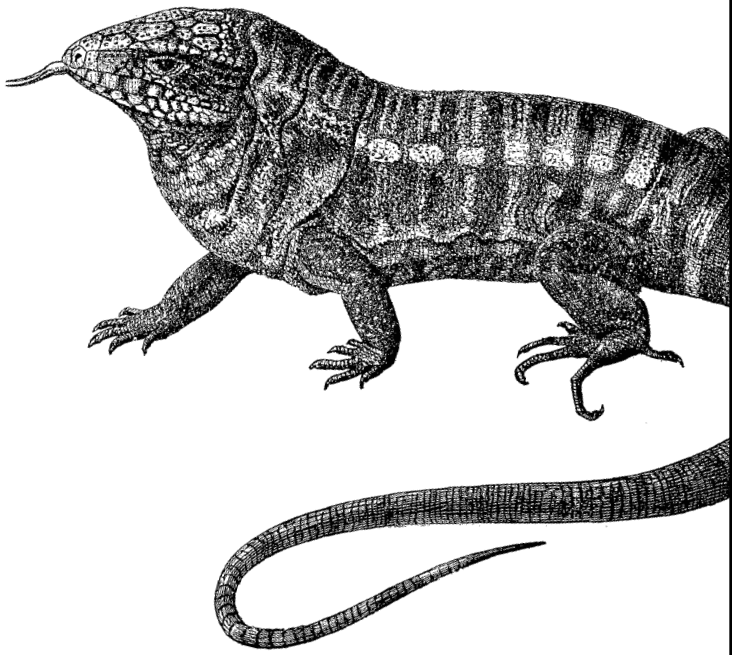
The Universal Document Format

RTF

Pocket Guide



O'REILLY®

Sean M. Burke

RTF

Pocket Guide

RTF
Pocket Guide

Sean M. Burke

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

PART I

RTF Tutorial

This book is a convenient reference for Rich Text Format (RTF). It covers the essentials of RTF, especially the parts that you need to know if you're writing a program to generate RTF files. This book is also a useful introduction to parsing RTF, although that is a more complex task.

RTF is a document format. RTF is not intended to be a markup language anyone would use for coding entire documents by hand (although it has been done!). Instead, it's meant to be a format for document data that all sorts of programs can read and write. For example, if you even just skim this book, you should be able to write a program (in the programming language of your choice) that can analyze the contents of a database and produce a summary of it as an RTF document with whatever kinds of formatting you want. The flexibility of RTF makes it an ideal format for everything from generating invoices or sales reports, to producing dictionaries based on databases of words.

This book is *not* a complete reference to every last feature of RTF; Microsoft's comprehensive but terse *Rich Text Format (RTF) Specification* is the closest you will find to that. The current version (v1.7) is available at *http://msdn.microsoft.com/library/en-us/dnrtfspec/html/rtfspec.asp*. In the Microsoft Knowledgebase at *support.microsoft.com*, its access number is 269575. Version 1.5 of the specification and before are more verbose, and might be more useful. Microsoft doesn't distribute copies of them anymore, but you can find them all

over the Internet by running a search on "Rich Text Format (RTF) Specification" in Google or a similar search engine.

Why RTF?

RTF is a handy format for several reasons.

RTF is a mature format. RTF's syntax is stable and straightforward, and its specification has existed for over a decade—an eternity in computer years. In fact, while there has been a proliferation of incompatible binary formats calling themselves "Microsoft Word file format," RTF has stayed the course and evolved along backward-compatible lines. That means if you generate an RTF file today, you should be able to read it in 10 years, and you should have no trouble reading an RTF file generated 10 years ago.

Many applications understand RTF. Since RTF has been around for so long, just about every word processor since the late 1980s can understand it. While not every word processor understands every RTF feature perfectly, most of them understand the RTF commands discussed in this book quite well. Moreover, RTF is the data format for "rich text controls" in MSWindows APIs; RTF-rendering APIs are part of the Carbon/Cocoa APIs in Mac OS X; and you can even read RTF documents on iPods, Apple's portable music players.

Most people have the software to read RTF. That is, if you email an RTF file to a dozen people you know, chances are that almost all of them can read it with a word processor already on their system, whether it's MSWord, some other word processor (ABIWord, StarOffice, TextEdit), or just the RTF-literate *write.exe* that has been part of MSWindows since at least Windows 98.

In RTF, format control is straightforward. In HTML, if you want to control the size and style of text or the positioning and justification of paragraphs, the best you can do is

try a long detour through CSS, a standard that is erratically implemented even today. In RTF, font size and style, paragraph indenting, page breaks, page numbering, page headers and footers, widow-and-orphan control, and dozens of other features are each a single, simple command.

RTF is a multilingual format. RTF now supports Unicode, so it can represent text in just about every human language ever written.

RTF is easy to generate. You can produce RTF without any knowledge of the font metrics needed for Adobe PostScript or PDF. In addition, since RTF files are text files, it's easy to produce RTF with a program in any programming language, whether it's Perl, Java, C++, Pascal, COBOL, Lisp, or anything in between.

This Book's Approach

This book does not discuss the task of parsing RTF documents. RTF is like many other formats, in that when you want to output in that format, you can stick to whatever syntactic and semantic subset of the language is most convenient for you. But in parsing, you have to be able to accept anything you're given, which may use every last syntactic and semantic oddity mentioned in the spec, and many more that aren't in the spec.

This book explains the simplest kind of RTF, which should work with just about any RTF-aware application. However, I may refer to less-portable commands when necessary, or demonstrate solving a specific problem with an RTF command that also has a broader, more abstract meaning that I do not discuss, for reasons of brevity. If you are particularly interested in the deeper complexities of RTF or of any particular command, you can read this book as a friendly introduction to RTF, and then read the RTF Specification and maybe even view raw RTF code generated by a few different word

processors. But for most programmers, this book is more or less everything you'd ever want to know about RTF.

Here's a quick rundown of what you'll find in this book:

Part I, *RTF Tutorial*
: Part I teaches RTF to the uninitiated. It explains the basic formatting commands and how to work with them.

Part II, *Creating MS Windows Help Files*
: Part II is about creating help files for Microsoft Windows with RTF.

Part III, *Example Programs*
: Part III shows several programming examples in RTF, using the Perl programming language.

Part IV, *Reference Tables*
: Part IV is the reference section of the book. It includes a character chart, a listing of the language codes, and a conversion table for twips measurements. For more help with measurements, see the twips ruler on the inside of this book's back cover.

RTF "Hello, World!"

The first program you ever learned to write probably looked like this:

```
10 PRINT "HELLO, WORLD!"
```

Even though RTF is a document language instead of a programming language, I'll start out the same. Here is a minimal RTF document:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Times New Roman;}}
\f0\fs60 Hello, World!
}
```

If you open a text editor, type that in, save it as *test.rtf*, and then open it with a word processor, it will show you a document consisting of the words "Hello, World!". Moreover, they'll be in the font Times New Roman, in 30-point type.

(We'll go over these commands later, but you may wonder about `\fs60` meaning 30 points—it so happens that the parameter for the font-size command is in half-points.) If you wanted them to be in 14-point Monotype Corsiva, change the document to read like this:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Monotype Corsiva;}}
\f0\fs28 Hello, World!
}
```

If you want the text to be 60-point, italic, bold, and centered, and to have each word on its own line, do it like this:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Monotype Corsiva;}}
\qc\f0\fs120\i\b Hello,\line World!
}
```

Viewed in an MSWord window, that document looks like Figure 1.

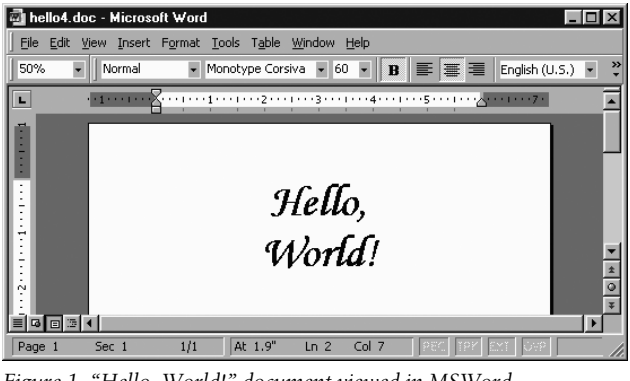


Figure 1. "Hello, World!" document viewed in MSWord

You can get exactly the same document if you remove all the newlines in your RTF, like so (because you're reading this in hardcopy instead of on a monitor, I had to break the line, but pretend I didn't):

```
{\rtf1\ansi\deff0{\fonttbl{\f0 Monotype Corsiva;
}}\qc\f0\fs120\i\b Hello,\line World!}
```

Or you can insert many newlines, in certain places:

```
{\rtf1\ansi\deff0
{\fonttbl
{\f0
Monotype Corsiva;
}
}
\qc
\f0\fs120\i
\b Hello,
\line
World!
}
```

Or you can insert just one or two newlines, but insert a space after each *foo* command, like so:

```
{\rtf1 \ansi \deff0 {\fonttbl {\f0 Monotype Corsiva;}}
\qc \f0 \fs120 \i
\b Hello,\line World!}
```

All these syntaxes mean exactly the same thing. However, this doesn't mean that RTF ignores all whitespace the way many computer languages do. I explain the rules for RTF syntax later, in the section "Basic RTF Syntax," so that you'll know when it's okay to add whitespace.

Overview of Simple RTF

Suppose that instead of "Hello, World!", you want something more classy—in fact, more classical. Suppose you want to say hello in Latin. Latin for "Hello, World!" is "Salvête, Omnês!" The question is, how do you get those "ê" characters? You can't just insert a literal "ê" into the RTF document; although a few word processors tolerate that, by-the-book RTF is limited to newline plus the characters between ASCII 32 (space) and ASCII 126 (the "~" character)—and "ê" is not in that range.

But ê is in the ANSI character set (also known as Code Page 1252, which is basically Latin-1 with some characters added between 128 and 159). An extended ASCII chart shows that ê is character 234 in those character sets. To express that

character in RTF, use the escape sequence \'*xy*, in which you replace *xy* with the two-digit hexadecimal representation of that character's number. Since 234 in hexadecimal is "ea" (14 * 16 + 10), ê is \'ea in RTF. The Latin phrase "Salvête, Omnês!" is expressed like this:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Monotype Corsiva;}}
\qc\f0\fs120\i\b Salv\'eate,\line Omn\'eas!}
```

Figure 2 shows how it looks in MSWord.

Figure 2. "Hello, World!", the Latin version

A text full of \'*xy* codes can make RTF source unreadable, but RTF was never designed with readability as a goal.

The ASCII character chart in Part IV is a table of characters along with the \'*xy* code that you need to reference each one. For Unicode characters (i.e., characters over 255), you can't use a \'*xy* code, since the codes for those characters don't fit in two hex digits. Instead, there's another sequence for Unicode characters, as explained in the section "Character Formatting."

Now, when you see this chunk of RTF code:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Monotype Corsiva;}}
\qc\f0\fs120\i\b Salv\'eate,\line Omn\'eas!}
```

you may wonder what each bit means. Commands are discussed in detail later, but to give you just a taste of how RTF works, the following is a token-by-token explanation. You don't need to remember any of the codes mentioned here, as they will be properly introduced later.

The {\rtf1 at the start of the file means "the file that starts here will be in RTF, Version 1," and it is required of all RTF documents. (It so happens that there is no RTF Version 0, nor is there likely to be an RTF Version 2 because the current version's syntax is extensible as it is.) The \ansi means that this document is in the ANSI character set (that is, normal Windows Code Page 1252).* Without this declaration, a reader wouldn't know what character set to use in resolving \'*xy* sequences.

The \deff0 says that the default font for the document is font #0 in the font table, which immediately follows. The {\fonttbl...} construct is for listing the names of all the fonts that may be used in the document, associating a number with each. The {\fonttbl...} construct contains a number of {\f*number fontname*;} constructs, for associating a number with a fontname. This ends the prolog to the RTF document; everything afterward is actual text.

The \qc, the first part of the actual text of the document, means that this paragraph should be centered. (It can be inferred that the "c" in \qc is for *center*, but it would surprise me if most users knew that the "q" is for *quadding*, a now rarely-used term for how to justify the paragraph. Normally there are mnemonics for the commands, but they occasionally become obscure.) The \f0 means that we should now use the font that in the font table we associated with the number 0, namely, Monotype Corsiva. The \fs120 means to

* Some old Macintosh word processors wrongly ignore the \ansi declaration and incorrectly resolve character numbers as MacAscii, so that \'ea comes out not as "ê" (character number hex-EA in the ANSI character set), but instead as "á", since that's the character number hex-EA in MacAscii.

change the font size to 60 point. As mentioned earlier, the `\fs` command's parameter is in half-points: so `\fs120` means 60pt, `\fs26` means 13pt, and `\fs15` means 7½pt.

As you probably inferred, `\i` means italic, and `\b` means bold. The space after the `\b` doesn't actually appear in the text, but merely ends the `\b` token.

The literal text `Salv\'eate,` (including the comma) is just "Salvête," with the ê escaped. The command `\line` means to start a new line within the current paragraph, just like a `<BR>` in HTML does. Then we have the literal text `Omn\'eas!`, which is just "Omnês!" escaped. And finally, we end the document with a }. While there are other }'s in the document, we know that it is the one that ends this document because it matches the { that started the document, and because there is nothing after it in this file. Unless an RTF file ends with a } that matches the leading {, it's an error, and errors are treated unpredictably by different RTF-reading applications.

Basic RTF Syntax

So far we've taken an informal view of RTF syntax. But to go any further, we'll need to explain it in more careful terms, considering the kinds of syntactic tokens that exist in the RTF language, and their range of meanings.

RTF breaks down into four basic categories: commands, escapes, groups, and plaintext.

An RTF *command* is like `\pard` or `\fs120`: a backslash, some lowercase letters, maybe an integer (which might have a negative sign before it), and then maybe a single meaningless space character. In terms of regular expressions, a command matches `/\\[a-z]+(-?[0-9]+)? ?/` (including the optional space at the end). An RTF parser knows that a command has ended when it sees a character that no longer matches that pattern. For example, an RTF parser knows that `\i\b` is two commands because the second "\" couldn't possibly be a

Syntax Errors

When an application sees a syntax error in RTF, it might:

- Reject the file entirely; for example, MSWord 2000 aborts when reading malformed RTF files and confusingly reports "The document name or path is not valid."
- Abort reading but leave the document text, up to the error.
- Ignore the error and keep reading; or it might infer that this is not actually an RTF file, and re-read it as a plain-text file.
- Crash.

Over the years, I've seen all these things happen as different applications try reading malformed RTF files. The lesson is that applications aren't forgiving of syntax errors in RTF.

Mercifully, just about the only real syntax error you are likely to make is in not having your {'s and }'s balanced—something that you can find easily with an editor's "balance" feature (on the "%" key in *vi*, or the "meta-(" key sequence in *emacs*). Just about every other kind of error, like misspelling a command name, will typically not be treated as a syntax error to an RTF-reading application.

For example, if you misspell the center-paragraph command `\qc` as `\cq`, the RTF-reading application parses it as a perfectly valid command. The fact that there is no actual "cq" command in RTF means that the application would just ignore it.

continuation of the `\i` command. For another example, in `\pard*`, the * couldn't possibly be a continuation of the `\pard` command, because an asterisk can't be part of a command name, nor could it even be part of the optional integer; and of course it can't be the optional meaningless space.

RTF *escapes* are like commands in that they start with a backslash, but that's where the similarity ends. The character after the backslash is *not* a letter. If the escape is `\'` then the next two characters will be interpreted as hex digits, and the escape is understood to mean the character whose ASCII

code is the given hex number. For example, the escape `\'ea` means the ê character, because character 0xEA in ASCII is ê. But if the character after the `\` isn't an apostrophe, the escape consists of just that one character.

There are only three escapes that are of general interest: `\~` is the escape that indicates a nonbreaking space; `\-` is an optional hyphen (a.k.a. a *hyphenation point*); and `\_` is a nonbreaking hyphen (that is, a hyphen that that's not safe for breaking the line after). The escape `\*` is also part of a construct discussed later. Be sure to note that there is no optional meaningless space after escapes; while `\foo\bar` is the same as `\foo \bar`, `\'ea\'ea` means something different than `\'ea \'ea`. The first one means "êê" (no space) and the second one means "ê ê" (with a space).

An RTF *group* is whatever is between a { and the matching }. For example, `{\i Hi there!}` is a group that contains the command `\i` and the literal text `Hi there!`. Some groups are only necessary for certain constructs (like the `{\fonttbl...}` construct we saw earlier). But most groups have a more concrete purpose: to act as a barrier to the effects of character formatting commands. If you want to italicize the middle word in "a sake cup", use the code `a {\i sake} cup`. In terms of how this is parsed, the { means "save all the character formatting attributes now," and the } means "restore the character formatting attributes to their most recently saved values."

The idea of a group in RTF is analogous to the idea of blocks in "Algol-like" or "block-structured" languages such as C, Perl, Pascal, modern Basic dialects, and so forth; if you are familiar with the idea of "block scope" in such languages, you should be at home with the notion of groups in RTF.

It's tempting to view code like `{\i...}` as just the RTF way to express what HTML or XML express with `<i>...</i>`. This is a useful way to look at it, but it doesn't explain RTF code like `{a \i sake} cup`, which in fact means the same thing as `a {\i sake} cup`.

The final bit of RTF syntax is *plaintext*: the text that is sent right through to the document, character for character. For example, when we had `Hello, World!` in our document, it turned into the text that said simply "Hello, World!".

Newlines and Spaces in RTF

Many computer languages don't distinguish between different kinds of whitespace. So, for example, in the PostScript graphics language, the code `newpath 300 400 100 0 180 arc fill` draws a black half-circle in about the middle of the page. You could write that code compactly on one line:

```
newpath 300 400 100 0 180 arc fill
```

Or in place of the spaces there, you could insert any mix of spaces, newlines, and tabs:

```
newpath
  300  400
  100   0
    180
       arc
 fill
```

Similarly, in Lisp, these two bits of code mean the same thing:

```
(while (search-forward "x" nil t) (replace-match "X" t t))

(while
  (
    search-forward  "x"  nil  t
  )
  (replace-match "X" t t)
)
```

And in HTML, these are the same:

```
<b>Get me a cup of <i>sake</i>, okay?<b>

<b>Get
 me   a     cup
        of <i>sake</i>,
   okay?<b>
```

However, RTF is not that kind of language. In RTF, the rules for dealing with whitespace are very different:

- A space character is meaningless *only* if it's the optional meaningless space at the end of a command. Otherwise, every space character means to insert a space character!
- A newline can mark the end of the command; but otherwise, it has no effect or meaning.

For example, consider `in{\i cred}ible`, in which the space ends the `\i` command, but doesn't actually add a space to the document. In all other cases, a space in the source means a space in the document. For example:

```
Space,     the {\b
final frontier}
```

Here, there are five spaces after the comma, and they really do insert five spaces into the document. Also, the newline after `\b` marks the end of the `\b` command. If that newline were removed, the code would look like `\bfinal frontier`, which the parser would interpret as a command called `\bfinal` (plus a meaningless space that ends the command) and then the word `frontier`.

A newline (or 2 or 15) doesn't indicated a linebreak. For example, this code:

```
in
cred

ible
```

Means the same as this:

```
incredible
```

This may seem a very strange way for a markup language to go about doing things. But RTF isn't really a markup language; it's a data format that happens to be used for expressing text documents. It was never meant to be something that people would find easy and intuitive for typing. Nor is machine-generated RTF typically easy for people to read. However, when you write RTF-generating programs, try to produce easy-to-read source, since it makes programs easier to debug.

Your programs should insert a newline for two reasons: first, in order to make logical divisions in the RTF source; second, to avoid lines that are overly long (like over 250 characters), which become unwieldy to look at in many editors, and occasionally cause trouble when transferred through email. Here are some rules of thumb for putting linebreaks in RTF:

- Put a newline before every `\pard` or `\par` (commands that are explained in the "Paragraphs" section).
- Put a newline before and after the RTF font-table, stylesheet, and other similar constructs (like the optional color table, described later).
- You can put a newline after every *N*th space, {, or }. (Alternately: put a newline after every space, {, or } that's after the 60th column.)

As you're massaging RTF source, consider avoiding having the word "From" occur at the start of a line—either break the line elsewhere, or represent it as `\'46rom`. Email protocols occasionally turn "From" at line start into ">From" even in text-encoded attachments like RTF files. Escaping the "F" keeps that from happening.

Paragraphs

This is the basic construct for a paragraph in RTF:

```
{\pard ... \par}
```

For example, consider this document of 2 plain paragraphs in 12pt Times:

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Times;}}\fs24

{\pard
Urbem Romam a principio reges habuere; libertatem et
consulatum L. Brutus instituit. dictaturae ad tempus
sumebantur; neque decemviralis potestas ultra biennium,
neque tribunorum militum consulare ius diu valuit.
\par}

{\pard
```

```
Non Cinnae, non Sullae longa dominatio; et Pompei Crassique potentia
cito in Caesarem, Lepidi atque Antonii arma in Augustum cessere, qui
cuncta discordiis civilibus fessa nomine principis sub imperium
accepit.
\par}
}
```

Formatted, they look like Figure 3.

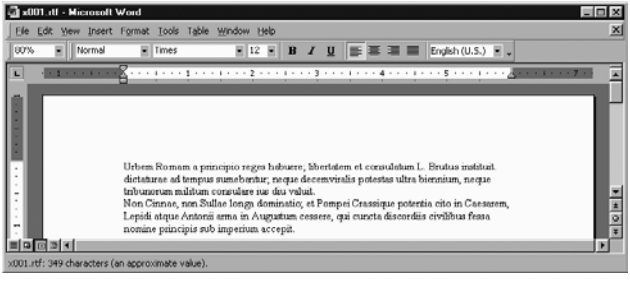


Figure 3. Two plain paragraphs

The RTF idea of "paragraph" is broader than what you're probably used to. We might call whatever HTML implements with a <p> tag a paragraph, but the RTF concept of "paragraph" corresponds to almost everything in HTML that isn't a character-formatting tag such as .... For example, what we would call a heading is implemented in RTF as just a (generally) short paragraph with (generally) large type. For example, consider the heading "Annalium Romae" in the following RTF (which appears formatted in Figure 4):

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Times;}}
\fs24
{\pard \fs44 Annalium Romae\par}
{\pard
Urbem Romam a principio reges habuere; libertatem et
consulatum L. Brutus instituit. dictaturae ad tempus
sumebantur; neque decemviralis potestas ultra biennium,
neque tribunorum militum consulare ius diu valuit.
\par}
{\pard
```

```
Non Cinnae, non Sullae longa dominatio; et Pompei Crassique potentia
cito in Caesarem, Lepidi atque Antonii arma in Augustum cessere, qui
cuncta discordiis civilibus fessa nomine principis sub imperium
accepit.
\par}
}
```

Figure 4. A heading

Right after the `\pard` in each paragraph, you can add commands that control the look of that paragraph. There are several kinds of paragraph-formatting commands, and we will go through them in groups.

A word of warning: if you're at home with very "semantic" markup languages like LaTeX or XML-based document languages, you may be be taken aback by the idea that RTF treats headings as a kind of paragraph—don't headings and paragraphs *mean* different things? RTF takes a different approach: RTF is about how things look. RTF doesn't care about the semantic difference between italics for emphasis (You said *what?*) versus italics for titles (*Naked Lunch*) versus italics for names of ships (*Lusitania*)—to RTF it's all just italics. Of course, if you do want to add a layer of semantic tagging to RTF documents, you can use styles (covered later in this part of the book); but even then, RTF is still about appearances, because RTF styles don't replace appearances, they just add to them.

About {\pard ... \par}

Since {\pard...\par} is such a common construct, it must be explained that its parts do have independent meanings. The \pard means to reset the paragraph-formatting attributes to their default value, instead of inheriting them (or some of them!) from the previous paragraph. The \par means to end the current paragraph. The {...} around the whole construct isn't totally necessary, but it keeps font changes from spilling into subsequent paragraphs. While a \par or a \pard could each exist on their own, and could exist outside of a surrounding {...} group, I have found that this makes for RTF code that is very hard to debug. My experience has been that {\pard...\par} is the sanest and safest way to express paragraphs.

In an ideal world, the construct for paragraphs would be something like {\p...}. Hindsight is 20/20. We have to make do with having a \pard command to start, and a command \par to end. Luckily, command-pairs like that are relatively rare in RTF.

Controlling Paragraph Justification and Centering

The first commands we will cover are ones that tell the word processor how to *justify* the lines on in the current paragraph—namely, whether to make the lines flush on the left margin, or the right, or both, or whether to center each line.

ql

Left-justify this paragraph (leave the right side ragged). This is generally the default.

qr

Right-justify this paragraph (leave the left side ragged). This is rarely used.

\qj

Full-justify this paragraph (try to make both sides smooth).

\qc

Center this paragraph. This is generally used only for headings, not normal paragraphs. Each line of the heading/paragraph is centered.

The following RTF code is an example that demonstrates centering, full justification, and left-justification. It is shown formatted in Figure 5.

```
{\rtf1\ansi\deff0 {\fonttbl {\f0 Times;}}
\fs34
{\pard \qc \fs60 Annalium Romae\par}
{\pard \qj
Urbem Romam a principio reges habuere; libertatem et
consulatum L. Brutus instituit. dictaturae ad tempus
sumebantur; neque decemviralis potestas ultra biennium,
neque tribunorum militum consulare ius diu valuit.
\par}
{\pard \ql
Non Cinnae, non Sullae longa dominatio; et Pompei Crassique potentia
cito in Caesarem, Lepidi atque Antonii arma in Augustum cessere, qui
cuncta discordiis civilibus fessa nomine principis sub imperium
accepit.
\par}
```

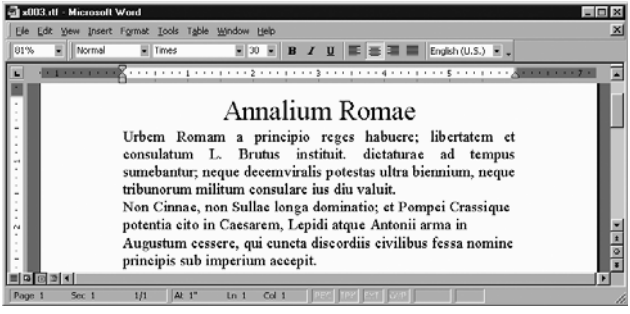


Figure 5. Centering, full justification, and left-justification

Spacing Between Paragraphs

There are two simple commands for adding space over and under the current paragraph: `\sb` and `\sa`, for space before and space after. But take note: they don't measure the space in centimeters or pixels or inches or anything else familiar. Instead, the space is measured in *twips*, a unit that RTF uses for almost everything. A twip is a 1,440th of an inch, or about a 57th of a millimeter. The name *twip* comes from a twentieth of a point (a *point* is a typesetting unit here defined as a 72nd of an inch). Points are rarely used except for expressing the size of a font. To help you measure distances in twips, the section "Converting to Twips" in Part IV shows conversions between twips and inches and centimeters. Also, see the twips ruler inside the back cover of this book.

sbN
: Add *N* twips of (vertical) space before this paragraph. By default, this is 0. For example, `\sb180` adds 180 twips (an eighth of an inch) before this paragraph.

saN
: Add *N* twips of (vertical) space after this paragraph. By default, this is 0. For example, `\sb180` adds 180 twips (an eighth of an inch) after this paragraph.

Generally, the effect you are after is space between paragraphs, and the simplest way to create it is to give every paragraph a `\sa` command. (If you had an `\sb` on each one, then the first paragraph on the first page would have some space before it, and it would look odd.)

For example, taking the RTF source that gave us Figure 4 and changing every paragraph to start out with `{\pard\sa180` gives Figure 6's formatting, which nicely separates the paragraphs visually.

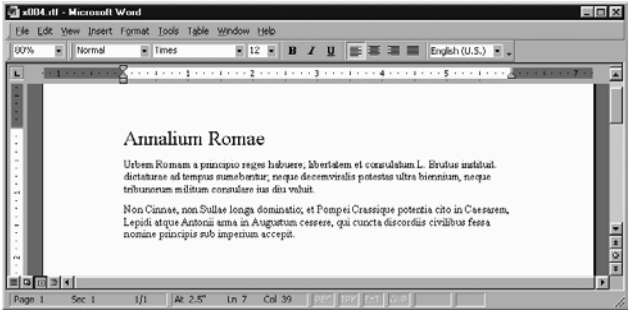


Figure 6. Space added between paragraphs

Paragraph Indenting

There are three paragraph-formatting commands that control *indenting*, in various ways:

fiN

Indent the first line of this paragraph by *N* twips. For example, `\fi720` will indent the first line by 720 twips (a half-inch). This is the common sense of the English word "indent." But you can also use a negative number to "outdent"—i.e., to have the first line start further to the *left* than the rest of the paragraph, as with `\fi-720`.

liN

This command and the following command control *block indenting*, i.e., indenting not just the first line, but the whole paragraph. The `\liN` command expresses how far in from the left margin this paragraph should be block-indented.

riN

This command sets how far in from the right margin this paragraph should be block-indented.

Note that `\fi` doesn't start indenting from the left margin of the page, but from the left margin of the paragraph (which `\li` may have set to something other than the left page-margin).

For example, consider the three paragraphs in Figure 7, the first of which starts out with {\pard \fi720 \qj\sa180, the second with {\pard \fi-1440 \li2800 \qj\sa180, and the third with {\pard \li2160 \ri2160 \qj\sa180.

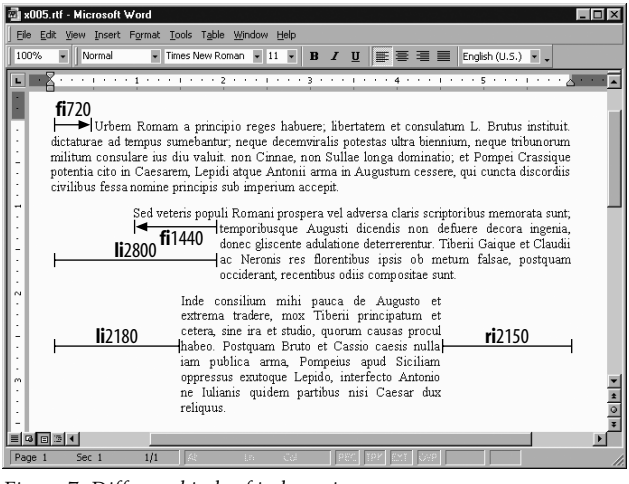


Figure 7. Different kinds of indentation

Bear in mind that the left and right margin do not mean the left and right edge of the page; typically, the margins are an inch in from the edge of the page. For more about changing the margins, see the "Page Margins" section later.

Paragraphs and Pagebreaks

There are seven commands that control how the pagebreaking and linebreaking settings interact with paragraphs:

pagebb
: Make this paragraph start a new page, i.e., put a pagebreak before this paragraph.

keep
: Try to not split this paragraph across pages, i.e., keep it in one piece.

keepn
: Try to avoid a pagebreak between this paragraph and the next—i.e., keep this paragraph with the next one. This command is often used on headings, to keep them together with the following text paragraph.

widctlpar
: Turn on widow-and-orphans control for this paragraph. This is a feature that tries to avoid breaking a paragraph between its first and second lines, or between its second-to-last and last lines, since breaking in either place looks awkward. Since you'd normally want this on for the whole document, not just for a particular paragraph, you probably want to just use a single `\widowctrl` command at the start of the document, as discussed in the "Preliminaries" section.

nowidctlpar
: Turn off widow-and-orphans control for this paragraph. This is useful when `\widowctrl` has turned on widows-and-orphans control for the whole document but you want to disable it for just this paragraph.

hyphpar
: This turns on automatic hyphenation for this paragraph. Since you normally want this on for the whole document, you probably want to just use a single `\hyphauto` at the start of the document, as discussed in "Preliminaries."

hyphpar0
: This command turns off automatic hyphenation for this paragraph. This is a useful when you have a `\hyphauto` set for this document, but you want to exempt a few paragraphs from hyphenation. (Technically, `\hyphpar0` isn't a separate command—it's just a `\hyphpar` command with a parameter value of 0.)

For example, the code below makes a heading that shouldn't be split up from the following paragraph, and then makes

that paragraph, which consists of several lines of verse that shouldn't be broken across pages. Figure 8 shows the result.

```
{\pard \fs60 \keepn \qc .IX.\par}
{\pard \keep
Nullus in urbe fuit tota qui tangere vellet\line
  uxorem gratis, Caeciliane, tuam,\line
dum licuit: sed nunc positis custodibus ingens\line
  turba fututorum est: ingeniosus homo es.
\par}
```

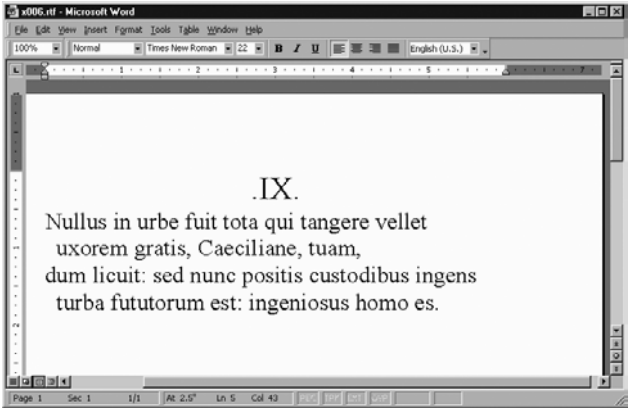


Figure 8. A centered heading and a paragraph with linebreaks

Double-Spacing

To double-space a paragraph, put the code `\sl480\slmult1` right after the `\pard`. To triple-space it, use `\sl720\slmult1`. To have just 1.5-spacing, use `\sl360\slmult1`. A single-spaced paragraph is the default, and doesn't need any particular code. (The magic numbers 480, 720, and 360 don't depend on the point size of the text in the paragraph.)

You might think it's a limitation that line-spacing is an attribute of paragraphs. For example, in WordPerfect, the internal code for line-spacing can be set anywhere in any paragraph—it takes effect starting on that line, and can last

for the rest of the document. Whereas with RTF's way of doing things, you can only set line-spacing (and many other features) for a paragraph, and the settings don't automatically apply to the following paragraphs. Why does RTF do it that way? Basically, because MSWord does it that way, and the designers of RTF tended to model it after the internal format of MSWord documents.

Exact Paragraph Positioning

Paragraphs are usually placed below the previous one and against the left margin. However, in some cases, such as when printing labels, you need to print text at a specific spot on the page. In that case, use the `\pvpg\phpg\posx`*X*`\posy`*Y*`\absw`*W* construct to place the start of the paragraph *X* twips across and *Y* twips down from the top left of the page, with a width of *W* twips. That construct goes after the `\pard` at the start of a paragraph.

For example, the following paragraph is positioned with the top-left tip of its first letter (the "U" in "Urbem") 2,160 twips left and 3,600 twips down from the top-left corner of the page. The paragraph will be 4,320 twips wide:

```
{\pard \pvpg\phpg \posx2160 \posy3600 \absw4320
Urbem Romam a principio reges habuere; libertatem et
 consulatum L. Brutus instituit. dictaturae ad tempus
 sumebantur; neque decemviralis potestas ultra biennium,
\par}
```

You can put a border around this paragraph by adding a rather large block of commands after the `\pard`:

```
\brdrt \brdrs \brdrw10 \brsp20
\brdrl \brdrs \brdrw10 \brsp80
\brdrb \brdrs \brdrw10 \brsp20
\brdrr \brdrs \brdrw10 \brsp80
```

Normally, the paragraph (and any border around it) extends just as far down as the paragraph needs in order to show all its lines. However, you can add a `\absh`*MinHeight* command after the `\absw`, to force the paragraph to be at least *MinHeight* twips high; the word processor will format this by adding

space to the bottom of the paragraph if it would otherwise come out at shorter than *MinHeight* twips high. If you have borders on this paragraph, any added space will be between the bottom of the paragraph and the bottom border (instead of just being under the bottom border).

Or you can use the \absh-*ExactHeight* command to set the exact height of the paragraph (and any borders). For example, the following paragraph will be inside a box (created by the border lines) 5,760 twips high:

```
{\pard \pvpg\phpg \posx2160 \posy3600 \absw4320
\absh-5760
\brdrt \brdrs \brdrw10 \brsp20
\brdrl \brdrs \brdrw10 \brsp80
\brdrb \brdrs \brdrw10 \brsp20
\brdrr \brdrs \brdrw10 \brsp80
Urbem Romam a principio reges habuere; libertatem et
 consulatum L. Brutus instituit. dictaturae ad tempus
 sumebantur; neque decemviralis potestas ultra biennium,
\par}
```

If the current font and point size makes the text take up only a part of the space in that box, then there is just blank space left at the bottom. But if the text is too large to fit in that box, the extra text is hidden. That is, only as much of the paragraph is shown as actually fits in that box, because of the \absh-*ExactHeight* command.

The only syntactic difference between the \absh*MinHeight* command and the \absh-*ExactHeight* command is the negative sign. This is indeed an unusual use of the negative sign, since it uses *-N* to mean something very different from *N*, whereas you would expect *-N* to mean simple "*N*, but in the other direction."

The full set of commands for exact positioning are explained in the "Positioned Objects and Frames" section of the RTF specification. The border commands are explained in the "Paragraph Borders" section of the RTF specification. We will also see a very similar construct for table cell borders in the "Preliminaries" section of this book.