Network Working Group
Request for Comments: 1896
Obsoletes: 1523, 1563
Category: Informational

P. Resnick
QUALCOMM
A. Walker
InterCon

February 1996

The text/enriched MIME Content-type

Status of this Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Abstract

MIME [RFC-1521] defines a format and general framework for the representation of a wide variety of data types in Internet mail. This document defines one particular type of MIME data, the text/enriched MIME type. The text/enriched MIME type is intended to facilitate the wider interoperation of simple enriched text across a wide variety of hardware and software platforms. This document is only a minor revision to the text/enriched MIME type that was first described in [RFC-1523] and [RFC-1563], and is only intended to be used in the short term until other MIME types for text formatting in Internet mail are developed and deployed.

The text/enriched MIME type

In order to promote the wider interoperability of simple formatted text, this document defines an extremely simple subtype of the MIME content-type "text", the "text/enriched" subtype. The content-type line for this type may have one optional parameter, the "charset" parameter, with the same values permitted for the "text/plain" MIME content-type.

The text/enriched subtype was designed to meet the following criteria:

1. The syntax must be extremely simple to parse, so that even teletype-oriented mail systems can easily strip away the formatting information and leave only the readable text.

2. The syntax must be extensible to allow for new formatting commands that are deemed essential for some application.

3. If the character set in use is ASCII or an 8-bit ASCII superset, then the raw form of the data must be readable enough to be largely unobjectionable in the event that it is displayed on the screen of the user of a non-MIME-conformant mail reader.

4. The capabilities must be extremely limited, to ensure that it can represent no more than is likely to be representable by the user's primary word processor. While this

limits what can be sent, it increases the likelihood that what is sent can be properly displayed.

There are other text formatting standards which meet some of these criteria. In particular, HTML and SGML have come into widespread use on the Internet. However, there are two important reasons that this document further promotes the use of text/enriched in Internet mail over other such standards:

1. Most MIME-aware Internet mail applications are already able to either properly format text/enriched mail or, at the very least, are able to strip out the formatting commands and display the readable text. The same is not true for HTML or SGML.

2. The current RFC on HTML [RFC-1866] and Internet Drafts on SGML have many features which are not necessary for Internet mail, and are missing a few capabilities that text/enriched already has.

For these reasons, this document is promoting the use of text/enriched until other Internet standards come into more widespread use. For those who will want to use HTML, Appendix B of this document contains a very simple C program that converts text/enriched to HTML 2.0 described in [RFC-1866].

Syntax

The syntax of "text/enriched" is very simple. It represents text in a single character set--US-ASCII by default, although a different character set can be specified by the use of the "charset" parameter. (The semantics of text/enriched in non-ASCII character sets are discussed later in this document.) All characters represent themselves, with the exception of the "<" character (ASCII 60), which is used to mark the beginning of a formatting command. A literal less-than sign ("<") can be represented by a sequence of two such characters, "<<".

Formatting instructions consist of formatting commands surrounded by angle brackets ("<>", ASCII 60 and 62). Each formatting command may be no more than 60 characters in length, all in US-ASCII, restricted to the alphanumeric and hyphen ("-") characters. Formatting commands may be preceded by a solidus ("/", ASCII 47), making them negations, and such negations must always exist to balance the initial opening commands. Thus, if the formatting command "<bold>" appears at some point, there must later be a "</bold>" to balance it. (NOTE: The 60 character limit on formatting commands does NOT include the "<", ">", or "/" characters that might be attached to such commands.) Formatting commands are always case-insensitive. That is, "bold" and "BoLd" are equivalent in effect, if not in good taste.

Line break rules

Line breaks (CRLF pairs in standard network representation) are handled specially. In particular, isolated CRLF pairs are translated into a single SPACE character. Sequences of N consecutive CRLF pairs, however, are translated into N-1 actual line breaks. This permits long lines of data to be represented in a natural looking manner despite the frequency of line-wrapping in Internet mailers. When preparing the data for mail transport, isolated line breaks should be inserted wherever necessary to keep each line shorter than 80 characters. When preparing such data for presentation to the user, isolated line breaks should be replaced by a single SPACE character, and N consecutive CRLF pairs should be presented to the user as N-1 line breaks.

Thus text/enriched data that looks like this:

```
This is
a single
line

This is the
next line.


This is the
next section.
```

should be displayed by a text/enriched interpreter as follows:

```
This is a single line
This is the next line.

This is the next section.
```

The formatting commands, not all of which will be implemented by all implementations, are described in the following sections.

Formatting Commands

The text/enriched formatting commands all begin with <commandname> and end with </commandname>, affecting the formatting of the text between those two tokens. The commands are described here, grouped according to type.

Parameter Command

Some of the formatting commands may require one or more associated parameters. The "param" command is a special formatting command used to include these parameters.

Param

Marks the affected text as command parameters, to be interpreted or ignored by the text/enriched interpreter, but *not* to be shown to the reader. The "param" command always immediately follows some other formatting command, and the parameter data indicates some additional information about the formatting that is to be done. The syntax of the parameter data (whatever appears between the initial "<param>" and the terminating "</param>") is defined for each command that uses it. However, it is always required that the format of such data must not contain nested "param" commands, and either must not use the "<" character or must use it in a way that is compatible with text/enriched parsing. That is, the end of the parameter data should be recognizable with either of two algorithms: simply searching for the first occurrence of "</param>" or parsing until a balanced "</param>" command is found. In either case, however, the parameter data should not be shown to the human reader.

Font-Alteration Commands

The following formatting commands are intended to alter the font in which text is displayed, but not to alter the indentation or justification state of the text:

Bold

causes the affected text to be in a bold font. Nested bold commands have the same effect as a single bold command.

Italic

causes the affected text to be in an italic font. Nested italic commands have the same effect as a single italic command.

Underline

causes the affected text to be underlined. Nested underline commands have the same effect as a single underline command.

Fixed

causes the affected text to be in a fixed width font. Nested fixed commands have the same effect as a single fixed command.

FontFamily

causes the affected text to be displayed in a specified typeface. The "fontfamily" command requires a parameter that is specified by using the "param" command. The parameter data is a case-insensitive string containing the name of a font family. Any currently available font family name (e.g. Times, Palatino, Courier, etc.) may be used. This includes font families defined by commercial type foundries such as Adobe, BitStream, or any other such foundry. Note that implementations should only use the general font family name, not the specific font name (e.g. use "Times", not "TimesRoman" nor "TimesBoldItalic"). When nested, the inner "fontfamily" command takes precedence. Also note that the "fontfamily" command is advisory only; it should not be expected that other implementations will honor the typeface information in this command since the font capabilities of systems vary drastically.

Color

causes the affected text to be displayed in a specified color. The "color" command requires a parameter that is specified by using the "param" command. The parameter data can be one of the following:

red
blue
green
yellow
cyan
magenta
black
white

or an RGB color value in the form:

####,####,####

where '#' is a hexadecimal digit '0' through '9', 'A' through 'F', or 'a' through 'f'. The three 4-digit hexadecimal values are the RGB values for red, green, and blue respectively, where each component is expressed as an unsigned value between 0 (0000) and 65535 (FFFF). The default color for the message is unspecified, though black is a common choice in many environments. When nested, the inner "color" command takes precedence.

Smaller

causes the affected text to be in a smaller font. It is recommended that the font size be changed by two points, but other amounts may be more appropriate in some environments. Nested smaller commands produce ever smaller fonts, to the limits of the implementation's capacity to reasonably display them, after which further smaller commands have no incremental effect.

Bigger

causes the affected text to be in a bigger font. It is recommended that the font size be changed by two points, but other amounts may be more appropriate in some environments. Nested bigger commands produce ever bigger fonts, to the limits of the implementation's capacity to reasonably display them, after which further bigger commands have no incremental effect.

While the "bigger" and "smaller" operators are effectively inverses, it is not recommended, for example, that "<smaller>" be used to end the effect of "<bigger>". This is properly done with "</bigger>".

Since the capabilities of implementations will vary, it is to be expected that some implementations will not be able to act on some of the font-alteration commands. However, an implementation should still display the text to the user in a reasonable fashion. In particular, the lack of capability to display a particular font family, color, or other text attribute does not mean that an implementation should fail to display text.

Fill/Justification/Indentation Commands

Initially, text/enriched text is intended to be displayed fully filled (that is, using the rules specified for replacing CRLF pairs with spaces or removing them as appropriate) with appropriate kerning and letter-tracking, and using the maximum available margins as suits the capabilities of the receiving user agent software.

The following commands alter that state. Each of these commands force a line break before and after the formatting environment if there is not otherwise a line break. For example, if one of these commands occurs anywhere other than the beginning of a line of text as presented, a new line is begun.

Center
causes the affected text to be centered.

FlushLeft
causes the affected text to be left-justified with a ragged right margin.

FlushRight
causes the affected text to be right-justified with a ragged left margin.

FlushBoth
causes the affected text to be filled and padded so as to create smooth left and right margins, i.e., to be fully justified.

ParaIndent
causes the running margins of the affected text to be moved in. The recommended indentation change is the width of four characters, but this may differ among implementations. The "paraindent" command requires a parameter that is

specified by using the "param" command. The parameter data is a comma-seperated list of one or more of the following:

Left

causes the running left margin to be moved to the right.

Right

causes the running right margin to be moved to the left.

In

causes the first line of the affected paragraph to be indented in addition to the running margin. The remaining lines remain flush to the running margin.

Out

causes all lines except for the first line of the affected paragraph to be indented in addition to the running margin. The first line remains flush to the running margin.

Nofill

causes the affected text to be displayed without filling. That is, the text is displayed *without* using the rules for replacing CRLF pairs with spaces or removing consecutive sequences of CRLF pairs. However, the current state of the margins and justification is honored; any indentation or justification commands are still applied to the text within the scope of the "nofill".

The "center", "flushleft", "flushright", and "flushboth" commands are mutually exclusive, and, when nested, the inner command takes precedence.

The "nofill" command is mutually exclusive with the "in" and "out" parameters of the "paraindent" command; when they occur in the same scope, their behavior is undefined.

The parameter data for the "paraindent" command may contain multiple occurances of the same parameter (i.e. "left", "right", "in", or "out"). Each occurance causes the text to be further indented in the manner indicated by that parameter. Nested "paraindent" commands cause the affected text to be further indented according to the parameters. Note that the "in" and "out" parameters for "paraindent" are mutually exclusive; when they appear together or when nested "paraindent" commands contain both of them, their behavior is undefined.

For purposes of the "in" and "out" parameters, a paragraph is defined as text that is delimited by line breaks *after* applying the rules for replacing CRLF pairs with spaces or removing consecutive sequences of CRLF pairs. For example, within the scope of an "out", the line following each CRLF is made flush with the running margin, and subsequent lines are indented.

Within the scope of an "in", the first line following each CRLF is indented, and subsequent lines remain flush to the running margin.

Whether or not text is justified by default (that is, whether the default environment is "flushleft", "flushright", or "flushboth") is unspecified, and depends on the preferences of the user, the capabilities of the local software and hardware, and the nature of the character set in use. On systems where full justification is considered undesirable, the "flushboth" environment may be identical to the default environment. Note that full justification should never be performed inside of "center", "flushleft", "flushright", or "nofill" environments. Note also that for some non-ASCII character sets, full justification may be fundamentally inappropriate.

Note that [RFC-1563] defined two additional indentation commands, "Indent" and "IndentRight". These commands did not force a line break, and therefore their behavior was unpredictable since they depended on the margins and character sizes that a particular implementation used. Therefore, their use is deprecated and they should be ignored just as other unrecognized commands.

Markup Commands

Commands in this section, unlike the other text/enriched commands are declarative markup commands. Text/enriched is not intended as a full markup language, but instead as a simple way to represent common formatting commands. Therefore, markup commands are purposely kept to a minimum. It is only because each was deemed so prevalent or necessary in an e-mail environment that these particular commands have been included at all.

Excerpt

causes the affected text to be interpreted as a textual excerpt from another source, probably a message being responded to. Typically this will be displayed using indentation and an alternate font, or by indenting lines and preceding them with "> ", but such decisions are up to the implementation. Note that as with the justification commands, the excerpt command implicitly begins and ends with a line break if one is not already there. Nested "excerpt" commands are acceptable and should be interpreted as meaning that the excerpted text was excerpted from yet another source. Again, this can be displayed using additional indentation, different colors, etc.

Optionally, the "excerpt" command can take a parameter by using the "param" command. The format of the data is unspecified, but it is intended to uniquely identify the text from which the excerpt is taken. With this information, an implementation should be able to uniquely identify the source of any particular excerpt, especially if two or more excerpts in the message are from the same source, and display it in some way that makes this apparent to the user.

Lang

causes the affected text to be interpreted as belonging to a particular language. This is most useful when two different languages use the same character set, but may require a different font or formatting depending on the language. For instance, Chinese and Japanese share similar character glyphs, and in some character sets like UNICODE share common code points, but it is considered very important that different fonts be used for the two languages, especially if they appear together, so that meaning is not lost. Also, language information can be used to allow for fancier text handling, like spell checking or hyphenation.

The "lang" command requires a parameter using the "param" command. The parameter data can be any of the language tags specified in [RFC-1766], "Tags for the Identification of Languages". These tags are the two letter language codes taken from [ISO-639] or can be other language codes that are registered according to the instructions in the Langauge Tags RFC. Consult that memo for further information.

Balancing and Nesting of Formatting Commands

Pairs of formatting commands must be properly balanced and nested. Thus, a proper way to describe text in ***bold italics*** is:

```
<bold><italic>the-text</italic></bold>
```

or, alternately,

```
<italic><bold>the-text</bold></italic>
```

but, in particular, the following is illegal text/enriched:

```
<bold><italic>the-text</bold></italic>
```

The nesting requirement for formatting commands imposes a slightly higher burden upon the composers of text/enriched bodies, but potentially simplifies text/enriched displayers by allowing them to be stack-based. The main goal of text/enriched is to be simple enough to make multifont, formatted email widely readable, so that those with the capability of sending it will be able to do so with confidence. Thus slightly increased complexity in the composing software was deemed a reasonable tradeoff for simplified reading software. Nonetheless, implementors of text/enriched readers are encouraged to follow the general Internet guidelines of being conservative in what you send and liberal in what you accept. Those implementations that can do so are encouraged to deal reasonably with improperly nested text/enriched data.

Unrecognized formatting commands

Implementations must regard any unrecognized formatting command as "no-op" commands, that is, as commands having no effect, thus facilitating future extensions to "text/enriched". Private extensions may be defined using formatting commands that begin with "X-", by analogy to Internet mail header field names.

In order to formally define extended commands, a new Internet document should be published.

White Space in Text/enriched Data

No special behavior is required for the SPACE or TAB (HT) character. It is recommended, however, that, at least when fixed-width fonts are in use, the common semantics of the TAB (HT) character should be observed, namely that it moves to the next column position that is a multiple of 8. (In other words, if a TAB (HT) occurs in column n, where the leftmost column is column 0, then that TAB (HT) should be replaced by 8-(n mod 8) SPACE characters.) It should also be noted that some mail gateways are notorious for losing (or, less commonly, adding) white space at the end of lines, so reliance on SPACE or TAB characters at the end of a line is not recommended.

Initial State of a text/enriched interpreter

Text/enriched is assumed to begin with filled text in a variable-width font in a normal typeface and a size that is average for the current display and user. The left and right margins are assumed to be maximal, that is, at the leftmost and rightmost acceptable positions.

Non-ASCII character sets

One of the great benefits of MIME is the ability to use different varieties of non-ASCII text in messages. To use non-ASCII text in a message, normally a charset parameter is specified in the Content-type line that indicates the character set being used. For purposes of this RFC, any legal MIME charset parameter can be used with the text/enriched Content-type. However, there are two difficulties that arise with regard to the text/enriched Content-type when non-ASCII text is desired. The first problem involves difficulties that occur when the user wishes to create text which would normally require multiple non-ASCII character sets in the same text/enriched message. The second problem is an ambiguity that arises because of the text/enriched use of the "<" character in formatting commands.

Using multiple non-ASCII character sets

Normally, if a user wishes to produce text which contains characters from entirely different character sets within the same MIME message (for example, using Russian Cyrillic characters from ISO 8859-5 and Hebrew characters from ISO 8859-8), a multipart message is used. Every time a new character set is desired, a new MIME body part is started with different character sets specified in the charset parameter of the Content-type line. However, using multiple character sets this way in text/enriched messages introduces problems. Since a change in the charset parameter requires a new part, text/enriched formatting commands used in the first part would not be able to apply to text that occurs in subsequent parts. It is not possible for text/enriched formatting commands to apply across MIME body part boundaries.

[RFC-1341] attempted to get around this problem in the now obsolete text/richtext format by introducing different character set formatting commands like "iso-8859-5" and "us-ascii". But this, or even a more general solution along the same lines, is still undesirable: It is common for a MIME application to decide, for example, what character font resources or character lookup tables it will require based on the information provided by the charset parameter of the Content-type line, before it even begins to interpret or display the data in that body part. By allowing the text/enriched interpreter to subsequently change the character set, perhaps to one completely different from the charset specified in the Content-type line (with potentially much different resource requirements), too much burden would be placed on the text/enriched interpreter itself.

Therefore, if multiple types of non-ASCII characters are desired in a text/enriched document, one of the following two methods must be used:

1. For cases where the different types of non-ASCII text can be limited to their own paragraphs with distinct formatting, a multipart message can be used with each part having a Content-Type of text/enriched and a different charset parameter. The one caveat to using this method is that each new part must start in the initial state for a text/enriched document. That means that all of the text/enriched commands in the preceding part must be properly balanced with ending commands before the next text/enriched part begins. Also, each text/enriched part must begin a new paragraph.

2. If different types of non-ASCII text are to appear in the same line or paragraph, or if text/enriched formatting (e.g. margins, typeface, justification) is required across several different types of non-ASCII text, a single text/enriched body part should be used with a character set specified that contains all of the required characters. For example, a charset parameter of "UNICODE-1-1-UTF-7" as specified in [RFC-1642] could be used for such purposes. Not only does UNICODE contain all of the characters that can be represented in all of the other registered ISO 8859 MIME character sets, but UTF-7 is fully compatible with other aspects of the text/enriched standard, including the use of the "<" character referred to below. Any other character sets that are specified for use in MIME which contain different types of non-ASCII text can also be used in these instances.

Use of the "<" character in formatting commands

If the character set specified by the charset parameter on the Content-type line is anything other than "US-ASCII", this means that the text being described by text/enriched formatting commands is in a non-ASCII character set. However, the commands themselves are still the same ASCII commands that are defined in this document. This creates an ambiguity only with reference to the "<" character, the octet with numeric value 60. In single byte character sets, such as the ISO-8859 family, this is not a problem; the octet 60 can be quoted by including it twice, just as for ASCII. The problem is more complicated, however, in the case of multi-byte character sets, where the octet 60 might appear at any point in the byte sequence for any of several characters.

In practice, however, most multi-byte character sets address this problem internally. For example, the UNICODE character sets can use the UTF-7 encoding which preserves all of the important ASCII characters in their single byte form. The ISO-2022 family of character sets can use certain character sequences to switch back into ASCII at any moment. Therefore it is specified that, before text/enriched formatting commands, the prevailing character set should be "switched back" into ASCII, and that only those characters which would be interpreted as "<" in plain text should be interpreted as token delimiters in text/enriched.

The question of what to do for hypothetical future character sets that do not subsume ASCII is not addressed in this memo.

Minimal text/enriched conformance

A minimal text/enriched implementation is one that converts "<<" to "<", removes everything between a <param> command and the next balancing </param> command, removes all other formatting commands (all text enclosed in angle brackets), and, outside of <nofill> environments, converts any series of n CRLFs to n-1 CRLFs, and converts any lone CRLF pairs to SPACE.

Notes for Implementors

It is recognized that implementors of future mail systems will want rich text functionality far beyond that currently defined for text/enriched. The intent of text/enriched is to provide a common format for expressing that functionality in a form in which much of it, at least, will be understood by interoperating software. Thus, in particular, software with a richer notion of formatted text than text/enriched can still use text/enriched as its basic representation, but can extend it with new formatting commands and by hiding information specific to that software system in text/enriched <param> constructs. As such systems evolve, it is expected that the definition of text/enriched will be further refined by future published specifications, but text/enriched as defined here provides a platform on which evolutionary refinements can be based.

An expected common way that sophisticated mail programs will generate text/enriched data is as part of a multipart/alternative construct. For example, a mail agent that can generate enriched mail in ODA format can generate that mail in a more widely interoperable form by generating both text/enriched and ODA versions of the same data, e.g.:

```
Content-type: multipart/alternative; boundary=foo

--foo
Content-type: text/enriched

[text/enriched version of data]
--foo Content-type: application/oda

[ODA version of data]
--foo--
```

If such a message is read using a MIME-conformant mail reader that understands ODA, the ODA version will be displayed; otherwise, the text/enriched version will be shown.

In some environments, it might be impossible to combine certain text/enriched formatting commands, whereas in others they might be combined easily. For example, the combination of <bold> and <italic> might produce ***bold italics*** on systems that support such fonts, but there exist systems that can make text bold or italicized, but not both. In such cases, the most recently issued (innermost) recognized formatting command should be preferred.

One of the major goals in the design of text/enriched was to make it so simple that even text-only mailers will implement enriched-to- plain-text translators, thus increasing the likelihood that enriched text will become "safe" to use very widely. To demonstrate this simplicity, an extremely simple C program that converts text/enriched input into plain text output is included in Appendix A.

Extensions to text/enriched

It is expected that various mail system authors will desire extensions to text/enriched. The simple syntax of text/enriched, and the specification that unrecognized formatting commands should simply be ignored, are intended to promote such extensions.

An Example

Putting all this together, the following "text/enriched" body fragment:

```
From: Nathaniel Borenstein <nsb@bellcore.com>
To: Ned Freed <ned@innosoft.com>
```

```
Content-type: text/enriched

<bold>Now</bold> is the time for <italic>all</italic>
good men
<smaller>(and <<women>)</smaller> to
<ignoreme>come</ignoreme>

to the aid of their


<color><param>red</param>beloved</color>
country.

By the way,
I think that <paraindent><param>left</param><<smaller>

</paraindent>should REALLY be called

<paraindent><param>left</param><<tinier></paraindent>
and that I am always right.

-- the end
```

represents the following formatted text (which will, no doubt, look somewhat cryptic in the text-only version of this document):

Now is the time for *all* good men (and <women>) to come
to the aid of their

beloved country.
By the way, I think that
    <smaller>
should REALLY be called
    <tinier>
and that I am always right.
-- the end

where the word "beloved" would be in red on a color display.

Security Considerations

Security issues are not discussed in this memo, as the mechanism raises no security issues.

Author's Address

For more information, the authors of this document may be contacted via Internet mail:

Peter W. Resnick
QUALCOMM Incorporated
6455 Lusk Boulevard
San Diego, CA 92121-2779
Phone: +1 619 587 1121
FAX: +1 619 658 2230
e-mail: presnick@qualcomm.com

Amanda Walker
InterCon Systems Corporation
950 Herndon Parkway
Herndon, VA 22070
Phone: +1 703 709 5500
FAX: +1 703 709 5555
e-mail: amanda@intercon.com

Acknowledgements

The authors gratefully acknowledge the input of many contributors, readers, and implementors of the specification in this document. Particular thanks are due to Nathaniel Borenstein, the original author of RFC 1563.

References

[RFC-1341]
Borenstein, N., Freed, N., "MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies", 06/11/1992.

[RFC-1521]
Borenstein, N., Freed, N., "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", 09/23/1993.

[RFC-1523]
Borenstein, N., "The text/enriched MIME Content-type", 09/23/1993.

[RFC-1563]
Borenstein, N., "The text/enriched MIME Content-type", 01/10/1994.

[RFC-1642]
Goldsmith, D., Davis, M., "UTF-7 - A Mail-Safe Transformation Format of Unicode", 07/13/1994.

[RFC-1766]
Alvestrand, H., "Tags for the Identification of Languages", 03/02/1995.

[RFC-1866]
Berners-Lee, T., Connolly, D., "Hypertext Markup Language - 2.0", 11/03/1995.

Appendix A--A Simple enriched-to-plain Translator in C

One of the major goals in the design of the text/enriched subtype of the text Content-Type is to make formatted text so simple that even text-only mailers will implement enriched-to-plain-text translators, thus increasing the likelihood that multifont text will become "safe" to use very widely. To demonstrate this simplicity, what follows is a simple C program that converts text/enriched input into plain text output. Note that the local newline convention (the single character represented by "\n") is assumed by this program, but that special CRLF handling might be necessary on some systems.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main() {
        int c, i, paramct=0, newlinect=0, nofill=0;
        char token[62], *p;

        while ((c=getc(stdin)) != EOF) {
                if (c == '<') {
                        if (newlinect == 1) putc(' ', stdout);
                        newlinect = 0;
                        c = getc(stdin);
                        if (c == '<') {
                                if (paramct <= 0) putc(c, stdout);
                        } else {
                                 ungetc(c, stdin);
                                 for (i=0, p=token; (c=getc(stdin)) != EOF && c != '>'; i++) {
                                        if (i < sizeof(token)-1)
                                                *p++ = isupper(c) ? tolower(c) : c;
                                 }
                                 *p = '\0';
                                 if (c == EOF) break;
                                 if (strcmp(token, "param") == 0)
                                         paramct++;
                                 else if (strcmp(token, "nofill") == 0)
                                         nofill++;
                                 else if (strcmp(token, "/param") == 0)
                                         paramct--;
                                 else if (strcmp(token, "/nofill") == 0)
                                         nofill--;
                         }
                } else {
                        if (paramct > 0)
                                ; /* ignore params */
                        else if (c == '\n' && nofill <= 0) {
                                if (++newlinect > 1) putc(c, stdout);
                        } else {
                                if (newlinect == 1) putc(' ', stdout);
                                newlinect = 0;
                                putc(c, stdout);
                        }
                }
        }
        /* The following line is only needed with line-buffering */
        putc('\n', stdout);
```

```
        exit(0);
}
```

It should be noted that one can do considerably better than this in displaying text/enriched data on a dumb terminal. In particular, one can replace font information such as "bold" with textual emphasis (like *this* or _T_H_I_S_). One can also properly handle the text/enriched formatting commands regarding indentation, justification, and others. However, the above program is all that is necessary in order to present text/enriched on a dumb terminal without showing the user any formatting artifacts.

Appendix B--A Simple enriched-to-HTML Translator in C

It is fully expected that other text formatting standards like HTML and SGML will supplant text/enriched in Internet mail. It is also likely that as this happens, recipients of text/enriched mail will wish to view such mail with an HTML viewer. To this end, the following is a simple example of a C program to convert text/enriched to HTML. Since the current version of HTML at the time of this document's publication is HTML 2.0 defined in [RFC-1866], this program converts to that standard. There are several text/enriched commands that have no HTML 2.0 equivalent. In those cases, this program simply puts those commands into processing instructions; that is, surrounded by "<?" and ">". As in Appendix A, the local newline convention (the single character represented by "\n") is assumed by this program, but special CRLF handling might be necessary on some systems.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main() {
        int c, i, paramct=0, nofill=0;
        char token[62], *p;

        while((c=getc(stdin)) != EOF) {
                if(c == '<') {
                        c = getc(stdin);
                        if(c == '<') {
                                fputs("&lt;", stdout);
                        } else {
                                ungetc(c, stdin);
                                for (i=0, p=token; (c=getc(stdin)) != EOF && c != '>'; i++) {
                                        if (i < sizeof(token)-1)
                                                *p++ = isupper(c) ? tolower(c) : c;
                                }
                                *p = '\0';
                                if(c == EOF) break;
                                if(strcmp(token, "/param") == 0) {
                                        paramct--;
                                        putc('>', stdout);
                                } else if(paramct > 0) {
                                        fputs("&lt;", stdout);
                                        fputs(token, stdout);
                                        fputs("&gt;", stdout);
                                } else {
                                        putc('<', stdout);
                                        if(strcmp(token, "nofill") == 0) {
                                                nofill++;
                                                fputs("pre", stdout);
                                        } else if(strcmp(token, "/nofill") == 0) {
                                                nofill--;
                                                fputs("/pre", stdout);
```

```
                                        } else if(strcmp(token, "bold") == 0) {
                                                fputs("b", stdout);
                                        } else if(strcmp(token, "/bold") == 0) {
                                                fputs("/b", stdout);
                                        } else if(strcmp(token, "italic") == 0) {
                                                fputs("i", stdout);
                                        } else if(strcmp(token, "/italic") == 0) {
                                                fputs("/i", stdout);
                                        } else if(strcmp(token, "fixed") == 0) {
                                                fputs("tt", stdout);
                                        } else if(strcmp(token, "/fixed") == 0) {
                                                fputs("/tt", stdout);
                                        } else if(strcmp(token, "excerpt") == 0) {
                                                fputs("blockquote", stdout);
                                        } else if(strcmp(token, "/excerpt") == 0) {
                                                fputs("/blockquote", stdout);
                                        } else {
                                                putc('?', stdout);
                                                fputs(token, stdout);
                                                if(strcmp(token, "param") == 0) {
                                                        paramct++;
                                                        putc(' ', stdout);
                                                        continue;
                                                }
                                        }
                                        putc('>', stdout);
                                }
                        }
                } else if(c == '>') {
                        fputs("&gt;", stdout);
                } else if (c == '&') {
                        fputs("&amp;", stdout);
                } else {
                        if(c == '\n' && nofill <= 0 && paramct <= 0) {
                                while((i=getc(stdin)) == '\n') fputs("<br>", stdout);
                                ungetc(i, stdin);
                        }
                        putc(c, stdout);
                }
        }
        /* The following line is only needed with line-buffering */
        putc('\n', stdout);
        exit(0);
}
```