of it!

2. Random Implementation Notes

In the output produced by *rtf2troff*, a distinction is made between *content* (or document) text and *formatting* (or control) text. Content text consists of the characters that are actually supposed to appear in the finished document. Formatting text affects how those characters appear. Formatting text may be inline with content text (e.g., “\fR”, “\s+3”) or on a line by itself (e.g., “.ft R”, “.ps +3”).

2.1. State Maintenance Issues

It is possible to write out control language whenever any changes are made to document, section, paragraph or character formatting properties, but that would result in more output than is necessary. Instead, *rtf2troff* maintains notions of two kinds of state: an *internal* state, which is the current state of formatting properties as indicated by control words encountered in the RTF input stream, and *written* state, which tracks the state corresponding to the *troff* control language that has been written to the output (i.e., the state that *troff* will be in).

Changes to formatting properties are simply accumulated in the internal state without writing any output. When content text is to be written out, a check is made for any discrepancy between the accumulated changes in the internal state, and the written state. If there are any differences, control language is generated to bring the written state into sync with the internal state, before writing the content text. This guarantees that the correct formatting properties will apply to the text, and minimizes the amount of control language generated.

Control language to set up the initial state is flushed before anything else comes out. It's flushed when any of the following are about to be written: (i) any content text for the main body of the document (ii) anything at all for headers or footers; (iii) the beginning of a table. The initial state is written using absolute values. State changes are generally written using relative changes to the current state values. Use of relative values allows manual changes to be made to the initial part of the output and have the rest of the document be affected. For instance, you can change the initial line indent, and the rest of the document will follow the change.

RTF files may contain groups. Normally, a group inherits the state of the group containing it, and changes made within the group are discarded when the group ends. To mimic this, a stack of internal states is maintained by *rtf2troff*. When a group begins, a new internal state is pushed on the stack, with the same values as the previous state. This action does not cause any change to the state values, but the occurrence of document, section, paragraph and character formatting control symbols does. When a group ends, the current state is popped off the stack, and the previous state becomes the current state. The may well change the current state values, if changes were made within the group; when the next content text is written, control language to undo those changes is generated.

Internal state 0 is special. It contains all the RTF default values and is the base state in which the writer starts. Moreover, it is never changed because the first token that should be found in an RTF document is ‘‘{’’, which causes a new state (state 1) to be pushed on the stack immediately. Thus the contents of state 0 can be used to restore section, paragraph and character formatting defaults.

Section Defaults—The section properties of state 0 are set to the RTF defaults and are used to restore the section state when ‘‘\sectd’’ occurs.

Paragraph Defaults—The paragraph properties of state 0 are set to the RTF defaults and are used to restore the paragraph state when ‘‘\pard’’ occurs. The ‘‘Normal’’ style is then applied, since the real defaults include not only the static initial values, but also the formatting produced by that style.

Character Defaults—The character properties of state 0 are set to the RTF defaults and are used to restore the character state when ‘‘\plain’’ occurs.

Some groups, such as headers and footers, do *not* inherit the formatting properties of the enclosing group, presumably because the output that results from those groups is not contiguous with that of the preceding or following groups—they generate output that appears possibly far away. To force non-inheritance of the enclosing group's formatting properties, the effects of the ‘‘\pard’’ and ‘‘\plain’’ tokens are applied at the beginning of this kind of group. The specification says that ‘‘\sectd’’ should also be applied, but I don't believe it. Why should the section break style, title page special value, etc. be reset just because you're collecting a header?

A related problem for such groups is that any changes made to the written state while processing them must not be allowed to affect the formatting of text following the group. In other words, the state that *troff* ends up in when the group ends needs to be rewound to the state it was in when the group began. To allow changes to the written state to be forgotten properly at the end of the group, two things must happen. *troff* must be told to revert to the pre-group written state, and *rtf2troff* must revert its own notion of written state. To allow *troff* to revert its state, such groups are processed using environment switches within diversions, to collect the group output in a separate environment and to allow the environment to be restored. To revert the *rtf2troff* written state, a copy of the state is saved before and restored after processing the group.

rtf2troff currently needs only one level of diversion, so only a single state copy is needed. However, the implementation uses a stack in case a more general mechanism is needed in the future.

2.2. Output Line Length Control

RTF paragraphs can contain very long strings of text. To make *rtf2troff* output more readable and editable, long paragraphs are broken into multiple lines. All output is written for fill mode, so these lines will be joined back together by *troff*. However, for this to work, lines must be broken carefully. The best ‘‘natural break’’ is when there is a single space between words. Several implications follow from this observation.

(i) Must not break to a new line when the next character to be written is whitespace, or *troff* will not join the lines back together (whitespace at the beginning of an input line forces a break).

(ii) Must not break in the middle of a word, or there will be extra whitespace in the middle of it when lines are joined.

(iii) Must not lose whitespace at end of broken lines (*troff* tosses whitespace at ends of lines), so only break when there is a single space between words.

(iv) Must not put out an extra newline at the end of the paragraph, if the output line was just broken after the last character written.

To complicate matters, it is also desirable that the following be true: If underlining or stikeout is enabled, the ugly sequences to do them should be forced onto separate lines for each character, no matter what, to make the output more editable. Since this may violate the conditions above (e.g., if only the middle of a word is underlined), use of ‘‘\c’’ may be necessary. However, ‘‘\c’’ should only be written when absolutely necessary, to avoid cluttering up the output.

Three variables are used to keep track of output written to the current paragraph. *inPara* is true if any characters have been written out to the current paragraph. *oLen* is the number of characters written to the current output line of the paragraph. It is zero if the current line is empty. *breakOK* is non-zero if it’s OK to break a line when the next content character is written (if the next character isn’t a space).

When a character is about to be written for a paragraph, if *inPara* is zero, it is assumed that a paragraph is just beginning, otherwise that part but not all of a paragraph has been written out. This assumption is used when writing both content and formatting text. If a content character is about to be written and *inPara* is zero, beginning-of-paragraph processing takes place: space before paragraphs is written out; if the paragraph has a top border, a line is drawn; control language is generated to set the temporary indent (the RTF ‘‘first line indent’’). *inPara* is also checked when writing formatting text that must be on a line by itself: if *inPara* is not zero, content text has been written which must be ‘‘flushed’’ by writing a newline.

2.3. Underlining/Strikethrough

rtf2troff can do continuous or word underlining. If a document conversion is simply for printing purposes, it makes sense to leave underlining conversion on. If you want to edit the converted file, you may want to turn underlining conversion off (with the **−u** option), because it may be difficult to edit the portions of text that are underlined. Another reason to turn underlining off is that some printers seem to take a long time to print documents containing a lot of underlining.

These remarks apply to strikeout text as well.

2.4. Page Orientation

Some versions of Word (e.g., WfM) don't seem to write out the ‘‘\landscape’’ control word properly, so when *rtf2troff* writes out the initial state-setting control language, it checks to see whether the page height is less than the width. If so, it prints a message and assumes landscape is on.

2.5. Tab Handling

The defaults are every .5 inch, left justified, motion only (no leader character). There are *maxTab* tabs (*maxTab* = 20, which might be enough). When the state is pushed, the new state inherits the previous state's tab settings, but if any tabs are set explicitly in the new state, they override inherited tabs. A flag is used to know when the first tab is being set in the new state. When that happens, a new tab set is started instead of adding the tab to the end of the existing set.

Tabs may have position and justification. For a given tab, the position is the *last* attribute specified. Thus, when a justification indicator is encountered in the input stream, it's entered into the *next* tab slot, but the tab count is not incremented. When the position is specified, the count is incremented and any justification previously specified automatically becomes part of the new tab stop.

In RTF, tabs may also be associated with a leader character, but *rtf2troff* only maintains a single tab character. Thus, if multiple tab leader characters are specified for a paragraph, only the last one is used, and it applies to all the tabs. This is certainly unsatisfactory, but that's the way it is.

Bar tabs are ignored. Decimal tabs are treated as right-justified tabs.

2.6. Tables

rtf2troff includes some table support, but tables are hard to do well, so you'll easily find examples which confuse it and convert poorly. At least it was hard for *me* to figure out how to do them well, but then, I don't understand *tbl*. Does anyone?

Tables are written out under the assumption that you have the *tbl* program available. Each row of a table is written out between .TS/.TE pairs. Cell contents are written out using *tbl*'s ‘‘T{’’...‘‘T}’’ construct.

One problem with writing tables using the T{/T} mechanism is that *tbl* tries to keep font, point size and vertical spacing changes within one cell from affecting the next. Since RTF tables may well expect changes to these three parameters to carry over to following cells, *tbl*'s invisibly resetting these introduces a problem. The solution to this is straightforward, but results in output that's even uglier than usual: force out the current font, point size and vertical spacing at the beginning of each cell. It's also necessary to do this after the ‘‘.TE’’ since that seems to mess with point size and vertical spacing.

One optimization that ought to be possible (says he) to minimize the amount of output generated is to compare the current values to the values in force at the beginning of the table and only write them out if they're different. Unfortunately, for reasons I don't understand, this doesn't always work. Thus brute force prevails.

Another problem arises in connection with column widths. *tbl* assumes a default column separation of 3 ens, in the current point size. *rtf2troff* knows the width that a cell should be and can write the table specification as such. Unfortunately, the column separation is added to that width, it's not a part of it, so the table ends up wider than it should be. A separation of zero can be used, which will make the table the right width, but the text is too jammed together. If there are borders, the text hits the borders.

Specifying a 1-en separation spaces the text away from the cell borders, but then the table is again too wide. This could be handled if there were some easy way of subtracting 1 en from the column width, but there isn't. Cell widths are absolute values, whereas ens depend on the current point size. You can't specify a column width in the table heading, e.g., as ‘‘l1w(1.5i-1n)’’, either. It *almost* always works, but some tables botch terribly. The ‘‘solution’’ adopted in *rtf2troff* is to try to guess how big an en is in the current point size and subtract it from the column width before writing the table header. This works pretty well if you can get accurate width data for your version of *troff*.

Cell borders are handled to a small extent. One problem is that each row of an RTF table is written as a separate *tbl* table, and the .TS/.TE macros can result in a little space before and after the table if you use a macro package such as **−me**. This means that ugly gaps between rows may result.

Tabs within cells are ignored, i.e., mishandled.

Merged cells are botched.

In general, the table-writing code tends to do fairly well with simple tables and less well with more complex ones. The problem with the output generated for tables is that it is very ‘‘busy’’ and if it’s incorrect, it’s not always evident how to correct it, should you wish to do so manually.

Some newer Xerox 4045 printers have problems with complex tables generated by *xroff*. This is not specific to output generated by *rtf2troff*, so if tables appear to be botched, it might not be *rtf2troff*’s fault.

2.7. Character Translation

rtf2troff in RTF Tools distributions up through release 1.08 used a complicated character mapping based on a division of the character set into ASCII and ‘‘special’’ characters. The special characters were mapped to *troff* output sequences based on the charset used in the input file. Character mapping has been substantially revised as of release 1.09. There is no longer any distinction made between ASCII and non-ASCII characters, nor does writer code need to know anything about the charset. See the document *RTF Tools Character Mapping*.

There is an output map for each version of *troff* you want *rtf2troff* to know about. There may also be macro-package-specific output maps if you like.

The maps are selected from the command line with the **−t** and **−me|−mm|−ms** options. For instance, if your site supports *xroff* and *pstroff*, you can select maps for one or the other with

```
rtf2troff -t xroff
rtf2troff -t pstroff
```

It might be more convenient simply to create shell scripts, e.g., *rtf2xroff*, which would contain

```
#!/bin/sh
exec rtf2troff -t xroff "$@"
```

If you write output maps for *troff* versions other than those supported by the distribution, please send them to me.

2.8. Error messages

Tokens not recognized by the reader are echoed.

‘‘Uh-oh! RTF group nesting exceeded maximum level,’’: the maximum stack depth needs to be increased.

‘‘unbalanced brace level’’: the RTF file is malformed; some ‘‘{’’ is not matched by a ‘‘}’’.

‘‘unrestored environment’’: indicates a bug in *rtf2troff*.

‘‘unrestored indirection’’: indicates a bug in *rtf2troff*.

If you get a message ‘‘Trap Loop Death detected’’, it means the header and footer overlap. This is legal in RTF, but *troff* doesn’t like it and can end up in an infinite loop because the header triggers the footer trap, which triggers the header trap, which, etc. You should only see this message if you’ve specified +*h* to cause header/footer macros and traps to be generated. Try again without +*h* and the output should be better behaved.

Most other messages come from inside the reader, e.g., if the stylesheet or font table readers get confused.

2.9. Registers and Macros

A number of register and macro names are used by *rtf2troff*. If these conflict with those used by any pre-processors you might use, you can change them by editing *rtf2troff.h*.