MIME Parser API

Manual version $Revision: 1.7 $
$Date: 1995/03/16 01:55:41 $

Bob Glickstein

This document describes the Z-Mail MIME Parser API. It is a PROPRIETARY and CONFIDENTIAL DRAFT.

Introduction

This is a PROPRIETARY and CONFIDENTIAL DRAFT. Comments and critique are urgently requested.

This API is not actually a parser at all. Instead, it is a family of functions with which any of several kinds of MIME parser can be built. Some of the job of parsing MIME messages is left to the caller of these functions.

A full-fledged parser is not sufficiently general-purpose. The caller may need to exert control over the parsing process at many different points; for instance, to skip a multipart subpart without parsing it, or to record the source stream's seek position when a particular subpart is found. Without a bewildering maze of callbacks and special data types, the best way to achieve this is with a flattened, piecewise approach to parsing. Each function is meant to perform all of the processing up to a decision point; for instance, reaching the end of a block of headers, where the caller may then decide to parse the following body, or to skip it. There is no recursive parsing of nested bodyparts. Nesting of subparts is not reflected by recursion in the parse routines, but in an external data structure which tracks the nesting structure.

1 Functions

`const char * mime_Readline` (*struct dpipe* *`dp`, *struct dynstr* *`d`) [Function]

Read bytes from *dp*, appending them to *d*, until end-of-file or until a newline is encountered. The newline is not appended to *d*. A `NUL`-terminated version of the newline sequence recognized is returned. For convenience, the three possible newline sequences are available in global string variables: `mime_CR`, `mime_CRLF`, and `mime_LF`. If end-of-file is encountered before a newline, the return value is 0. If *d* is 0, the line is read and discarded.

For purposes of the functions in this document, the definition of *newline* is any of the three major line-terminating conventions: the canonical line terminator, `CR`-`LF`, is accepted as a single newline, as are `CR` and `LF` by themselves.

`void mime_Header` (*struct dpipe* *`dp`, *struct dynstr* *`name`, *struct dynstr* *`value`, *const char* *`newline`) [Function]

Given a readable stream *dp* which is presumed to be positioned at the beginning of an RFC822 header, append the name of the header to *name* and the content of the header to *value*. Each of *name* and *value* may be 0 to indicate that the corresponding data should be discarded after being read.

If the header spans multiple lines (using RFC822's newline-whitespace continuation syntax), all lines are appended to *value* complete with their terminating newline sequences. If *newline* is non-zero, then it points to a `NUL`-terminated string to which all newline sequences encountered will be converted. Typically, *newline* will contain one of the values `mime_CR`, `mime_LF`, or `mime_CRLF`, for canonicalization.

If end-of-file or an illegal character is found in the header name, this function raises the exception `mime_err_Header`.

`int mime_Headers` (*struct dpipe* *`dp`, *struct glist* *`hlist`, *const char* *`newline`) [Function]

Given a readable stream *dp* which is presumed to be positioned at the beginning of a (possible empty) block of RFC822 headers, parse each header using `mime_Header`, appending the name/value pairs to *hlist*. The data type of the records in *hlist* is `struct mime_pair`, which is defined as follows:

```
struct mime_pair {
    struct dynstr name, value;
};
```

Returns the number of headers read. The stream is left positioned after the terminating newline of the last header, meaning that if the header block is terminated as usual, by newline-newline, the first newline will have been read and the second newline will be waiting to be read. However, parsing of headers will stop at any line that does not look like a header; for instance, one beginning with a "special" character like '`@`'.

Each header is read with `mime_Header`, and the parameter *newline* is as in that function.

`void mime_ContinueHeader` (*struct dpipe* *`dp`, *struct dynstr* *`value`, *const char* *`newline`) [Function]

Read zero or more header continuation lines from *dp*, appending them to *value*, transforming embedded newlines according to *newline*, which is as in `mime_Header`.

This function is useful if one has already read a line, e.g. with `mime_Readline`, discovered it to be the first line of a message header, and wishes to slurp up any remaining lines. Lines that begin with whitespace are appended to *value*, up until the first line that does not begin with whitespace, which is where *dp* is left positioned after this call.

`void mime_Unfold` (*struct dynstr* *`value`, *int* `collapse`) [Function]

Given a header value *value*, rewrite it in place to perform RFC822 *unfolding*. Unfolding is the process of removing embedded newlines in multi-line header values. If *collapse* is non-zero, also collapse all whitespace characters following each newline into a single space.

`char * mime_NextToken` (*char* *`str`, *char* **`end`, *int* `tspecial`) [Function]

Given a string *str*, presumably the content of a mail header, return the next *token*, and place in `*end` the position in *str* of the first byte following the token. The tokenizing rules are selected by *tspecial*; if 0, RFC822 "special" characters are delimiters; otherwise, MIME "tspecial" characters are delimiters. MIME "tspecials" are for tokenizing the MIME Content-Type header.

The return value is stored in an internal buffer which is overwritten with each call. The token is either an RFC822 "atom," a "special" or "tspecial" character as appropriate, or a "quoted-string."

If the token is a quoted string, the delimiting quotation marks are removed and backslash-escapes in the quoted-string are decoded. Tokenizing skips whitespace and comments.

If the token is a "special" or "tspecial," the global `int` variable `mime_SpecialToken` is set to its value; if not, that variable is set to 0. This allows one to distinguish between a "special" character in the header, and a quoted string containing a special character. For example, if the tokenizer encounters '`:`' in the input, then the return value will be the string `":"` and `mime_SpecialToken` will be set to 58; but if the tokenizer encounters '`":"`' in the input, then the return value will still be the string `":"` but `mime_SpecialToken` will be 0.

The RFC822 "special" characters are:

> '`(`', '`)`', '`<`', '`>`', '`@`', '`,`', '`;`',
> '`:`', '`\`', '`"`', '`.`', '`[`', '`]`'.

The MIME "tspecial" characters are:

> '`(`', '`)`', '`<`', '`>`', '`@`', '`,`', '`;`',
> '`:`', '`\`', '`"`', '`/`', '`[`', '`]`', '`?`',
> '`=`'.

Return value is 0 when no token follows in *str*.

If a quoted-string is unterminated or contains an illegal character, this function raises the exception `mime_err_String`. If a comment is unterminated or contains an illegal character, this function raises the exception `mime_err_Comment`.

`void mime_MultipartStart` (*struct glist* *`stack`, *const char* *`boundary`, *void* (*`cleanup`)(*void* *), *void* *`cleanup_data`) [Function]

This function must be called in order to parse the subparts of a multipart. The caller presumably has read a block of headers using `mime_Headers` and has discovered (possibly using `mime_ParseContentType` or `mime_AnalyzeHeaders`) a Content-Type header indicating a multipart type and a boundary string. No recursive call is made to parse the subparts of a multipart, instead, *stack* is used to keep track of multipart nesting.

The elements of *stack* are of type `struct mime_stackelt`. A new element is placed on *stack* indicating the pending boundary string, *boundary*, which is the value of the `boundary` parameter in the `Content-Type` header. The function *cleanup* will be called when the new stack frame is unwound (usually by encountering the end of this multipart, but see `mime_NextBoundary`). When *cleanup* is called, it is passed *cleanup_data* as an argument.

`int mime_NextBoundary` (*struct dpipe* *`dp`, *struct dpipe* *`dest`, *struct glist* *`stack`, *const char* *`newline`) [Function]

Given a readable stream *dp* which is presumed to be positioned at the beginning of a line, skip to the next occurrence of any multipart boundary appearing in *stack*. If *dest* is non-zero, text up to but not including the boundary is written to it. If *stack* is 0 or empty, text up to end-of-file is read. If end-of-file is encountered while a boundary is expected, the entire stack is unwound.

The text is read line by line from *dp* using `mime_ReadLine`, which recognizes three different character sequences as a valid newline (*vide supra*). When copying lines to *dest*, the parameter *newline* controls how newlines are to be depicted. If *newline* is 0, newline sequences read from *dp* are copied as-is to *dest*. If *newline* is a string, that string is used as the newline sequence. (Typically, *newline* will be one of the convenience variables `mime_CR`, `mime_CRLF`, or `mime_LF`.) When an unterminated line is read from *dp* (because end-of-file was encountered), no newline is written to *dest*.

In a properly-formatted MIME stream, only the innermost multipart boundary is expected to be found; but we expect that nesting errors will be common in received MIME mail. Hence any multipart boundary placed on the stack with `mime_MultipartStart` is accepted when found. If the found boundary does not correspond to the top element of the stack, the stack is unwound with `mime_Unwind` until it is the top element. If the found boundary is a multipart terminator rather than merely a separator (that is, `--boundary--` instead of `--boundary`), then that stack frame is unwound also. In the normal case—a non-terminating boundary corresponding to the top of the stack—no stack frames are unwound.

Return value is the number of stack frames unwound. The stream is left positioned at the start of the line following the found boundary.

`void mime_Unwind` (*struct glist* *`stack`, *int* `n`) [Function]

Unwind *n* stack frames from *stack*. As each frame is unwound, its *cleanup* function (as provided in `mime_MultipartStart`), if any, is invoked with the corresponding *cleanup_data* as an argument.

`char * mime_ParseContentType` (*char *`str`, char **`subtype`, struct glist *`plist`*) [Function]

Given *str*, which is presumed to be the value of a Content-Type header, parse the MIME type/subtype pair and any "name=value" parameters. If parsing was successful, the MIME type string is returned and `*subtype` is set to the subtype string. Parameters are added to *plist*, which is a Glist of `struct mime_pair`s. If parsing fails, the return value is 0. Each of *subtype* and *plist* may be 0 to indicate that the corresponding data is not needed.

The string that is returned and the string to which `*subtype` points are stored in private static buffers which are overwritten with each call.

`void mime_AnalyzeHeaders` (*struct glist *`hlist`, struct glist **`plistp`, char **`type`, char **`subtype`, char **`boundary`, char **`encoding`*) [Function]

Given a list of headers *hlist* such as that yielded by `mime_Headers`, look for and extract MIME information.

If a Content-Transfer-Encoding header is found, `*encoding` is set to its value (which is extracted using `mime_NextToken`).

If a Content-Type header is found, then `mime_ParseContentType` is called as follows:

```
*type = mime_ParseContentType(header,
                              *subtype,
                              *plistp);
```

The parameter *plistp* may be 0 to indicate that the parameter list is not needed. On the other hand, if *plist* is not 0 but `*plistp` is 0, then `*plistp` is first set to point to a static, internally-allocated, empty Glist which is overwritten with each call.

After the call to `mime_ParseContentType`, if `*type` is `"multipart"`, then a `boundary` parameter is sought in `*plistp`. (If *plistp* was passed as 0, an internal parameter list is computed for this purpose anyway.) If found, `*boundary` is set to the value of the parameter.

Each of `*type`, `*subtype`, `*boundary`, and `*encoding` is set to 0 if an appropriate value is not found. The values of `*type` and `*subtype`, if set, will be static buffers in `mime_ParseContentType`. The value of `*encoding`, if set, is a static buffer in `mime_AnalyzeHeaders`, overwritten with each call. The value of `*boundary`, if set, is the same copy of the boundary string as stored in the corresponding entry of `*plistp`.

Any of *subtype*, *boundary*, and *encoding* may be passed as 0 to indicate that the corresponding data is not needed. If both *boundary* and *plistp* are 0, no internal parameter list is created in the case that `"multipart"` is encountered.

`void mime_pair_init` (*struct mime_pair *`p`*) [Function]

Initializes the pair pointed to by *p* (by calling `dynstr_Init` on its two fields).

`void mime_pair_destroy` (*struct mime_pair *`p`*) [Function]

Finalizes the pair pointed to by *p* (by calling `dynstr_Destroy` on its two fields).

`void mime_GenMultipart` (*struct dpipe* *`dp`, *const char* *`subtype`, *struct glist* *`parts`) [Function]

Write to *dp* the MIME syntax for a `multipart/subtype` bodypart whose subparts are in *parts*. See RFC1521 for legal values of *subtype*.

Each element of *parts* is a pointer to a Dpipe. Each Dpipe must be a readable stream containing a complete, MIME-conformant stream for one subpart. This includes any relevant headers and a properly-encoded body. Note that the output of this function (in *dp*) may be used as input to a future invocation of this function, to create nested multiparts.

The stream generated by this function looks something like this:

```
Content-Type: multipart/subtype; boundary=boundary-string


--boundary-string
contents of first subpart

--boundary-string
contents of second subpart
...

--boundary-string
contents of last subpart

--boundary-string--
```

where *boundary-string* is an automatically-generated unique string.

2 Usage

This chapter outlines how MIME parsing using this library should proceed. In the examples below, the variable *dp* refers to a readable Dpipe which is the source of a MIME-conformant stream. For clarity, exception-handling constructs which ought to be present have been omitted.

The following example illustrates parsing a MIME stream with possible deeply-nested multiparts. Each leaf part is written to a destination Dpipe named `dest`. The caller should naturally initialize `dest` on each iteration to point to an appropriate destination for the data given its MIME type and subtype, and its depth in the multipart hierarchy. The nesting depth of the leaf part being read at any point is given (in this example) by `glist_Length(&stack)`.

Note that the call to `mime_NextBoundary` which sends a leaf part to `dest` does not perform any base64 or quoted-printable decoding. The caller may want to initialize `dest` to be one end of a Dpipeline which performs the appropriate decoding and then sends the output to its final destination (e.g., a file).

```
struct glist hlist, stack, *plist;
char *type, *subtype, *boundary, *encoding;
int loop;

glist_Init(&stack, sizeof (struct mime_stackelt), 4);

do {
    loop = 0;
    glist_Init(&hlist, sizeof (struct mime_pair), 8);
    mime_Headers(dp, &hlist, 0);
    mime_Readline(dp, 0); /* discard the separator line */
    mime_AnalyzeHeaders(&hlist, &plist,
                        &type, &subtype,
                        &boundary, &encoding);
    if (type && !strcasecmp(type, "multipart")) {
        /* It's a multipart; push the boundary string on
         * the stack and scan for the first boundary
         */
        mime_MultipartStart(&stack, boundary, 0, 0);
```

At this point, the boundary string for the pending multipart has been placed on `stack`. Next we search forward for the first occurrence of the boundary using `mime_NextBoundary`. For robustness, however, we don't just call mime_NextBoundary once; we call it as many times as it returns non-zero. Naturally we expect it to return zero, but if the message syntax is garbled, we might encounter a different boundary line first, in which case we want to keep slurping up possibly-terminating boundaries until we're again positioned at the start of a new bodypart (which condition is indicated by a zero return from `mime_NextBoundary`).

```
            while (mime_NextBoundary(dp, 0, &stack, 0))
                ;
        } else if (type && !strcasecmp(type, "message")) {
            /* It's a message/something.  Loop again to
             * read its (embedded) headers and body
             */
            loop = 1;
        } else {
            /* It's a leaf part */
            struct dpipe dest;
            int unwound;

            dpipe_Init(&dest, ...whatever is appropriate... */);
            unwound = mime_NextBoundary(dp, &dest, &stack, mime_LF);
```

Of course, the use of `mime_LF` here is just an example.

```
            dpipe_Close(&dest);
            dpipe_Destroy(&dest);
```

Again, call `mime_NextBoundary` repeatedly until we're at the beginning of a new bodypart somewhere.

```
            if (unwound)
                while (mime_NextBoundary(dp, 0, &stack, 0))
                    ;
        }
        glist_CleanDestroy(&hlist, mime_pair_finalize);
    } while (loop || !glist_EmptyP(&stack));

    glist_Destroy(&stack);
```

The next example builds on the first one but treats `multipart/alternative` in the way intended by RFC1521: that is, upon entering a `multipart/alternative`, each subpart is stashed somewhere; and upon exiting the `multipart/alternative`, a single suitable subpart is selected for presentation. While scanning the subparts of a `multipart/alternative`, parsing does not descend into subsubparts. The type `struct part_data` is a hypothetical structure for holding information about each subpart while a `multipart/alternative` is being scanned.

```
    struct glist hlist, stack, *plist;
    char *type, *subtype, *boundary, *encoding;
    int loop;

    glist_Init(&stack, sizeof (struct mime_stackelt), 4);

    do {
        loop = 0;
        glist_Init(&hlist, sizeof (struct mime_pair), 8);
        mime_Headers(dp, &hlist, 0);
        mime_Readline(dp, 0); /* discard separator line */
```

```
mime_AnalyzeHeaders(&hlist, &plist,
                    &type, &subtype,
                    &boundary, &encoding);
if (type && !strcasecmp(type, "multipart")) {
    if (subtype && !strcasecmp(type, "alternative")) {
        /* it's a multipart/alternative */
        struct glist subparts;
        int unwound;

        glist_Init(&subparts, sizeof (struct part_data), 4);
        mime_MultipartStart(&stack, boundary,
                            alternative_finish,
                            &subparts);
        if (unwound = mime_NextBoundary(dp, 0, &stack, 0)) {
            while (mime_NextBoundary(dp, 0, &stack, 0))
                ;
        } else {
            /* doing multipart/alternative subparts */
            struct glist hlist2;
            char *type, *subtype;
            struct dpipe dest;

            glist_Init(&hlist2, sizeof (struct mime_pair), 4);
            do {
                mime_Headers(dp, &hlist2, 0);
                mime_Readline(dp, 0);
                mime_AnalyzeHeaders(&hlist2, 0,
                                    &type, &subtype,
                                    0, 0);
                dpipe_Init(&dest, ...whatever...);
                unwound = mime_NextBoundary(dp, &dest,
                                            &stack, mime_LF);
```

```
                    dpipe_Close(&dest);
                    dpipe_Destroy(&dest);
                    /* Add an entry to subparts */
                } while (!unwound);
                while (mime_NextBoundary(dp, 0, &stack, 0))
                    ;
            }
        } else {
            /* multipart/something-else */
            mime_MultipartStart(&stack, boundary, 0, 0);
            while (mime_NextBoundary(dp, 0, &stack, 0))
                ;
        }
    } else if (type && !strcasecmp(type, "message")) {
        /* It's a message/something.  Loop again to
         * read its (embedded) headers and body
         */
        loop = 1;
    } else {
        /* It's a leaf part */
        struct dpipe dest;
        int unwound;

        dpipe_Init(&dest, ...whatever...);
        unwound = mime_NextBoundary(dp, &dest, &stack, mime_LF);
        dpipe_Close(&dest);
        dpipe_Destroy(&dest);
        if (unwound)
            while (mime_NextBoundary(dp, 0, &stack, 0))
                ;
    }
    glist_CleanDestroy(&hlist, mime_pair_finalize);
} while (loop || !glist_EmptyP(&stack));

glist_Destroy(&stack);
```

Index

Table of Contents