A Tool for RTF Processing
Version 1.06a1

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 5 April 1991

Also installs pict group reader, presumably because it's horrible to parse and it's easier to just skip it.

1. Introduction

This document describes a general purpose tool for processing RTF files—an RTF reader which may be configured in a well-defined manner to allow it to be used with a variety of writers generating different output formats. This provides a method for generating RTF-to-*XXX* translators. In theory.

I assume that you have some familiarity with RTF syntax and semantics, and that you're willing to study the source code of the RTF distribution described here. If you don't have the RTF specification, you can get it from the FTP site listed under ''Distribution Availability.'' References to ''the specification'' refer to this document.

If you use this tool and find that you have an RTF file that won't pass through the sample translator *rtf2null*, or for which *rtf2null* announces unknown symbols, please contact me so the tool can be improved. It is best if you can supply the RTF file for which this behavior is observed.

2. Theory of Operation

2.1. Translator Architecture

There are three components to an RTF translator (at least as conceived here): reader code, writer code, and setup code. These break down as follows.

reader
: Responsible for peeling tokens out of the input stream, classifying them, and causing the writer to process them.

writer
: Responsible for translating tokens from the input stream into the required output format.

setup
: Responsible for making sure the reader and writer are initialized, and for calling the reader, to cause translation to occur.

This architecture allows the reader to remain constant, so that different translaters can be built by supplying different writer and setup code.

In practice, to build a new translator, you supply a *main()* function and the writer code, and link in the RTF reader. *main()* includes the setup code and is responsible to see that the following are done:

- Process command-line arguments

- Configure the reader, which may involve:
  - Reset the input stream if necessary
  - Configure other reader behavior, such as whether to process the font and color tables internally.
  - Install writer callbacks into the reader so it knows what functions to call when various kinds of tokens occur
- Initialize the writer
- Call the reader to process input stream
- Terminate the writer

The minimal translator looks something like this:

```
# include	<stdio.h>
# include	"rtf.h"

int main ()
{
	RTFInit ();
	RTFRead ();
	exit (0);
}
```

This initializes the reader, and calls it to read *stdin*. The writer portion is null (i.e., there is no writer), so all that happens is that the reader tokenizes the input and discards it. That isn’t very interesting; most of the sample translators are examples of more elaborate translators.

2.2. Reader Operation

Tokens are classified using up to three numbers: token class, and major and minor numbers. The class number can be:

rtfUnknown	unrecognized token
rtfGroup	‘‘{’’ or ‘‘}’’
rtfText	plain text character
rtfControl	token beginning with ‘‘\’’
rtfEOF	fake class number; indicates end of input stream

There are some exceptions. A few tokens beginning with ‘‘\’’ actually belong to other classes, a tab character is treated like ‘‘\tab’’, and unrecognized tokens are put in class *rtfUnknown* no matter what they look like.

Within a class, tokens are assigned a major number, and perhaps a minor number. For the *rtfText* class, the major number is the value of the character (0..255), and there is no minor number. For the *rtfControl* class, most tokens have both a major and minor number. For instance, all paragraph attribute control symbols have major number *rtfParAttr* and a minor number indicating which property, such as *rtfLeftIndent* or *rtfSpaceBefore*. A few oddball control tokens have no minor number.

A ‘‘plain text’’ character can be a literal character, a character specified in hex notation (‘‘\`*xx*’’) or one of the special escaped characters (‘‘\{’’, ‘‘\}’’, ‘‘\\’’). The sequence ‘‘\:’’ is treated as a plain text colon. This is arguably wrong; the rationale is given later under the description of the *RTFGetToken()* function.

Ideally, there should never be any tokens in the *rtfUnknown* class, but as the RTF standard continues to develop, unknown tokens are inevitable.

To write a translator, you'll need to familiarize yourself with the token classification scheme by reading *rtf.h*. A skeleton translator *rtfskel.c* is included with the distribution and may be used as a basis for new translators.

Each time a token is read, several global variables are set. *rtfClass*, *rtfMajor*, and *rtfMinor* indicate the token class, and major and minor numbers. (The major and minor numbers may be meaningless depending on the kind of token.) Control symbols may have a parameter value, e.g., ''\margr720'' specifies a right margin (in units of 720 twentieths of a point). The parameter value is stored in *rtfParam*. The text of the token (including the parameter text) is placed in *rtfTextBuf* and its length in *rtfTextLen*.

If no parameter value is given, *rtfParam* is 0, which is indistinguishable from an explicitly specified parameter of ''0''. If you need to tell the difference, examine *rtfTextBuf[rtfTextLen-1]* to see if it's a digit or not.

The reader assumes a 7-bit character set. The specification indicates that character values ≥ 128 may be encoded with the ''\'*xx*'' sequence. If the reader sees a character with the high bit set, it prints a message and exits.

Generally, a translator will configure the RTF reader to call particular writer functions when certain kinds of tokens are encountered in the input stream. These functions are known as *class callbacks*. Writer callbacks can be registered with the reader using *RTFSetClassCallback()* for each token class.

The reader reads each token, classifies it, and sends it to a token routing function *RTFRouteToken()*, tries to find a writer callback function to process the token. Tokens in a given class are ignored if no callback is registered for the class.

Class callbacks make it quite easy to receive notification when certain types of tokens occur in the input. For instance, a crude RTF text extractor could be written by installing a callback function for the *rtfText* class.[1] Whenever the function is invoked, *rtfMajor* will contain a value in the range 0..255 representing the character value.

```
# include     <stdio.h>
# include     "rtf.h"

void TextCallback ()
{
        putchar (rtfMajor);
}

int main ()
{
        RTFInit ();
        RTFSetClassCallback (rtfText, TextCallback);
        RTFRead ();
        exit (0);
}
```

Callbacks for the *rtfControl* and *rtfGroup* classes typically operate by selecting on the token major number to determine the action to take. A callback for the *rtfGroup* class usually will do something like this:

[1] The reasons this is a crude translator are that: (i) some text characters occur in contexts where the characters are not intended to be output, e.g., font tables, stylesheets; (ii) character values greater than 127 probably should be translated into the normal ASCII range; (iii) some control symbols like ''\tab'' represent output text characters. The sample translator *rtf2text* addresses these problems in a (slightly) more sophisticated manner.

```
void BraceCallback ()
{
	switch (rtfMajor)
	{
	case rtfBeginGroup:
		...push state...
		break;
	case rtfEndGroup:
		...pop state...
		break;
	}
}
```

2.3. Destination Readers

Grouping in RTF documents occurs within braces ‘‘{’’ and ‘‘}’’. One kind of group is the *destination*. The token immediately following the opening brace is a destination control symbol. These indicate such things as headers, footers, footnotes, etc.

Three destinations which specify information for internal use (i.e., information which affects output but isn’t itself written) are the font table, color table and stylesheet. Since these three destinations occur so commonly and have a special syntax, the RTF reader by default gobbles them up itself when it recognizes them. The functions which do this are called *destination readers* and are probably the nearest thing in the reader to what might be called parsers. They are installed by default so that translators can be written without the burden of understanding the syntax or digesting the contents of these destinations. Each of them constructs a list of the entries specified in the destination and the reader includes functions providing access to these lists.

Translators can turn off or override these defaults with *RTFSetDestinationCallback()* if necessary. To override one, pass the address of a different destination reader function. To turn one off, pass NULL.

Destination callbacks may be called for any destination, not just *rtfFontTbl*, *rtfColorTbl* and *rtfStyleSheet*. Destinations for which no callback is registered are not treated specially.

Other destinations for which there is a default reader are the information (‘‘\info’’) and picture (‘‘\pict’’) destinations; all they do is skip to the end of the group.

2.3.1. Using the Built-in Destination Readers

The font table, color table and stylesheet information is maintained internally, and the reader either acts on that information itself, or allows itself to be queried by the writer about it, as described below. These descriptions do not apply if the translator shuts off or overrides the default destination readers, of course.

Stylesheet—The reader acts on this itself. When the stylesheet destination is encountered, the style contents are remembered. Thereafter, whenever the writer receives notification that a style number control symbol (‘‘\s*nnn*’’) has occurred, it can call *RTFExpandStyle(rtfParam)* to cause the style to be expanded. The reader consults contents of the stylesheet and each token in the style definition is routed in turn back to the writer. This effects a sort of macro expansion.

If the writer doesn’t care about style expansion, it simply refrains from calling *RTFExpandStyle()*.

If the writer wants information about a style, it can call *RTFGetStyle()*.

Font table—For each entry in the font table, the font number, type and name are maintained by the reader. The writer finds out that a font number has been specified in the input when its control class callback is invoked and *rtfMajor = rtfCharAttr* and *rtfMinor = rtfFontNum*. To obtain a pointer to the appropriate *RTFFont* structure, the reader function *RTFGetFont(rtfParam)* may be called.

Color table—For each entry in the color table, the color number is maintained along with the red, green and blue values. The writer finds out that a color number has been specified in the input when its control class callback is invoked and *rtfMajor* = *rtfCharAttr* and *rtfMinor* = *rtfColorNum*. To obtain a pointer to the appropriate *RTFColor* structure, the reader function *RTFGetColor(rtfParam)* may be called.

One subtle point about the built-in destination readers: destinations cannot be recognized until *after* the occurrence of the ‘‘{’’ symbol that begins the destination. This means the writer, if it maintains a state stack, will already have pushed a state. In order to allow the writer to properly pop that state in response to the ‘‘}’’, these destination readers feed the ‘‘}’’ back into the token router after they pull it from the input stream. What the writer actually sees is a ‘‘{’’ followed immediately by a ‘‘}’’.

Applications that maintain a state stack may find it necessary to do something similar if they supply their own destination readers.

3. Programming Interface

Source files using the RTF reader should #include *rtf.h*. *reader.c* should be compiled to produce *reader.o*, which should be part of the final application link.

The best way to learn how these source files work is to study the sample translators, which vary in complexity from very simple (e.g., *rtf2text*, *rtfwc*), to wretchedly messy (e.g., *rtf2troff*). You should be aware that one implication of the way the translators are built (callbacks and switch statements) is that it’s quite easy to build them incrementally. You can start with a very bare-bones model, and start plugging in callbacks as you progress. Within the callbacks, your switch statements can progressively handle more cases.

An alternative approach is to start with a copy of *rtfskel.c*, which includes a full set of class callbacks and complete switch statements for all tokens. Each case is empty; you simply add code for those cases you want to handle. You can also rip out the code for the cases you know you’ll never care about.

3.1. Global variables

The global RTF reader variables are:

int	rtfClass;	token class
int	rtfMajor;	token major number
int	rtfMinor;	token minor number
int	rtfParam;	parameter value for control symbols
char	rtfTextBuf[rtfBufSiz];	token text
int	rtfTextLen;	length of token text

These variables always apply to the token with which the writer should be concerned. This may be either the last token read or the current token within a style which is being reprocessed.

3.2. Functions

void RTFInit ()

Initialize the RTF reader. This is the first RTF routine that should be called. It performs some initialization such as computing hash values for the token lookup table and installation of the built-in destination readers.

RTFInit() may be called multiple times. Each invocation resets the reader’s state completely, except that the input stream is not disturbed.

void RTFRead ()

RTFRead() calls *RTFGetToken()* to tokenize the input stream and *RTFRouteToken()* to process each token, until input is exhausted. When *RTFRead()* returns, input has been completely read and the writer can perform any cleanup or termination needed.

If you want to read multiple files per invocation of your translator, you should do all your setup prior to each call to *RTFRead()*. That is, you should call *RTFInit()*, install callbacks, etc., then call *RTFRead()*.

void RTFRouteToken ()

This routine decides what to do with the current token and routes it to the correct place for processing. Usually this is directly to the writer via a class callback. The token is *not* passed to the writer (i.e., the class callback is bypassed) when it is a destination token for which a reader callback is installed.

By default, built-in readers are installed for font table, color table, stylesheet and information and picture group destinations. The built-in readers can be disabled if the writer wants to see all tokens directly.

int RTFGetToken ()

Reads one token from the input stream, classifies it, sets the global variables, and returns the class number. If the class is *rtfEOF* the end of the input stream has been reached. Newlines (‘‘\n’’) and carriage returns (‘‘\r’’) are silently discarded by *RTFGetToken()*, as they have no meaning. Both are passed to the token hook if one is installed, however.

The sequence ‘‘\:’’ is treated as a plain text character, with *rtfClass* set to *rtfText* and *rtfMajor* set to the colon ASCII code. Strictly speaking, ‘‘\:’’ is the control word for an index subentry, but some versions of Microsoft Word write out plain text colons with a preceding backslash, while others don’t. This unfortunate ambiguity results in an ugly dilemma. It seems the lesser burden to require translators to recognize that plain text colons should ‘‘really’’ be treated as index subentry indicators while inside of an index entry destination, than to recognize that an index subentry control word should ‘‘really’’ be treated as a plain text colon everywhere else.

Writers probably should not need to use *RTFGetToken()* directly unless they install their own destination readers. One reason you might want to call it is to implement a ‘‘peek at next token’’ capability. Call *RTFGetToken()* and examine the global variables. Then call *RTFRouteToken()* to cause the symbol to be processed normally. This way you get to look at the token before it goes through the usual routing mechanism.

void RTFSetToken (class, major, minor, param, text)
int class, major, minor, param;
char *text;

It is sometimes useful to construct a fake token and run it through the token router to cause the effects of the token to be applied. *RTFSetToken()* allows you to do this, by setting the reader’s global variables to the values supplied. If *param* is non-negative, the token text *rtfTextBuf* is constructed from *text* and *param*, otherwise *rtfTextBuf* is just copied from *text*.

void RTFSetReadHook (f)
void (*f) ();

Install a function to be called by *RTFGetToken()* after each token is read from the input stream. The function takes no arguments and returns no value. Within the function, information about the current token can be obtained from the global variables. This function is for token examination purposes only, and should not modify those variables.

void (*RTFGetReadHook ()) ()

Returns a pointer to the current read hook, or NULL if there isn’t one.

void RTFSkipGroup ()

This function can be called to skip to the end of the current group (including any subgroups). It’s useful for explicitly ignoring ‘‘**dest*’’ groups, where *dest* is an unrecognized destination, or for causing groups that you don’t want to deal with to effectively ‘‘disappear’’ from the input stream.

Calling this function in the middle of expanding a style may cause problems. However, it is typically called when you have just seen a destination symbol, which won’t happen during a style expansion—I think.

Be careful with this function if your writer maintains a state stack, because you will already have pushed a state when the opening group brace was seen. After *RTFSkipGroup()* returns, the group closing brace has been read, and you'll need to pop a state. All global token variables will still be set to the closing brace, so you may only need to call *RTFRouteToken()* to cause the state to be unstacked.

void RTFExpandStyle (num)
int num;

Performs style expansion of the given style number, or does nothing if there is no such style. The writer should call this when it notices that the current token is a style number indicator.

void RTFSetStream (stream)
FILE *stream;

Redirects the RTF reader to the given stream. This should be called before any reading is done. The default input stream is *stdin*. An alternative to *RTFSetStream()* is to simply *freopen()* the input file on *stdin* (that's what all the sample translators do).

The input stream is *not* modified by *RTFInit()*.

void RTFSetClassCallback (class, callback)
int class;
void (*callback) ();

Installs a writer callback function for the given token class. The first argument is a class number, the second is the function to call when tokens from that class are encountered in the input stream. This will cause *RTFRouteToken()* to invoke the callback when it encounters a token in the class. If *callback* is NULL (which is the default for all classes), tokens in the class are ignored, i.e., discarded.

The callback should take no arguments and return no value. Within the callback, information about the current token can be obtained from the global variables.

Installing a callback for the *rtfEOF* ''class'' is silly and has no effect.

void (*RTFGetClassCallback (class)) ()
int class;

Returns a pointer to the callback function for the given token class, or NULL if there isn't one.

void RTFSetDestinationCallback (dest, callback)
int dest;
void (*callback) ();

Installs a callback function for the given destination (*dest* is a token minor number). When *RTFRouteToken()* sees a token with class *rtfControl* and major number *rtfDestination*, it checks whether there is a callback for the destination indicated by the minor number. If so, it invokes it. If *callback* is NULL, the given destination is not treated specially (the control class callback is invoked as usual). By default, destination callbacks are installed for the font table, color table, stylesheet, and information and picture group.

The callback should take no arguments and return no value. When the functon is invoked, the current token will be the destination token following the destination's initial opening brace ''{''. (For optional destinations, the destination token follows the ''*'' symbol.)

void (*RTFGetDestinationCallback (dest)) ()
int dest;

Returns a pointer to the callback function for the given token class, or NULL if there isn't one.

RTFStyle *RTFGetStyle (num)
int num;

Returns a pointer to the *RTFStyle* structure for the given style number. The ‘‘Normal’’ style number is 0. Pass −1 to get a pointer to the first style in the list. Styles are not stored in any particular order.

Be sure to check the result; it might be NULL.

This function is meaningless if the default stylesheet destination reader is overridden.

RTFFont *RTFGetFont (num)
int num;

Returns a pointer to the *RTFFont* structure for the given font number. Pass −1 to get a pointer to the first font in the list. Fonts are not stored in any particular order.

Be sure to check the result; it might be NULL. In particular, you might think that passing the number specified with the ‘‘\deff’’ (default font) control symbol would always yield a valid font structure, but that’s not true. The default font might not be listed in the font table.

This function is meaningless if the default font table destination reader is overridden.

RTFColor *RTFGetColor (num)
int num;

Returns a pointer to the *RTFColor* structure for the given color number. (I think black is 0.) Pass −1 to get a pointer to the first color in the list. Colors are not stored in any particular order. If the color values in the entry are −1, the default color should be used. The default color is writer-dependent.

Be sure to check the result; it might be NULL. I think this means you should use the default color.

This function is meaningless if the default color table destination reader is overridden.

int RTFCheckCM (class, major)
int class, major;

Returns non-zero if *rtfClass* and *rtfMajor* are equal to *class* and *major*, respectively, zero otherwise.

int RTFCheckCMM (class, major, minor)
int class, major, minor;

Returns non-zero if *rtfClass*, *rtfMajor* and *rtfMinor* are equal to *class*, *major* and *minor*, respectively, zero otherwise.

int RTFCheckMM (major, minor)
int major, minor;

Returns non-zero if *rtfMajor* and *rtfMinor* are equal to *major* and *minor*, respectively, zero otherwise.

char *RTFAlloc (size)
int size;

Returns a pointer to a block of memory *size* bytes long, or NULL if insufficient memory was available.

char *RTFStrSave (s)
char *s;

Allocates a block of memory big enough for a copy of the given string (including terminating null byte), copies the string into it, and returns a pointer to the copy. Returns NULL if insufficient memory was available.

void RTFFree (p)
char *p;

Frees the block of memory pointed to by *p*, which should have been allocated by *RTFAlloc()* or *RTFStrSave()*. It is safe to pass NULL to this routine.

4. Distribution Availability

This software may be redistributed without restriction and used for any purpose whatsoever.

The RTF distribution is available for anonymous *ftp* access in the *˜ftp/pub/RTF* directory on host *indri.primate.wisc.edu* (Internet address 128.104.230.11). Updates appear there as they become available.

A version of the RTF specification is available in this directory, as either a binhex'ed Word for Macintosh document, or in RTF format. It is known to have a few errors, as it's a scanned version of a paper copy. Some of these errors have been fixed, but others remain (see, for instance, the example table text on page 17). The document is not quite as up to date as the one sent out by Microsoft, but it is much more complete than the one beginning ''Specification for RTF'' that may be found on some other archive sites.

If you do not have Internet access, send a request to one of the following:

Internet:	software-request@primate.wisc.edu
UUCP:	rhesus!software-request

Bug reports and questions should be sent to one of these addresses as well.

If you use this software as the basis for a translater not included in the current collection, please consider contributing it for inclusion in a future distribution. In particular, an RTF-to-LaTeX translator seems to be an item of interest. I don't use LaTeX myself and am unlikely to write one, but it would probably be fairly popular.