

NAME

mparse – parse and generate Internet text messages and their components

SYNOPSIS

```
#include <mparse.h>
```

```
int mparse_parse(struct mparse_message *, FILE *, FILE *);
```

```
...
```

DESCRIPTION

Mparse is a library of functions for parsing and generating Internet Message Format messages. Such messages are used by a number of applications, such as email, Internet fax, voice messaging, EDI, and Usenet news. The messages consist of a header, which contains one or more fields, and usually has some body content (a more detailed description of message format is provided below).

This software is OSI Certified Open Source Software. OSI Certified is a certification mark of the Open Source Initiative.

To understand the role of messages in applications, as well as the role of **mparse**, some definitions of terms are in order:

A format is an arrangement of content according to a combination of defined syntax and semantics for transferring said content in a manner which facilitates processing by machine. There are several types of formats, some of which interact in complex ways. The Internet Message Format breaks content into three broad categories: the message header which has defined fields with specified syntax, a separator line which serves to distinguish message header from message body, and finally the message body which consists of lines of text. The header fields each consist of a field name, a colon character which separates the field name from the field body, and finally the field body which has a defined syntax. MIME is a collection of standards which describe how to represent various types of complex text, non-textual, and combinations of content within the Internet Message Format (and extended to other contexts such as HTTP). MIME defines a number of media types, including `message/rfc822` which can be used to encapsulate an Internet Message Format message within another Internet Message. Various other media types are defined by MIME and extensions to the MIME standards. These fit into two broad categories: discrete media types used to represent a particular type of content, and composite media types which represent potentially complex collections of content. Composite types are message and multipart categories, each with several subtypes; discrete types include various subtypes of text, audio, image, video, and model types, as well as application-specific types. Many media types are specified with specific formats.

A network protocol is a defined procedure for regulating data transfer. Examples of network protocols include Simple Mail Transfer Protocol (SMTP), Local Mail Transfer Protocol (LMTP), Network News Transfer Protocol (NNTP), File Transfer Protocol (FTP), Internet Message Access Protocol (IMAP), Message Posting Protocol (MPP), and Post Office Protocol (POP).

An application consists of a collection of programs that implement some function, possibly using network protocols to transfer some content in a defined format from a source to one or more destinations. Examples of applications include email, Usenet news, Internet fax, voice messaging, and Electronic Document Interchange (EDI). An application is an end-to-end process.

The relationship between applications, the Internet Message Format, transport protocols, etc. can be seen in the following diagram, which is based on the OSI model:

Application	application-specific and media-specific fields and media types	
	MIME message structure, media types, and fields	
	Internet Message Format (header fields, separator, body)	
	low-level field body components	
Presentation ----- Session	encoding, encryption, packaging LMTP, SMTP, MPP, MTP, EMSD, NNTP, FTP, IMAP, POP, rmail, rnews, submission supplementary notification messages: MDN, DSN, MTSN, bounces, rejections SIP	
Transport	TCP	UUCP
Network	IP	
Data Link		
Physical		

Mparse handles the message format shown at the Application layer, except for media-specific body content (with some notable cases which are handled), and assists with the interface to the next level. That is, mparse handles the Internet Message Format, MIME message structure media types, fields defined in a few specific MIME media types, and the low-level field body components used by the Internet Message Format and MIME and their extensions. The criteria for handling MIME media types which include fields is that the media type must be defined as a group of fields optionally followed by a separator line and other content, and all fields defined must be compatible with RFC 2822 field syntax. Examples of media types that fail to conform to those criteria include message/http and message/sip (which begin with a "start-line" rather than with fields), and message/CPIM (whose syntax for several fields conflicts with RFC 2822 fields). Such non-conforming media type content is treated by mparse as opaque content (much like application/octet-stream). Application authors interested in parsing such media type content may of course do so using application-specific methods.

The message format (excluding media-specific message body content) is specified in the following groups of RFCs:

Overall non-MIME message structure and related issues:

number	description
561	Standardizing Network Mail Headers (obsolete)
724	Proposed Official Standard for the Format of ARPA Network Messages (obsoleted by RFC 733)
733	STANDARD FOR THE FORMAT OF ARPA NETWORK TEXT MESSAGES (obsoleted by RFC 822)
822	Standard for the format of ARPA Internet text messages (see also 2822)
2368	The mailto URL scheme
2822	Internet Message Format
4096	Policy-Mandated Labels Such as "Adv:" in Email Subject Headers Considered Ineffective At Best

Non-MIME extension message header fields, and supplementary message header field definitions:

number	description
788	Simple Mail Transfer Protocol (see also 821, 2821)
821	Simple Mail Transfer Protocol (see also 2821)
850	Standard for interchange of USENET messages (obsoleted by 1036)
886	Proposed Standard for Message Header Munging
987	Mapping Between X.400 and RFC 822 (updated by 1026, obsoleted by 1148)
1026	Addendum to RFC 987 (Mapping between X.400 and RFC-822) (obsoleted by 1327, 1148)
1036	Standard for interchange of USENET messages
1049	A CONTENT-TYPE HEADER FIELD FOR INTERNET MESSAGES
1138	Mapping between X.400(1988) / ISO 10021 and RFC 822 (obsoleted by 1148)
1148	Mapping between X.400(1988) / ISO 10021 and RFC 822 (obsoleted by 2156)
1154	Encoding Header Field for Internet Messages (obsoleted by 1505)
1327	Mapping between X.400(1988) / ISO 10021 and RFC 822 (obsoleted by 2156)
1505	Encoding Header Field for Internet Messages
2156	MIXER (Mime Internet X.400 Enhanced Relay): Mapping between X.400 and RFC 822/MIME
2369	The Use of URLs as Meta-Syntax for Core Mail List Commands and their Transport through Message Header Fields
2821	Simple Mail Transfer Protocol
2919	List-Id: A Structured Field and Namespace for the Identification of Mailing Lists
3834	Recommendations for Automatic Responses to Electronic Mail
3458	Message Context for Internet Mail
3865	A No Soliciting Simple Mail Transfer Protocol (SMTP) Service Extension
3939	Calling Line Identification for Voice Mail Messages
4021	Registration of Mail and MIME Header Fields

Low-level field components:

number	description
1034	Domain names - concepts and facilities (updated by 2181, 4343, 4592)
1035	Domain names - implementation and specification (updated by 2181, 4343)
1123	Requirements for Internet Hosts - Application and Support (updated by 2181)
1342	Representation of Non-ASCII Text in Internet Message Headers (obsoleted by 1522)
1893	Enhanced Mail System Status Codes (obsoleted by 3463)
1958	Architectural Principles of the Internet
2181	Clarifications to the DNS Specification (updated by 4343)
2277	IETF Policy on Character Sets and Languages
2373	IP Version 6 Addressing Architecture
2396	Uniform Resource Identifiers (URI): Generic Syntax
2978	IANA Charset Registration Procedures
3066	Tags for the Identification of Languages
3463	Enhanced Mail System Status Codes
3692	Assigning Experimental and Testing Numbers Considered Useful
3848	ESMTP and LMTP Transmission Types Registration
3938	Video-Message Message-Context
4343	Domain Name System (DNS) Case Insensitivity Clarification
4592	The Role of Wildcards in the Domain Name System

MIME message structure and fields, MIME enhancements to fields, MIME extension field definitions, and definition of MIME media types requiring specific Content-Type parameters:

number	description
1341	MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies (obsoleted by 1521)
1344	Implications of MIME for Internet Mail Gateways
1521	MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies (obsoleted by 2045, 2046, 2047, 2048, 2049; updated by 1590)
1522	MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text (obsoleted by 2045, 2046, 2047, 2048, 2049; updated by 1590)
1847	Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted
1864	The Content-MD5 Header Field
2017	Definition of the URL MIME External-Body Access-Type
2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies
2046	Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types
2047	Multipurpose Internet Mail Extensions (MIME) Part Two: Message Header Extensions for Non-ASCII Text
2049	Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples
2184	MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations (obsoleted by 2231)
2231	MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations
2311	S/MIME Version 2 Message Specification
2387	The MIME Multipart/Related Content-type
2421	Voice Profile for Internet Mail - version 2
2424	Content Duration MIME Header Definition (obsoleted by 3803)
2425	A MIME Content-Type for Directory Information
2480	Gateways and MIME Security Multiparts
2533	A Syntax for Describing Media Feature Sets (updated by 2738)
2557	MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)
2586	The Audio/L16 MIME content type
2616	Hypertext Transfer Protocol -- HTTP/1.1
2633	S/MIME Version 3 Message Specification (obsoleted by 3851)
2634	Enhanced Security Services for S/MIME
2652	MIME Object Definitions for the Common Indexing Protocol (CIP)
2660	The Secure HyperText Transfer Protocol
2738	Corrections to "A Syntax for Describing Media Feature Sets"
2912	Indicating Media Features for MIME Content
2913	MIME Content Types in Media Feature Expressions
3156	MIME Security with OpenPGP
3282	Content Language Headers
3801	Voice Profile for Internet Mail - version 2 (VPIMv2)
3803	Content Duration MIME Header Definition
3804	Voice Profile for Internet Mail (VPIM) Addressing
3851	Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification
4194	The S Hexdump Format

Notification message structure and applications:

number	description
1892	The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages (obsoleted by 3462)
1894	An Extensible Message Format for Delivery Status Notifications (obsoleted by 3464)
2298	An Extensible Message Format for Message Disposition Notifications (obsoleted by 3798)
2530	Indicating Supported Media Features using Extensions to DSN and MDN
2634	Enhanced Security Services for S/MIME
3297	Content Negotiation for Messaging Services
3335	MIME-based Secure Peer-to-Peer Business Data Interchange over the Internet
3462	The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages
3464	An Extensible Message Format for Delivery Status Notifications
3798	An Extensible Message Format for Message Disposition Notifications
3834	Recommendations for Automatic Responses to Electronic Mail
3886	An Extensible Message Format for Message Tracking Responses

RFC 822 was preceded by RFCs 561, 724, and 733. Message content (header and body) according to those older RFCs is recognized and parsed by **mparse** to the extent possible consistent with modern syntax, however many constructs provided for by those RFCs are no longer valid and a few are incompatible with modern syntax. Some are hopelessly ambiguous.

Mparse also assists with network message transfer protocols that return status responses. Applicable network message transfer protocols are defined in the following RFCs:

number	description
765	FILE TRANSFER PROTOCOL (obsoleted by 959)
772	MAIL TRANSFER PROTOCOL (obsoleted by 780)
780	MAIL TRANSFER PROTOCOL (see also 784, 785, 786; obsoleted by 788)
788	Simple Mail Transfer Protocol (see also 821, 2821)
821	Simple Mail Transfer Protocol (see also 2821)
977	Network News Transfer Protocol
1204	Message Posting Protocol (MPP)
2033	Local Mail Transfer Protocol
2476	Message Submission (obsoleted by 4409)
2821	Simple Mail Transfer Protocol
4409	Message Submission for Mail

Note that some RFCs define both message format and message transfer protocols.

Note that some RFCs contain errors. There is an errata page at <http://www.rfc-editor.org/errata.html>.

The database at <http://www.iana.org/assignments/numbers.html> which was formerly in RFC 1700, is also applicable.

The remaining lists of RFCs are provided for reference.

Other network protocols (message access, transfer of messages without status responses, non-message file transfer, extensions) that may be of interest include:

number	description
850	Standard for interchange of USENET messages (obsoleted by 1036)
918	POST OFFICE PROTOCOL (obsoleted by 937)
937	POST OFFICE PROTOCOL - VERSION 2 (historic)
959	FILE TRANSFER PROTOCOL (FTP)
976	UUCP Mail Interchange Format Standard
1036	Standard for interchange of USENET messages
1064	INTERACTIVE MAIL ACCESS PROTOCOL- VERSION 2 (obsoleted by 1176, 1203)
1081	Post Office Protocol - Version 3 (obsoleted by 1225)
1082	Post Office Protocol - Version 3 Extended Service Offerings
1123	Requirements for Internet Hosts -- Application and Support
1176	INTERACTIVE MAIL ACCESS PROTOCOL- VERSION 2 (experimental)
1203	INTERACTIVE MAIL ACCESS PROTOCOL- VERSION 3 (historic)
1225	Post Office Protocol - Version 3 (obsoleted by 1460)
1425	SMTP Service Extensions (obsoleted by 1651)
1426	SMTP Service Extension for 8bit-Mime transport (obsoleted by 1652)
1427	SMTP Service Extension for Message Size Declaration (obsoleted by 1653)
1460	Post Office Protocol - Version 3 (obsoleted by 1725)
1651	SMTP Service Extensions (obsoleted by 1869)
1652	SMTP Service Extension for 8bit-Mime transport
1653	SMTP Service Extension for Message Size Declaration (obsoleted by 1870)
1725	Post Office Protocol - Version 3 (obsoleted by 1939)
1730	INTERNET MESSAGE ACCESS PROTOCOL- VERSION 4 (obsoleted by 2060, 2061)
1734	POP3 AUTHentication command
1830	SMTP Service Extensions for Transmission of Large and Binary MIME Messages (obsoleted by 3030)
1845	SMTP Service Extension for Checkpoint/Restart (experimental)
1846	SMTP 521 Reply Code
1854	SMTP Service Extension for Command Pipelining (obsoleted by 2197)
1869	SMTP Service Extensions (obsoleted by 2821)
1891	SMTP Service Extension for Delivery Status Notifications (obsoleted by 3461)
1893	Enhanced Mail System Status Codes (obsoleted by 3463)
1939	Post Office Protocol - Version 3 (updated by 1957, 2449)
1985	SMTP Service Extension for Remote Message Queue Starting
2034	SMTP Service Extension for Returning Enhanced Error Codes
2060	INTERNET MESSAGE ACCESS PROTOCOL- VERSION 4rev1 (obsoleted by 3501)
2197	SMTP Service Extension for Command Pipelining (obsoleted by 2920)
2449	POP3 Extension Mechanism
2524	Neda's Efficient Mail Submission and Delivery (EMSD) Protocol Specification Version 1.3
2920	SMTP Service Extension for Command Pipelining
3030	SMTP Service Extensions for Transmission of Large and Binary MIME Messages
3207	SMTP Service Extension for Secure SMTP over Transport Layer Security
3461	Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs)
3463	Enhanced Mail System Status Codes
3501	INTERNET MESSAGE ACCESS PROTOCOL- VERSION 4rev1 (updated by 4466)
3885	SMTP Service Extension for Message Tracking
3887	Message Tracking Query Protocol
3888	Message Tracking Model and Requirements
4466	Collected Extensions to IMAP4 ABNF
4496	Open Pluggable Edge Services (OPES) SMTP Use Cases

The following RFCs specify MIME media types which have no processing implications for **mparse**. The list is provided for reference only.

number	description
1523	The text/enriched MIME Content-type (Obsoleted by RFC1563, RFC1896)
1563	The text/enriched MIME Content-type (Obsoleted by RFC1896)
1740	MIME Encapsulation of Macintosh Files - MacMIME
1741	MIME Content Type for BinHex Encoded Files
1767	MIME Encapsulation of EDI Objects
1872	The MIME Multipart/Related Content-type
1896	The text/enriched MIME Content-type
1927	Suggested Additional MIME Types for Associating Documents (1 April 1996)
2112	The MIME Multipart/Related Content-type
2159	A MIME Body Part for FAX
2161	A MIME Body Part for ODA
2301	File Format for Internet Fax (obsoleted by 3949)
2302	Tag Image File Format (TIFF) - image/tiff MIME Sub-type
2422	Toll Quality Voice - 32 kbit/s ADPCM MIME Sub-type Registration
2423	VPIM Voice Message MIME Sub-type Registration
2426	vCard MIME Directory Profile
2503	MIME Types for Use with the ISO ILL Protocol
2646	The Text/Plain Format Parameter (obsoleted by 3676)
2927	MIME Directory Profile for LDAP Schema
3009	Registration of parityfec MIME types
3028	Sieve: A Mail Filtering Language
3073	Portable Font Resource (PFR) - application/font-tdpfr MIME Sub-type Registration
3204	MIME media types for ISUP and QSIG Objects
3240	Digital Imaging and Communications in Medicine (DICOM) - Application/dicom MIME Sub-type Registration
3250	Tag Image File Format Fax eXtended (TIFF-FX) - image/tiff-fx MIME Sub-type Registration (obsoleted by 3950)
3302	Tag Image File Format (TIFF) - image/tiff MIME Sub-type Registration
3362	Real-time Facsimile (T.38) - image/t38 MIME Sub-type Registration
3391	The MIME Application/Vnd.pwg-multiplexed Content-Type
3459	Critical Content Multi-purpose Internet Mail Extensions (MIME) Parameter (Updates RFC3204)
3555	MIME Type Registration of RTP Payload Formats
3676	The Text/Plain Format and DelSp Parameters
3745	MIME Type Registrations for JPEG 2000 (ISO/IEC 15444)
3778	The application/pdf Media Type
3802	Toll Quality Voice - 32 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM) MIME Sub-type Registration (obsoletes 2422)
3823	MIME Media Type for the Systems Biology Markup Language (SBML)
3839	MIME Type Registrations for 3rd Generation Partnership Project (3GPP) Multimedia files
3949	File Format for Internet Fax. R. Buckley (Obsoletes RFC2301)
3950	Tag Image File Format Fax eXtended (TIFF-FX) - image/tiff-fx MIME Sub-type Registration (Obsoletes RFC3250)
3870	application/rdf+xml Media Type Registration
3902	The "application/soap+xml" media type
4027	Domain Name System Media Types
4047	MIME Sub-type Registrations for Flexible Image Transport System (FITS)
4155	The application/mbox Media Type
4180	Common Format and MIME Type for Comma-Separated Values (CSV) Files
4263	Media Subtype Registration for Media Type text/troff
4329	Scripting Media Types
4337	MIME Type Registration for MPEG-4
4374	The application/xv+xml Media Type
4393	MIME Type Registrations for 3GPP2 Multimedia Files
4536	The application/smil and application/smil+xml Media Types

- 4539 Media Type Registration for the Society of Motion Picture and Television Engineers (SMPTE) Material Exchange Format (MXF)
- 4573 MIME Type Registration for RTP Payload Format for H.224
- 4627 The application/json Media Type for JavaScript Object Notation (JSON)

The following RFCs contain information useful in interpreting RFCs, and the list is provided for reference:

number	description
1958	Architectural Principles of the Internet
2026	The Internet Standards Process -- Revision 3
2028	The Organizations Involved in the IETF Standards Process
2119	Key words for use in RFCs to Indicate Requirement Levels
2277	IETF Policy on Character Sets and Languages
3117	On the Design of Application Protocols
3160	The Tao of IETF - A Novice's Guide to the Internet Engineering Task Force
3536	Terminology Used in Internationalization in the IETF
3692	Assigning Experimental and Testing Numbers Considered Useful
3930	The Protocol versus Document Points of View in Computer Protocols
3935	A Mission Statement for the IETF
3967	Clarifying when Standards Track Documents may Refer Normatively to Documents at a Lower Level

The following RFCs contain information which may be useful to implementors using **mparse**, and the list is provided for reference:

number	description
1344	Implications of MIME for Internet Mail Gateways
1428	Transition of Internet Mail from Just-Send-8 to 8bit-SMTP/MIME
1496	Rules for Downgrading Messages from X.400/88 to X.400/84 When MIME Content-Types are Present in the Messages
1556	Handling of Bi-directional Texts in MIME
1820	Multimedia E-mail (MIME) User Agent Checklist (obsoleted by 1844)
1844	Multimedia E-mail (MIME) User Agent Checklist
1865	EDI Meets the Internet
2048	Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures (obsoleted by RFCs 4288 and 4289)
2305	A Simple Mode of Facsimile Using Internet Mail (obsoleted by 3965)
2532	Extended Facsimile Using Internet Mail
3805	Printer MIB v2
3808	IANA Charset MIB
3864	Registration Procedures for Message Header Fields
3965	A Simple Mode of Facsimile Using Internet Mail
4134	Examples of S/MIME Messages
4143	Facsimile Using Internet Mail (IFAX) Service of ENUM
4160	Internet Fax Gateway Requirements
4161	Guidelines for Optional Services for Internet Fax Gateways
4239	Internet Voice Messaging (IVM)
4249	Implementer-Friendly Specification of Message and MIME-Part Header Fields and Field Components
4270	Attacks on Cryptographic Hashes in Internet Protocols
4288	Media Type Specifications and Registration Procedures
4289	Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures
4355	IANA Registration for Enumservices email, fax, mms, ems, and sms
4356	Mapping Between the Multimedia Messaging Service (MMS) and Internet Mail
4367	What's in a Name: False Assumptions about DNS Names
4550	Internet Email to Support Diverse Service Environments (Lemonade) Profile

Mparse can serve as the skeleton for applications which need to be able to generate or process text

messages (mail and news user agents, submission agents, transfer agents, delivery agents, spam filters, cancelbots, etc.). **Mparse** recognizes and reports syntax errors and provides hooks for user processing of message components (header lines, addresses, body sections, etc.). A program using **mparse** must be compiled with **-lmparse** on the **cc** command line.

Mparse is implemented as a lexical analyzer, a parser, and a number of support functions and keyword databases. The primary goal of **mparse** is correct parsing and generation of Internet messages. Because the Internet message format is specified in terms of a modified BNF, a parser generator using source which can be constructed from the modified BNF is the central component of **mparse**. The parser and associated lexical analyzer are designed to accept a superset of legal Internet messages, so as to be able to accommodate common syntax errors. Support functions then check the parsed message components for such syntax errors and deprecated constructs, providing the capability of producing informative error messages and warnings.

A secondary goal of **mparse** is maintainability. Keywords which may change, such as registered values for charsets, language tags, etc. are maintained in separate source code files, most of which can be generated automatically from a reference source.

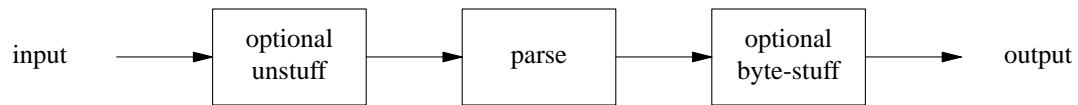
A third goal of **mparse** is flexibility. **Mparse** provides a mechanism where an application can configure different processing modes, and is designed to call application-provided functions for processing message components and in the event of exceptional conditions (errors and warnings). As such, **mparse** can be used by diverse applications. Implementation of **mparse** as a library of functions also facilitates realization of this goal.

Performance was also given consideration in the design of **mparse**, but not at the expense of the other goals. For example, highly-tuned hash tables are used for efficient case-insensitive matching of keywords such as header field names, day-of-week and month names, charset tags, etc. The use of a formal parser and lexical analyzer do have performance implications, but the goals of correctness and maintainability, which are enhanced via the formal parser, were considered more important than raw performance.

A novel approach is used in generation of message components such as header fields. The component is generated from supplied text, then is parsed to detect syntax errors. Each error is flagged, and those errors which are correctable are repaired.

Before describing the details of **mparse** operation, some context is necessary. Simple text messages consist of three parts: header fields, an empty line acting as a separator, and body text. A simple message might consist of header fields alone; i.e. no body text (in that case there might or might not be an empty line following the header fields). At least some header fields must be present. MIME multipart messages (see RFC 2046) consist of several sections separated by boundary delimiters, which are specially formatted text lines. Between these delimiters are encapsulated body parts, each of which consists of optional MIME header fields, a mandatory empty separator line, and some content (possibly encoded as text for transport). MIME also provides for message encapsulation, in which there are generally MIME header fields, a mandatory empty separator line, and the encapsulated message (header fields, separator, and body). Some exceptions are delivery status notification (DSN) and message tracking status notification (MTSN) messages, which have multiple sets of fields separated by empty lines. Other exceptions include media types `message/sip`, `message/sipfrag`, `message/CPIM`, `message/http` and `message/s-http` which do not have normal message header fields in the encapsulation.

Messages received via SMTP, POP, or NNTP protocols may have been byte-stuffed, i.e. lines beginning with a '.' character may have had an additional '.' inserted. Likewise, messages which must be passed via SMTP, POP, or NNTP protocols should be so processed. **Mparse** processes messages according to the following model:



When byte-stuffing is being removed from the input, a line consisting of a lone '.' is considered to be the end of the message (consistent with POP RETR command response, for example). When byte-stuffing is applied at the output, a line containing a lone '.' is output after the message body (consistent with the SMTP DATA command).

Input to **mparse** might come from a file or pipe or from a socket. **Mparse** provides for input timeout specifically for socket input.

Mparse can copy its input to output, or such copying can be suppressed. Additional output can be generated for errors in the input message, and application-provided functions can be called, which might generate additional output. Separate *stdio* **FILE** pointers are used for copied output, for general **mparse** function errors, and for error messages generated in response to errors in the input.

The latter error messages can be suppressed, and if not suppressed appear in three varieties:

line	description
X-NG:	An RFC violation. The offending input is pinpointed if possible.
X-Warning:	Context-sensitive constructs which may violate RFCs under some conditions. The offending input is pinpointed if possible.
X-Err:	An error in the number or type of header fields. Issued after the last header line.

Overall operation of **mparse** is controlled by setting members of a **mparse_message** structure, which is pointed to by a function call argument to one of the **mparse** functions. That structure is:

member (struct mparse_message)	description
void *userptr;	user pointer; not set or used by parser, may be used by applications to pass data to hooks
struct mparse_entity *top;	top-level entity
struct mparse_hooks *hooks;	application hooks
unsigned int context;	message processing context
const struct mparse_charset *charset;	default charset to use for generation and/or repair
const char *language_tag;	default language tag for generation and/or repair
double timeout;	input timeout in seconds
void *r_flex;	pointer to data used by reentrant lexical analyzer
int start_cond;	lexical analyzer start condition
FILE *merr;	X-NG, X-Err, X-Warning header fields go here
struct mparse_debug *dbg;	for debugging
char modes[]	bitmap of RFC processing modes. Use <code>mparse_add_mode()</code> , <code>mparse_clear_modes()</code> , <code>mparse_in_mode()</code> , <code>mparse_remove_mode()</code> to read/modify.
int language_index;	language for warning and error messages
int linelen	desired maximum body line length when generating content
unsigned int edebug : 1;	debug storage associated with error structures
unsigned int gdebug : 1;	provide debugging information related to parsing
unsigned int hdebug : 1;	debug storage associated with fields
unsigned int ldebug : 1;	provide debugging information related to lexical analysis
unsigned int ndebug : 1;	debug storage associated with entity
unsigned int pdebug : 1;	debug storage associated with MIME <code>mparse_parameters</code>
unsigned int rdebug : 1;	debug storage associated with protocol status
unsigned int sdebug : 1;	debug storage associated with lists
unsigned int tdebug : 1;	debug storage associated with tokens
unsigned int internal_parse : 1;	internal use
unsigned int ioerr : 1;	input timeout or other input error
unsigned int canonicalize : 1;	convert strings to canonical form when generating fields
unsigned int byte_stuff : 1;	byte-stuff output (body lines beginning with '.' have an additional '.' prepended)
unsigned int byte_unstuff : 1;	remove byte stuffing at input (strip '.' at beginning of body lines)
unsigned int experimental : 1;	support experimental and private-use names
unsigned int header_only : 1;	don't process body text
unsigned int no_copy : 1;	suppress output
unsigned int suppress_errors : 1;	suppress error X- header field generation
unsigned int suppress_warnings : 1;	suppress warning X- header field generation

The **mparse_message** structure provides a user pointer which may be used to pass arbitrary application data to application-defined functions.

The bitmap of **modes** should be accessed and modified through the library functions **mparse_add_mode**, **mparse_clear_modes**, **mparse_in_mode**, and **mparse_remove_mode**:

```
int mparse_add_mode(struct mparse_message *, int);
int mparse_clear_modes(struct mparse_message *);
int mparse_in_mode(struct mparse_message *, int);
int mparse_remove_mode(struct mparse_message *, int);
```

The integer arguments to **mparse_add_mode**, **mparse_in_mode**, and **mparse_remove_mode** are RFC numbers; e.g. **mparse_add_mode(p, 822)** sets processing to report errors relevant to RFC 822. Specifying 0 includes “common-sense” errors. If a NULL pointer or invalid RFC number is supplied, **mparse_add_mode** returns -1 and sets *errno* to EINVAL. **mparse_in_mode** returns 1 if processing for the specified RFC is in effect, 0 if not, and -1 (with *errno* set to EINVAL) on error. **mparse_clear_modes** turns off all processing.

Many **mparse** functions return an integer or a pointer. In event of an error, such as described above, these functions return a negative integer or a *NULL* pointer and set the global variable *errno* to indicate the error. Some of the *errno* values used are the same as those which may be set by system calls (such as EINVAL in the above functions), while there are message-specific values which are used where applicable:

symbolic name	description
MPARSE_ERRNO_ESIGNED	signed message or entity
MPARSE_ERRNO_EORIGINAL	original message or entity
MPARSE_ERRNO_EENCRYPTED	encrypted message or entity
MPARSE_ERRNO_ENOADDRESS	no address
MPARSE_ERRNO_EMULTIADDRESS	multiple addresses (where only one is appropriate)
MPARSE_ERRNO_ECOMPOSITE	composite media type
MPARSE_ERRNO_EINCOMPATIBLE	incompatible type, subtype, charset, etc.
MPARSE_ERRNO_ECACHED	information carried in field is cached
MPARSE_ERRNO_ENODEFAULT	cached field information has no default value

A string corresponding to the *errno* value may be obtained by calling the function **mparse_strerror**:

```
const char *mparse_strerror(const struct mparse_message *, int);
```

giving the pointer to the affected **mparse_message** structure as the first argument, and the value of *errno* as the second argument. If the *errno* value is one of the above messaging-specific values, a language-dependent string is returned; otherwise the system **strerror** function is called. Use of the *language_index* member of the **mparse_message** structure allows dynamically changing the language, or changing it on a per-message basis without repeated calls to change a system-dependent "locale" (which might not be available on non-POSIX systems). Note that such calls are necessary, where available, to change the language for the system **strerror** function, and further note that error messages generated by the parser-generator (bison) code are in a fixed language which is determined at the time that the parser is built.

The function

```
int mparse_set_msg_language(struct mparse_message *message, const char *str, unsigned int len);
```

returns zero after setting the *language_index* member of the *mparse_message* structure to the appropriate value for the language described by *str* if it is recognized. It returns a positive integer (which is always suitable for use as the *language_index* member, but which must be set by the application) if the language is not recognized, and a negative integer (with *errno* set appropriately) if *mparse_message* or *str* is a *NULL* pointer.

Mparse can be called from diverse applications for parsing or generating messages, for message syntax validation, etc. The *context* member of the **mparse_message** structure should be set to a value to indicate the processing environment. It is made up of two parts: one indicates one of the following primary processing roles:

symbolic name	description
MPARSE_PRIMARY_ROLE_GENERATION	generation of messages, content errors are repaired if possible; e.g. user agent
MPARSE_PRIMARY_ROLE_VALIDATION	content validity checks, e.g. mail submission agent, news injection agent, gateway
MPARSE_PRIMARY_ROLE_TRANSPORT	content transport; existing content not altered (except for MPARSE_SECONDARY_ROLE_TRANSFORM , etc.) (some content (trace fields) may be added)
MPARSE_PRIMARY_ROLE_ACCESS	content access for display, filtering, content extraction; no modification

Only one of the above primary roles should be specified. In addition to the primary role, one or more secondary roles may be indicated by bitwise ORing one or more of the following values with the primary role value:

symbolic name	description
MPARSE_SECONDARY_ROLE_REPAIR	repair content errors during MPARSE_PRIMARY_ROLE_GENERATION , MPARSE_PRIMARY_ROLE_VALIDATION , or at gateways
MPARSE_SECONDARY_ROLE_TRANSFORM	content transformations (transcoding, reassembly, etc.) during MPARSE_PRIMARY_ROLE_TRANSPORT or at gateways
MPARSE_SECONDARY_ROLE_TRANSFORM_7BIT	encode to 7bit domain for MPARSE_PRIMARY_ROLE_TRANSPORT or at gateways

The default value for *context* is **MPARSE_PRIMARY_ROLE_ACCESS**.

When generating a message, the *linelen* member of the **mparse_message** structure may be set to the desired maximum line length for body content. It is ignored when not generating message content.

Default operation of **mparse** is to pass the message at its input to its output, emitting lines documenting RFC violations in input header fields. (Technically, these lines are written to a separate *stdio FILE*, though it may refer to the same file or standard output stream.)

The **canonicalize** flag, when set, causes protocol "names" (RFC 1958) to be presented with canonical capitalization as shown in the relevant RFC. This flag should only be set in **MPARSE_PRIMARY_ROLE_GENERATION** *context* or with the *context* flag **MPARSE_SECONDARY_ROLE_REPAIR**, but **mparse** does not enforce correspondence between *canonicalize* and *context*.

The **byte_stuff** and **byte_unstuff** flags control processing according to the model presented earlier.

RFC 3692 requires that experimental and private-use names not be recognized by default. If the **experimental** flag is non-zero, experimental and private-use names can be recognized via the hooks provided for extension names (see below). By default (in accordance with RFC 3692) these are not recognized.

ldebug and **gdebug** will enable verbose debugging output for the lexical analyzer and parser, respectively. Changes made to the **ldebug** flag after one of the **mparse** function is called will have no effect on lexical analyzer debugging output. To change lexical analyzer debugging output after **mparse** has been called, the **m_flex_debug** member of the structure pointed to by **r_flex** (see below) must be modified via the **flex** function **mparse_set_debug** (refer to the flex documentation for **yyset_debug** for details).

edebbug, **hdebug**, **pdebug**, **sdebug**, and **tdebug** control debugging output associated with allocated memory for the structures used by **mparse**. **Mparse** itself has been tested to ensure that there are no memory leaks, but user code that calls allocation and/or deallocation functions should be tested.

If the flag **header_only** is set, body text is not parsed; processing ends after the header fields. This mode

may be useful if only header content is of interest. This flag should only be set in `MPARSE_PRIMARY_ROLE_ACCESS` *context*, but **mparse** does not enforce correspondence between *header_only* and *context*.

suppress_errors causes generation of X-Err and X-NG header fields to be suppressed from output.

The **suppress_warnings** flag will cause warnings about context-dependent constructs to be suppressed from output.

The **no_copy** flag suppresses the normal echoing of the input message to the stream pointed to by **mout**.

While echoing of input is sent (unless suppressed) to **mout**, error and warning messages are sent to the **merr** stream. If **merr** is not initialized before one of the **mparse** functions is called, it is set to the standard error stream *stderr*. It is possible to set **merr** to the same stream as **mout**, however that may technically violate some RFCs which prohibit adding header fields in some contexts. Debugging output and miscellaneous error messages are always sent to *stderr*.

When reading input from a stream associated with a socket connection (e.g. to a POP or NNTP server), it may be desirable to set a time limit for input operations. If blocking input is used, a dropped connection may result in the process hanging, waiting for more input. If non-blocking input is used (e.g. via the **O_NONBLOCK** flag used with *fcntl(2)*), **EOF** may be returned if there are network delays.

Mparse provides for a timeout for input. If **timeout** is a positive non-zero value, read operations that return EOF after a partial line, as may be the case when there are network delays and **O_NONBLOCK** is used, will cause **mparse** to check for availability of more input for up to the number of seconds (and partial seconds) given by **timeout**. The **ioerr** flag described earlier will be set if there is no input within the specified timeout window.

Lexical Analysis

As an input message is processed, the first step is breaking the input stream into tokens. These tokens correspond to a piece of input text which has some significance in the message formats.

Some information is stored with each lexical token returned to the parser by the lexical analyzer. This information is stored in a **mparse_token** structure, defined in the header file **mparse.h**:

member (struct mparse_token)	description
char *tok;	allocated copy of token text
size_t len;	length (not including terminating '\0' (len = 1 for a MPARSE_TOKEN_NUL token))
int col;	input stream starting column (0-based) of token (== line length for MPARSE_TOKEN_CRLF or MPARSE_TOKEN_EOH)
int type;	token type, as defined in mparse.tab.h
int val;	value associated with token, if any
struct mparse_token *next;	singly-linked list link to next lexical token in logical construct
struct mparse_token *next2;	forward link of doubly-linked list link; all tokens (to MPARSE_TOKEN_EOH in fields, to end in body)
struct mparse_token *prev2;	reverse link of doubly-linked list of tokens in field or body
struct mparse_token *trailer;	trailing CFWS after token
struct mparse_token *close;	matching close delimiter (double quote, right parentheses, right bracket, '>', semi-colon terminating group, etc.)
struct mparse_list *list;	for list navigation
struct mparse_field *field;	pointer to enclosing field structure
struct mparse_error *error;	error(s) for this token
struct mparse_parameter *parameter;	MIME-parameters parameter if token is in a parameter
void *userptr;	user pointer; not set or used by parser, may be used to associate data with token
unsigned int address_has_comment : 1;	address or mailbox containing token has or is adjacent to a comment
unsigned int contains_group : 1;	address or address list contains a group
unsigned int has_cfws : 1;	angle-addr (or addr-spec in Received field for component) has internal CFWS
unsigned int has_route : 1;	angle-addr specified with route (not valid as msg-id)
unsigned int in_phrase : 1;	in phrase (RFC 2047 encoding permitted)
unsigned int is_comment : 1;	token is part of a comment (including parentheses)
unsigned int is_dlit : 1;	token is part of the interior of a domain literal
unsigned int is_quoted : 1;	token is part of the interior of a quoted string
unsigned int is_route : 1;	token is in obsolete source route
unsigned int is_uctext : 1;	token is part of unstructured text
unsigned int non_ascii:1;	token contains a non-ascii octet
unsigned int fold_val : 3;	folding preference 0 - 7

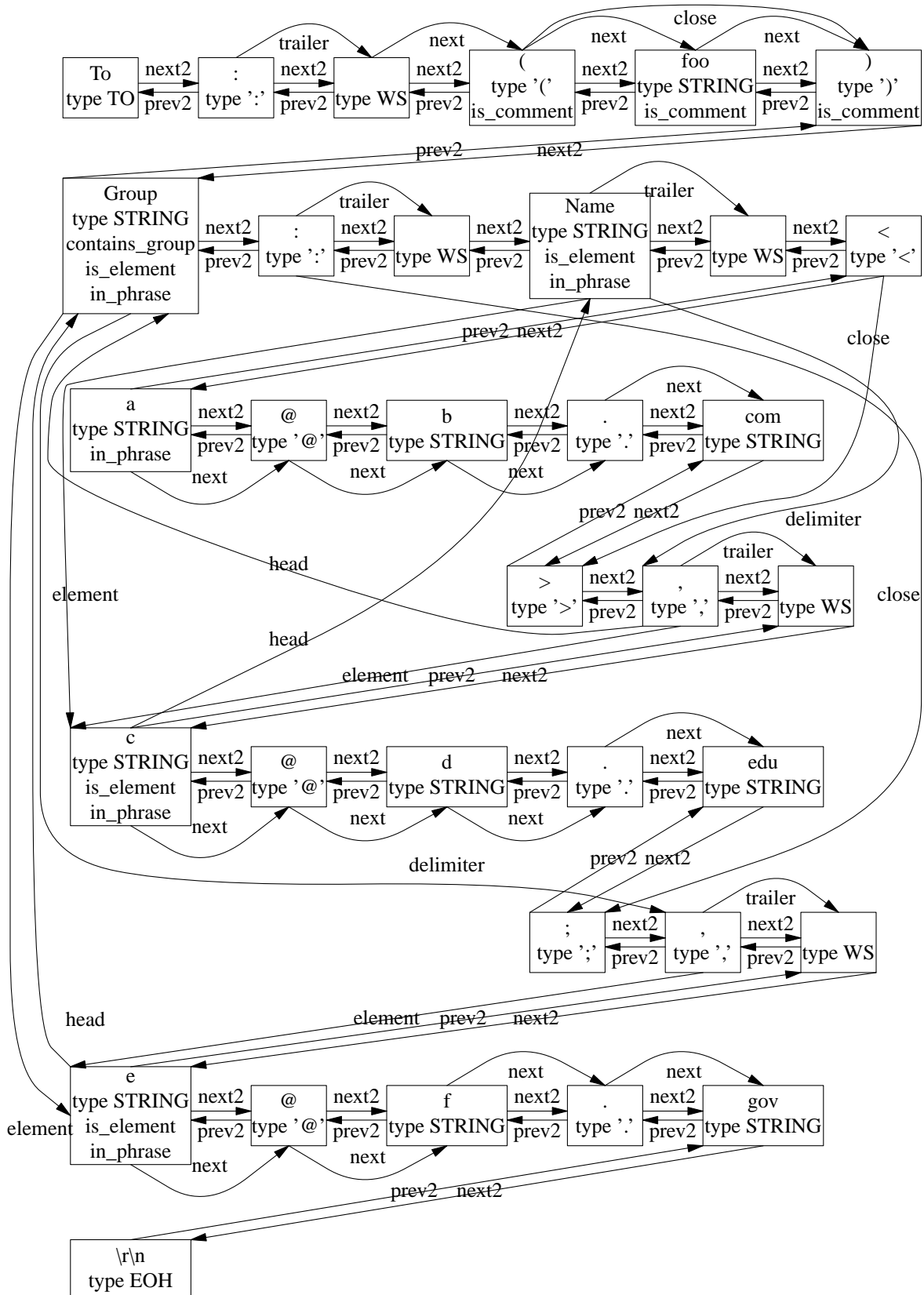
The **list** member of the **mparse_token** structure points to a **mparse_list** structure if the token is part of a list construct:

member (struct mparse_list)	description
struct mparse_token *head;	token at head of list
struct mparse_token *delimiter;	next delimiter separating list elements
struct mparse_token *element;	next list element token
struct mparse_token *next;	token following list end
struct mparse_list *sublist;	for nested lists
void *aux;	additional data (internal use)
int val;	additional data (internal use)
unsigned int nelements;	number of elements in list (valid only in head)
unsigned int ndelimiters;	number of delimiters in list (valid only in head)
unsigned int is_element:1;	this list item is a list element
unsigned int is_empty:1;	this list item is a delimiter with no corresponding list element

The use of the pointers and flags is illustrated by an example of a To field. The header field line:

To: (foo)Group: Name <a@b.com>, c@d.edu;, e@f.gov

is parsed into the following tokens and a rather complex structure of pointers:



This To field consists of a list of RFC 2822 **addresses**. They are the named **group** *Group* and the **mailbox** *e@f.gov*. The named **group** consists of a list of **mailboxes**, *<a@b.com>* and *c@d.edu*. Navigating these lists is simple; follow the **element** pointers in the **mparse_list** structure. Simple constructs such as an RFC 2822 **addr-spec** are linked using the **next** pointers. Comments, line folding, and whitespace (CFWS) are also linked together with the **next** pointers in structured fields, but are isolated from other tokens by being linked via the **trailer** pointers; the one exception being whitespace in an RFC 2822 **phrase**, which is linked via the **next** pointers with the other parts of the **phrase**.

Like the **mparse_message** structure, the **mparse_token** structure provides a user pointer which may be used to store arbitrary application data associated with a token.

Fields and Body Content

An additional structure is used to hold field lines and to hold the body section. This **mparse_field** structure also holds a user pointer, several flags, a pointer to the token list, and pointers to other **mparse_field** structures to form a doubly-linked list. There is also a pointer to a collection of useful field characteristics (unused for body sections).

member (struct mparse_field)	description
struct mparse_token *tokens;	first token in logical line
struct mparse_token *last_token;	most recently encountered token (used internally)
struct mparse_field *prev;	doubly-linked list link to previous field or body section "line"
struct mparse_field *next;	doubly-linked list link to next field or body section "line"
const struct mparse_field_state *state;	pointer to field characteristics
struct mparse_entity *entity;	pointer to enclosing entity structure
void *userptr;	user pointer; not set or used by parser, may be used to associate data with field
struct mparse_error *error;	error(s) for this field (entity type, etc.)
int linelen;	line length (max seen in body)
time_t dt;	date-time stamp value as time_t
int tokenpos;	token start column (for error messages)
unsigned int bit8 : 1;	found a non-ASCII character (used internally)
unsigned int lonecr : 1;	found a lone CR (used internally)
unsigned int lonelf : 1;	found a lone LF (used internally)
unsigned int nul : 1;	found an ASCII NUL (used internally)
unsigned int wsonly : 1;	found a whitespace-only continuation line (used internally)
unsigned int token_errors : 1;	some field token has an error (used internally)
unsigned int token_warnings : 1;	set if any token warnings (used internally)

The **mparse_field** structure is defined in the header file **mparse.h**.

Errors and Warnings

Mparse detects errors and potential errors (i.e. warnings) during parsing and stores relevant information which identifies the nature of the error. Error information is stored in a structure which contains the following members:

member (struct mparse_error)	description
struct mparse_error *prev;	doubly-linked circular list link to prev error struct
struct mparse_error *next;	doubly-linked circular list link to next error struct
struct mparse_token *token;	token responsible for error
struct mparse_field *field;	field responsible for error
struct mparse_entity *entity;	entity responsible for error
const struct mparse_charset *cs;	charset of error message
struct mparse_protocol_status *status;	head of singly-linked list of transfer/access protocol RFCs and relevant status reply codes
char *str;	reference string for error/warning
int rfc;	for sorting error messages by rfc number
const char *sect;	reference section string
int msgstr_index;	partial index into language-dependent error message string array
int count;	optional count for error message
int sufstr_index;	partial index into language-dependent suffix string array
int type;	MPARSE_REQUIREMENTS_MUST_NOT ... MPARSE_REQUIREMENTS_MUST
int val;	internal use (e.g. alternate value for repair)
int len;	length of str
unsigned int errors;	bitwise OR of MPARSE_ERR_* values
unsigned int remedies;	bitwise OR of MPARSE_FIX_* and/or MPARSE_FUBAR values

The **next** and **prev** pointers provide a doubly-linked circular list of **mparse_error** structures (more than one error message might be associated with an input error).

Errors might be associated with an input token, a field (or message body), or with the particular chunk of a message which is referenced by an **mparse_entity** structure (described below); there is provision in the **mparse_error** structure for pointers back to the relevant offending item.

A description of the error and the relevant standards documents is provided by the **str** member, whose length in bytes is given by **len**.

RFCs generally provide for a few categories of specifications: some things are expressly forbidden, usually indicated in the RFC text by a "MUST NOT" statement. Conversely, absolute requirements are usually specified with a "MUST" statement. Between these extremes, a strong recommendation which falls short of an absolute requirement is generally indicated by a "SHOULD" statement or "SHOULD NOT" statement. Relevant terms are defined in RFC 2119. The **type** member records the relevant category for the error or warning.

The **errors** member is an unsigned integer consisting of 1-bit flags which indicate the detected error conditions. The flags are determined by the requirements, prohibitions, and recommendations given in the RFCs relevant to text messages.

When generating message components, some types of errors can be automatically corrected (errors found during parsing are not corrected, as that would change the message). Details of the type of correction to be applied and any corresponding replacement value vary with the nature of the error. These are recorded in the **remedies** and **val** members.

While the **rfc** structure member records the applicable message format RFC corresponding to an error, there may be separate transfer protocol RFCs which have provision for a specific status response code or extended status response. The **status** structure member is a pointer to the head of a linked list of structures which record relevant status and extended status response values for various transfer protocols. The **mparse_protocol_status** structure is described below.

The function

```
void mparse_count_errors(const struct mparse_message *, const struct mparse_token *, unsigned int *, unsigned int *, unsigned int *, unsigned int *, unsigned int *, unsigned int *, unsigned int *, unsigned int *, unsigned int);
```

may be called to return total counts of errors and warnings. The first and second arguments point to a **mparse_message** structure and a **mparse_token** structure which is either the token of interest or any **mparse_token** structure in the **field**, **entity**, or **message** of interest. The error and warning counts are returned in unsigned integers pointed to by the third through tenth arguments. These consist of four pairs; hard errors and warnings for each of four categories: RFCs in effect per the message **modes**, RFCs not in effect, internet drafts supported by **mparse**, and miscellaneous non-RFC issues. The eleventh argument is comprised of bitwise ORed *MPARSE_COUNT_* macros defined in the header file **mparse.h**. The twelfth argument is comprised of bitwise ORed *MPARSE_ERR_* macros defined in the header file **mparse.h**, and defines which error types are to be counted. A total count of all errors and warnings in a message may be obtained by:

```
unsigned int errcount, warncount, other_errs, other_warnings;
mparse_count_errors(message, message->top->fields->tokens, &errcount, &warncount,
    &other_errs, &other_warnings, &other_errs, &other_warnings,
    &other_errs, &other_warnings,
    MPARSE_COUNT_ALL | MPARSE_COUNT_TOKENS | MPARSE_COUNT_FIELDS |
    MPARSE_COUNT_ENTITIES | MPARSE_COUNT_INSTANCES, ~(0U));
```

which should be called from the *hook_end_of_message* application hook (described below). Types of errors or warnings may be ignored by passing a *NULL* pointer in place of the appropriate unsigned integer pointer. As in the example above, a single unsigned integer may be used to accumulate multiple types of errors or warnings.

As noted above, transfer protocols may provide for specific *mparse_status* or extended status responses when an error is detected in the message during transfer. The **mparse_protocol_status** structure associates status and extended status values with a transfer protocol RFC for a specific error. The structure has the following members:

member (struct mparse_protocol_status)	description
struct mparse_error *error;	pointer to associated error structure
struct mparse_protocol_status *next;	singly-linked list pointer to next structure
int rfc;	applicable transport protocol RFC
unsigned int status;	status code for protocol, if applicable
unsigned int es_class;	class for extended status (RFCs 2034, 3463), if applicable
unsigned int es_subject;	subject for extended status (RFCs 2034, 3463), if applicable
unsigned int es_detail;	detail for extended status (RFCs 2034, 3463), if applicable

The status or extended status values appropriate for a given transfer protocol RFC and error are obtained through use of several functions:

```
unsigned int mparse_status(struct mparse_protocol_status *p);
```

returns the **status** member of the **mparse_protocol_status** structure pointed to by *p*. It returns zero with *errno* set appropriately in the event of an error.

```
int mparse_extended_status_string(struct mparse_protocol_status *p, char *buf, size_t sz);
```

interpolates an extended status string into the character array pointed to by *buf*, and whose size is *sz*. It returns -1 on error with *errno* set appropriately. Otherwise, it returns the number of characters written to *buf* if that array is sufficiently large, or the minimum number of characters necessary to hold the extended status string plus a terminating `'\0'` character if *buf* is a *NULL* pointer or points to an array which is too small (as specified by *sz*).

If a specific error is known, the appropriate **mparse_protocol_status** structure may be found by walking the list pointed to by the *status* member of the **mparse_error** structure, looking for the **mparse_protocol_status** structure with the *rfc* member corresponding to the transfer protocol in question.

More likely a status or extended status response must be provided after a message or message component has been received and parsed.

```
struct mparse_protocol_status *mparse_protocol_status(struct mparse_message *m, struct mparse_entity *e, struct mparse_field *f, struct mparse_token *t, int rfc);
```

may be used to return a pointer to an appropriate **mparse_protocol_status** structure for a given message, entity, field, or token corresponding to a particular transfer protocol RFC. If a token *t* is specified, the corresponding structure for the most serious error associated with that token, its field, and its entity's field or body (as appropriate) is returned. If *t* is a *NULL* pointer, all tokens in the field, entity, or message are checked according to which of those pointers are not *NULL*. If a field *f* is specified, the structure corresponding to the most serious error associated with that field or any of its tokens is returned. If *t* and *f* are both *NULL* and *e* points to an **entity** structure, then the **mparse_protocol_status** structure corresponding to the most serious error in that entity, its fields or body, and all of the tokens therein is returned. Likewise, if all of the pointers except *m* are *NULL*, all entities, fields, bodies, and tokens in the message are considered. In each case, the structure corresponding to the specified transport protocol *rfc* is returned, if one exists. If there is no such structure (*e.g.* if there is no error), a *NULL* pointer is returned. A *NULL* pointer is also returned in case of an error (*e.g.* all pointers are *NULL*), however in that case *errno* will be set appropriately (N.B. *errno* is not altered if there is no error).

Application-specific errors may be set for tokens, fields, entity header, entity body, or overall entity by calling one of the following functions, each of which returns a pointer to the error structure which (if not *NULL* due to some error) has been attached to the corresponding token, field, etc.:

```
struct mparse_error *mparse_set_token_error(const struct mparse_message *, struct mparse_token *, unsigned int, unsigned int, int, const char *, int, int, int, int, int);
```

sets an **mparse_error** structure attached to the specified token in the specified message. The two unsigned integer arguments specify the specific error types and remedies. The remaining seven integer arguments specify the RFC number containing the relevant specification, a language-neutral section specification string, the requirement level (**MPARSE_REQUIREMENTS_MUST_NOT**, ... **MPARSE_REQUIREMENTS_MUST**), an index into the possibly language-dependent array of strings describing the error (*msg_lang.gperf* and *msg.h*), an optional count (use **MPARSE_ERROR_COUNT_NO_COUNT** to cause it to be ignored), an index into a possibly language-dependent array of strings used as a suffix to the count (*msg_lang.gperf* and *msg.h*; **MPARSE_SUFFIX_NO_SUFFIX** to ignore it), and an integer value which may be used in some error messages or in repairing some types of errors.

Sometimes there is a string associated with an error message which cannot be referenced conveniently via an index. In such cases, the function

```
struct mparse_error *mparse_set_token_error_string(const struct mparse_message *, struct mparse_token *, unsigned int, unsigned int, int, const char *, const char *, int, const char *, const struct mparse_charset *, int, const char *, int);
```

may be used. The initial arguments are the same as above, Either an RFC number can be provided, or if one of the symbolic constants in *mparse.h* is used, a string reference may be provided (it is ignored if the *rfc* number is not negative). Instead of indices for the description and suffix, character string pointers are provided by the caller. The character set should be specified. Integer arguments for requirement level, count, and optional value are used as above.

Corresponding functions for field errors are:

```
struct mparse_error *mparse_set_field_error_string(const struct mparse_message *, struct mparse_field *, unsigned int,
unsigned int, int, const char *, const char *, int, const char *, const struct mparse_charset *, int, const char *, int);
```

and

```
struct mparse_error *mparse_set_field_error(const struct mparse_message *, struct mparse_field *, unsigned int, unsigned
int, int, const char *, int, int, int, int, int);
```

Functions for entity-related errors are also provided; they do not need a message structure pointer:

```
struct mparse_error *mparse_set_entity_error_string(struct mparse_entity *, unsigned int, unsigned int, int, const char *,
const char *, int, const char *, const struct mparse_charset *, int, const char *, int);
```

```
struct mparse_error *mparse_set_entity_error(struct mparse_entity *, unsigned int, unsigned int, int, const char *, int, int, int,
int, int);
```

```
struct mparse_error *mparse_set_entity_body_error_string(struct mparse_entity *, unsigned int, unsigned int, int, const char
*, const char *, int, const char *, const struct mparse_charset *, int, const char *, int);
```

```
struct mparse_error *mparse_set_entity_body_error(struct mparse_entity *, unsigned int, unsigned int, int, const char *, int,
int, int, int, int);
```

```
struct mparse_error *mparse_set_entity_header_error_string(struct mparse_entity *, unsigned int, unsigned int, int, const
char *, const char *, int, const char *, const struct mparse_charset *, int, const char *, int);
```

```
struct mparse_error *mparse_set_entity_header_error(struct mparse_entity *, unsigned int, unsigned int, int, const char *,
int, int, int, int, int);
```

Accumulated error and warning messages may be handled by calling the following functions:

```
size_t mparse_token_error_messages(const struct mparse_token *, size_t (*)(const char *, size_t, va_list), ...);
```

```
size_t mparse_field_error_messages(const struct mparse_message *, size_t (*)(const char *, size_t, va_list), ...);
```

```
size_t mparse_entity_body_error_messages(const struct mparse_entity *, size_t (*)(const char *, size_t, va_list), ...);
```

```
size_t mparse_entity_header_error_messages(const struct mparse_entity *, size_t (*)(const char *, size_t, va_list), ...);
```

```
size_t mparse_missing_close_delimiter_error_messages(const struct mparse_entity *, size_t (*)(const char *, size_t, va_list),
...);
```

```
size_t mparse_missing_separator_error_messages(const struct mparse_entity *, size_t (*)(const char *, size_t, va_list), ...);
```

Each function handles error messages by calling a function with a string of known length for each error message, and with any supplied arguments. The called function takes a pointer to a string, the length of the string, and a variable number of arguments (passed from the supplied arguments as a *va_list*).

Mparse includes the function

```
size_t mparse_fwrite_wrapper(const char *, size_t, va_list);
```

which takes a *stdio FILE ** as an additional argument via the *va_list*. A typical call to output token error messages to *stderr* might be:

```
#include <stdio.h>
size_t i;
struct mparse_token *token;
/* ... */
i = mparse_token_error_messages(token, mparse_fwrite_wrapper, stderr);
```

Applications could supply alternative functions to support handling error messages via *syslog* or other methods.

Status and extended status may be set for errors by calling the following functions, passing pointers to the message structure and corresponding error structure:

```
int mparse_success_status(const struct mparse_message *, struct mparse_error *);
```

sets a success status code

```
int mparse_address_syntax_status(const struct mparse_message *, struct mparse_error *);
```

sets status for known protocols appropriately for an address syntax error,

```
int mparse_general_transient_status(const struct mparse_message *, struct mparse_error *);
```

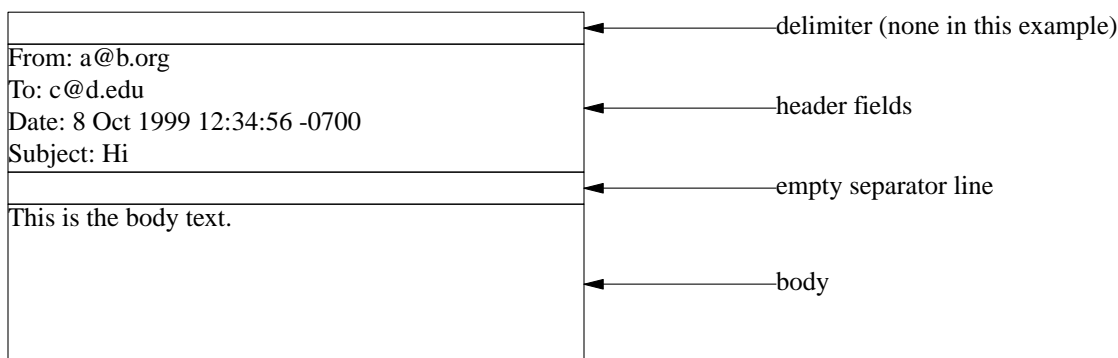
sets a transient status code for conditions that can be expected to be temporary, such as timeouts, temporary unavailability of resources, etc., and

```
int mparse_general_syntax_status(const struct mparse_message *, struct mparse_error *);
```

sets an appropriate status for general (not address-specific) syntax errors.

Message Parts

Mparse groups message content into chunks which may contain a boundary delimiter line, a set of header fields (possibly empty), a separator line, and body text. A simple message has no delimiter, does have header fields, may have body text, and has a separator line if there is body text (and might or might not have such a line if there is no body). Such a structure is sufficient to hold an entire simple message. It can be represented thus:



MIME messages are represented as linked collections of these structures. A MIME multipart message consists of a first section which has no delimiter, has regular and MIME header fields, may have body text (called a *preamble*), and has an empty separator line if there is a preamble, but might not have one if there is no preamble. Following that initial section comes one or more encapsulated parts, each of which has a delimiter line, may have MIME header fields, has a mandatory separator line, and has some body text. A MIME multipart message ends with a final section which has a specially formatted close delimiter line, no header fields, may have body text (called an *epilogue*), and has an empty separator line if there is an epilogue, but might not have one if there is no epilogue.

The structures are linked together in two dimensions using left (previous), right (next), up (parent), and down (child) pointers.

Many of the **mparse** functions take a pointer to a structure, called an **mparse_entity** structure, which is the structure described above. The data and links which are held in the **mparse_entity** structure are:

member (struct mparse_entity)	description
void *userptr;	user pointer; not set or used by parser, may be used by applications to pass data to hooks
struct mparse_message *message;	message enclosing entity
struct mparse_token *delimiter;	delimiter token (beginning of text associated with entity section)
struct mparse_field *fields;	message or MIME entity or MDN or DSN fields
struct mparse_field *last_field;	last field seen so far
struct mparse_token *separator;	CRLF token separating header from body, unless missing (e.g. in a malformed message)
struct mparse_field *body;	body section (or preamble, or epilogue, or phantom body)
struct mparse_field *last_body;	last body "line" seen so far
struct mparse_entity *parent;	points to enclosing entity in a MIME multipart entity; null pointer in top-level message
struct mparse_entity *child;	points to first (in order of appearance unless modified by an RFC 2387 <i>start</i> parameter) enclosed MIME multipart entity
struct mparse_entity *next_sibling;	points to next MIME multipart entity enclosed by same parent.
struct mparse_entity *prev_sibling;	points to previous MIME multipart entity enclosed by same parent.
struct mparse_cache cache;	MIME information cache
unsigned int mimepart;	mime section number; 0 for preamble, then incremented at each delimiter, 0 for epilogue
struct mparse_error *field_errors;	missing fields, etc.
struct mparse_error *body_errors;	excessively long lines, illegal content, etc.
unsigned int h_processed:1;	header_end called (used internally)
unsigned int b_processed:1;	body_end called (used internally)

member (struct mparse_cache)	description
struct mparse_token *content_type;	MIME content type
const struct mparse_type *media_type;	MIME media type information
struct mparse_token *content_subtype;	MIME content subtype
const struct mparse_subtype *subtype;	MIME media subtype information
struct mparse_token *content_parameters;	MIME content-type parameters
struct mparse_token *content_id;	MIME content-id
struct mparse_token *description;	MIME Content-Description
struct mparse_token *content_disposition;	MIME Content-Disposition disposition type
struct mparse_disposition *disposition;	MIME disposition information
struct mparse_token *disposition_parameters;	MIME Content-Disposition parameters
struct mparse_token *duration;	MIME Content-Duration
struct mparse_token *encoding;	MIME content-transfer-encoding
const struct mparse_encoding *enc;	MIME encoding/domain information
struct mparse_token *languages;	MIME Content-Language
struct mparse_token *location;	MIME Content-Location URI
struct mparse_token *md5;	MIME Content-MD5

The details of the structures pointed to by these members are discussed below.

The **userptr** may be used to point to arbitrary data needed by the application.

A pointer to the enclosing **mparse_message** structure is held in the **message** member.

The **last_field** member points to the most-recently added field, and may be used within user functions (discussed below) which are called when a field is recognized in the input stream.

A similar member is the **last_body** pointer, however that is only used internally when generating body content in a piecemeal manner; normal parsing of a complete message produces a single **body** section stored in a **mparse_field** structure.

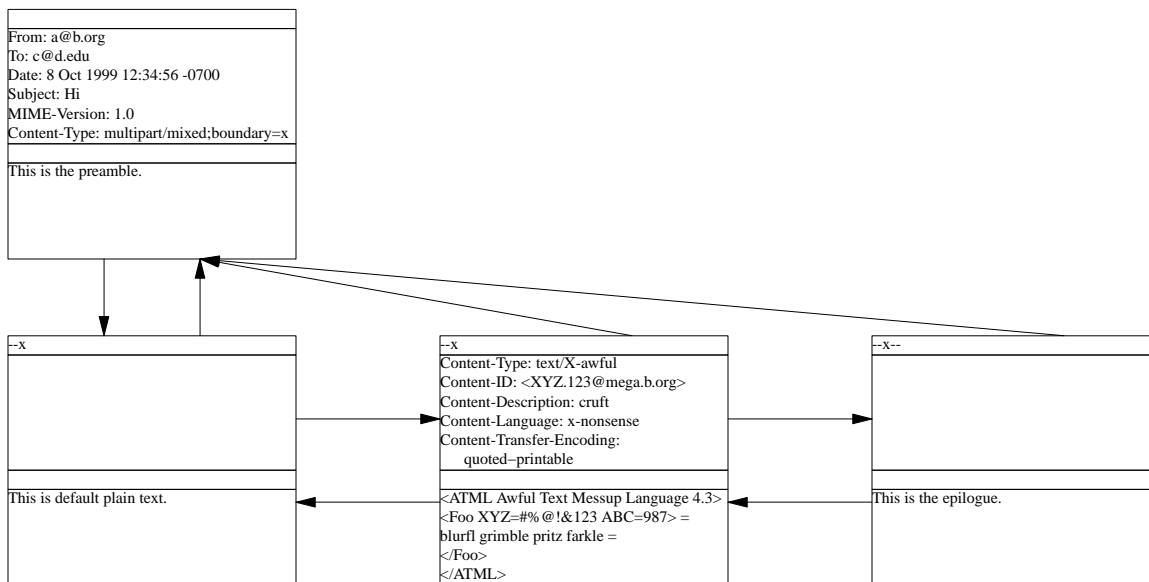
Some MIME information contained in the message or MIME-part header fields (or default information in the absence of a relevant MIME field) is cached in the **mparse_cache** structure. Examples include the **content_type**, **content_subtype**, **media_type**, and **subtype** members. **content_type** and **content_subtype**

point to the field tokens (structure details below) if a MIME Content-Type field is present. The **media_type** and **media_subtype** members point to structures described above, which are determined from the MIME Content-Type field if present, and are set to appropriate default values otherwise. This is an important distinction between the *content_* and *media_* member types; the *content_* pointers may be *NULL* (in the absence of a MIME Content-Type field) or may point to tokens with non-standard type or subtype, whereas the *media_* pointers (once initialized by *mparse*) will point to a structure for the corresponding standard type and subtype (defaults in the absence of a Content-Type field and applicable values used when non-standard values are provided in a Content-Type field are standardized by RFCs 2045 and 2046).

The difference between the struct *mparse_token* pointer **encoding** and the other struct pointer, **enc**, for the MIME media encoding information is that the struct *mparse_token* pointer points to the token in the input message header field, while the other pointer points to information which may be a default (in the absence of an explicit field specification, in which case the struct *mparse_token* pointer is a *NULL* pointer), or it may be set to a different type (e.g. if the specified type is unrecognized) according to the rules in RFCs 2045 and 2046.

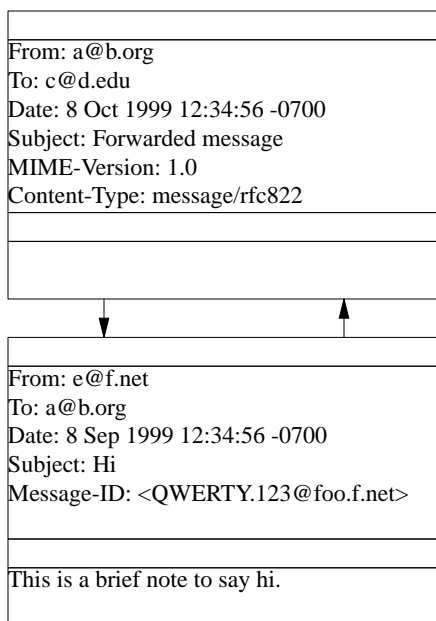
There are two pointers to lists of **mparse_error** structures provided in the **mparse_entity** structure: **field_errors** and **body_errors**. These point to errors for the message chunk field section and body, respectively, where the errors in question are not specifically tied to a particular input token or field.

A relatively simple MIME multipart message might look like this:



Mparse will construct such a linked structure as it parses a message, or functions can be called to build such a structure for generating a message. In the latter case, *mparse* takes care of some of the gory details of building a MIME message so that the programmer need not worry too much about those details, but can instead concentrate on the content.

In addition to multipart composite MIME entities, MIME message composite media types are provided for encapsulating messages. As already mentioned, several of these have unusual characteristics. Perhaps the simplest MIME message media type is the `message/rfc822` type defined in RFC 2046. The encapsulation consists of MIME-part header fields followed by an empty separator line. There is no delimiter and no body *per se*. Following the separator is the encapsulated message (header fields, separator, body). Note that the encapsulated message need not be a simple message; it may be a MIME message of arbitrary complexity. Because the structure discussed above does not provide for multiple sets of fields, the encapsulation links to a separate structure (or linked group of structures) for the encapsulated message. Here's a simple example:



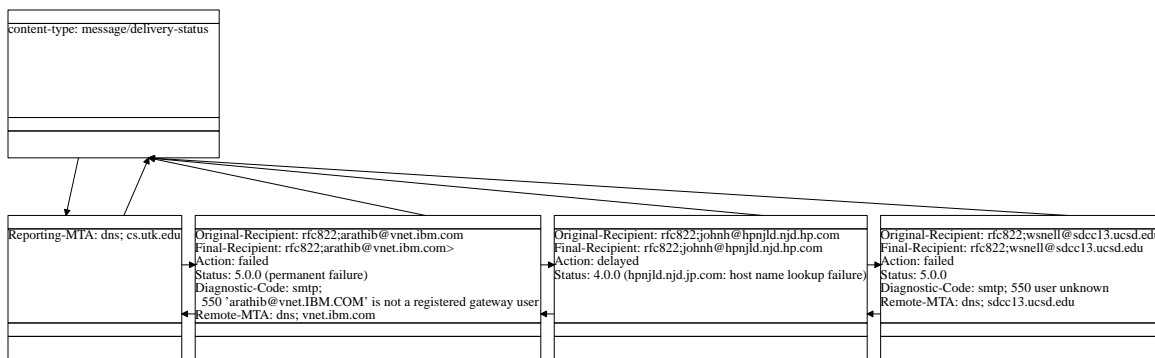
RFC 2046 also provides a `message/partial` media type. The first part of a series of message parts looks similar to the `message/rfc822`. Subsequent parts, however, do not have header fields associated with the encapsulated partial message (all of the header fields are in the first part) so the body of the encapsulated part is stored with the encapsulation (which, as noted above, does not have its own body).

`Message/external-body` is similar to `message/rfc822` except that the encapsulated body (called a phantom body) may contain instructions for automated retrieval of the external body. The structure is the same as used for `message/rfc822`.

As mentioned earlier, some message media types are peculiar because the encapsulated content does not begin with structured header fields as defined in RFCs 822 and 2822. Instead, the content is in a specific format which will need to be processed separately by applications due to the incompatibilities with RFCs 822 and 2822. Since the encapsulation does not have RFC 822/2822 header fields, it is stored with the encapsulation, just as with continuation portions of `message/partial`.

RFC 2298 (superseded by RFC 3798) defined a `message/disposition-notification` media type. The encapsulated entity (the disposition notification) consists entirely of structured fields. As with `message/rfc822`, the encapsulated entity is stored in a structure linked to the encapsulation. The structured fields are stored as fields.

The most complex message media type is the message/delivery-status type originally defined in RFC 1894 (currently defined in RFC 3464). The encapsulated entity consists of a series of groups of structured fields separated by empty lines. As noted above, the structure used does not provided for multiple sets of independent fields per structure, so these groups (per-message fields, and one or more groups of per-recipient fields) are each stored in a separate structure. The links between structures resemble those of a multipart entity; the per-message fields are held in the encapsulation's child structure, and the per-recipient fields are held in structures linked to the right of the per-message fields' structure. Here's an example (from RFC 3464, Multi-Recipient DSN example):



MIME Media Types and Subtypes

The media types registered by the Internet Assigned Numbers Authority (IANA) are recognized by **mparse**. A program may enumerate those media types (e.g. to create a menu) by calling the function

```
const struct mparse_type *mparse_media_type(int n);
```

with successive integer arguments beginning with 1. Each call will return a pointer to a structure (see below) which represents a media type; when all media types have been enumerated, **mparse_media_type** returns a *NULL* pointer.

Information regarding a registered media type tag can be obtained by calling:

```
const struct mparse_type *mparse_type_entry(const char *str, unsigned int len);
```

giving a pointer to the media type tag as **str** and its length (only the primary type tag) as **len**. The string pointed to by **str** need not be null-terminated: one could call **mparse_type_entry("message/rfc-822", 7U)** to obtain the information for the **message** media type. If the tag is not a registered type, a *NULL* pointer is returned. Otherwise, a pointer to the structure defined in **mparse.h** is returned. The members of that structure are:

member (struct mparse_type)	description
const char *type_name;	canonical type tag
unsigned int flags;	flags defined in mparse.h
const struct mparse_subtype *(*e)(int);	pointer to a function useful for enumerating subtypes of the type
const struct mparse_subtype *(*f)(const char *, unsigned int);	pointer to a function returning information about subtypes of the type

Likewise, information about subtypes may be obtained by calling the appropriate function returned in the *mparse_type* structure; they return a similar structure containing only the canonical subtype tag and flags. Note that the illegal "example" type will return a structure with *NULL* pointers for the subtype enumeration and information functions; there are no subtypes of the "example" type. User-defined types and subtypes can be supported via hooks provided for extension types and subtypes; these are application-provided functions which operate like the ones described above, but for the user-defined types (the extension subtype function also takes a pointer to the type). They are described below along with other application hooks.

In addition to accessing the media subtype enumeration and information functions via the **mparse_type** structure elements, it is possible to directly access those functions. For each media type except for the illegal "example" registered type, the name of the enumeration function is composed of the "mparse_" prefix,

the type tag, and by a "_subtype" suffix, and the information function is composed of the "mparse_" prefix, the type tag, and by a "_entry" suffix. So, for example, the functions for the **video** media type are **mparse_video_subtype** (for enumeration) and **mparse_video_entry**.

If the media type for an entity is not explicitly specified via a Content-Type field, there is a default type. Even if a type is specified via a Content-Type field, some other type might be in effect (e.g. if an unrecognized transfer coding is specified). The functions

```
const struct mparse_type *mparse_default_type(struct mparse_entity *, int);
```

and

```
const struct mparse_subtype *mparse_default_subtype(struct mparse_entity *, int);
```

return pointers to the default media type and subtype for the specified entity. if the integer argument is less than zero, indicating some error, the type and subtype returned correspond to application/octet-stream.

Application Hooks

Many of the application-defined functions take a pointer to the current **mparse_entity** structure as its first argument. Note that the pointer may point to a different location on different calls to a function; typically this happens as a result of a multipart MIME message. Functions called for fields may access the field via the **last_field** pointer in the **mparse_entity** structure.

Several of the application-defined functions take one or more pointers to a **struct mparse_field** defined in the header file **mparse.h**, or **struct mparse_token** which is defined in the header file **mparse.h**.

If the *userptr* should be copied when an **mparse_entity** structure is copied or allocated, or if it points to allocated storage which should be freed when an **mparse_entity** structure is freed, the appropriate functions for copying and freeing should be set in the user application function hooks. Likewise for **userptr** members of the **mparse_token** and **mparse_field** structures.

These function hooks are held in a structure pointed to by the *hooks* member of the **mparse_message** structure. It is defined in the header file **mparse.h** and contains:

function (member of struct mparse_hooks)	description
<code>void (*hook_accept_language)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Accept-Language field
<code>void (*hook_action)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Action field
<code>void (*hook_alt_recipient)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Alternate-Recipient field
<code>void (*hook_approved)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Approved field
<code>void (*hook_archive)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Archive field
<code>void (*hook_archived_at)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Archived-At field
<code>void (*hook_arrival_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Arrival-Date field
<code>void (*hook_autoforwarded)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Autoforwarded field
<code>void (*hook_autosubmitted)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Autosubmitted field
<code>void (*hook_auto_submitted)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Auto-Submitted field
<code>void (*hook_bad_field)(const struct mparse_field *);</code>	do something with bad mparse_field (use next2 pointers)
<code>void (*hook_bcc)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Bcc field
<code>void (*hook_bilateral_info)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Bilateral-Info field
<code>void (*hook_body_section)(const struct mparse_entity *);</code>	do something with body section (Use next2 pointers)
<code>void (*hook_body_section_end)(const struct mparse_entity *);</code>	do something after body section
<code>void (*hook_body_section_end_of_MIME_fields)(const struct mparse_entity *);</code>	do something at end of body section MIME fields
<code>void (*hook_body_section_end_of_fields)(const struct mparse_entity *);</code>	do something at end of body section fields
<code>void (*hook_body_section_start)(const struct mparse_entity *);</code>	prepare for body section (may include fields)
<code>void (*hook_cancel)(int, const struct mparse_field *, const struct mparse_token *);</code>	process cancel control message
<code>void (*hook_caller_id)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	process Caller-ID field

function (member of struct mparse_hooks)	description
<code>void (*hook_caller_name)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Caller-Name field
<code>void (*hook_cc)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Cc field
<code>void (*hook_checkgroups)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process checkgroups control message
<code>void (*hook_comments)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Comments field
<code>void (*hook_content_alternative)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content alternative filter
<code>void (*hook_content_base)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content base
<code>void (*hook_content_description)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content description
<code>void (*hook_content_disposition)(int, const struct mparse_field *);</code>	process content disposition
<code>void (*hook_content_duration)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content duration
<code>void (*hook_content_features)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content features
<code>void (*hook_content_id)(int, const struct mparse_field *);</code>	process content ID
<code>void (*hook_content_language)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content language
<code>void (*hook_content_location)(int, const struct mparse_field *, const struct mparse_token *);</code>	process content location
<code>void (*hook_content_md5)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Content-MD5 field
<code>void (*hook_content_type)(int, const struct mparse_field *, struct mparse_token *, struct mparse_token *, struct mparse_token *, struct mparse_token *);</code>	process Content-Type field. RFC 1049 Content-Type fields may be distinguished from MIME Content-Type fields as the type of the second <code>struct mparse_token *</code> in the latter is <code>'/'</code> (i.e. a slash character), but not in the former (where it may be a <code>NULL</code> pointer or may point to a token with some other value. MIME media type, subtype, and <code>mparse_parameters</code> are cached; RFC 1049 Content-Type field body content is not cached.
<code>void (*hook_conv_w_loss)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Conversion-With-Loss field
<code>void (*hook_conversion)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Conversion field
<code>void *(*hook_copy_token_user_data)(const struct mparse_token *);</code>	return a pointer to (an allocated, if <code>hook_free_token_userptr</code> is used) copy of token userptr
<code>void *(*hook_copy_entity_user_data)(const struct mparse_entity *);</code>	return a pointer to (an allocated, if userptr is to be freed) copy of userptr
<code>void *(*hook_copy_field_user_data)(const struct mparse_field *);</code>	return a pointer to (an allocated, if userptr is to be freed) copy of userptr
<code>void (*hook_cte)(int, const struct mparse_field *);</code>	process content-transfer-encoding
<code>void (*hook_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Date field
<code>void (*hook_date_received)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Date-Received field
<code>void (*hook_def_delivery)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Deferred-Delivery field
<code>void (*hook_delivery_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Delivery-Date field
<code>void (*hook_diag_code)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Diagnostic-Code field
<code>void (*hook_discarded_x400_ipms_extensions)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Discarded-X400-IPMS-Extensions field
<code>void (*hook_discarded_x400_mts_extensions)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Discarded-X400-MTS-Extensions field
<code>void (*hook_disclose_recip)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Disclose-Recipients field
<code>void (*hook_disposition)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	process disposition
<code>void (*hook_disposition_notification_opts)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Disposition-Notification-Options field
<code>void (*hook_disposition_notification_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Disposition-Notification-To field
<code>void (*hook_distribution)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Distribution field
<code>int (*hook_distribution_validate)(const struct mparse_field *, const struct mparse_token *);</code>	validate (0: OK) distribution name
<code>void (*hook_dl_exp_history)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with DL-Expansion-History field
<code>void (*hook_drc_bi)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Delivery-Report-Content-Billing-Information field
<code>void (*hook_drc_it)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Delivery-Report-Content-Intermediate-Trace field
<code>void (*hook_drc_rri)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Delivery-Report-Content-Reported-Recipient-Info field

function (member of struct mparse_hooks)	description
<code>void (*hook_drc_ua_c_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Delivery-Report-Content-UA-Content-ID field
<code>void (*hook_drc_original)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Delivery-Report-Content-Original field
<code>void (*hook_dsn_gateway)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with DSN-Gateway field
<code>void (*hook_encoding)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Encoding field
<code>void (*hook_encrypted)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process (obsolete) Encrypted field
<code>int (*hook_end_of_message)(const struct mparse_message *);</code>	do something at end of message, return zero for normal mparse_status, non-zero to indicate an abnormal condition
<code>void (*hook_end_of_mdn_fields)(const struct mparse_entity *);</code>	do something at end of MDN
<code>void (*hook_end_of_per_message_fields)(const struct mparse_entity *);</code>	do something at end of DSN per-message fields
<code>void (*hook_end_of_per_recipient_fields)(const struct mparse_entity *);</code>	do something at end of DSN per-recipient fields
<code>void (*hook_error)(int, const struct mparse_field *, const struct mparse_token *);</code>	process RFC 2298, 3798 Error field
<code>void (*hook_error_message)(const struct mparse_message *, const char *);</code>	do something with error message before output
<code>void (*hook_expires)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process Expires field
<code>void (*hook_extension)(int, const struct mparse_field *, const struct mparse_token *);</code>	process extension field (contents)
<code>const char *(*hook_extension_MTA_name_type)(const char *, unsigned int);</code>	support x- and unregistered MTA-name-types
<code>const struct mparse_name_val *(*hook_extension_access_type)(const char *, unsigned int);</code>	support x- and unregistered access-type for message/external-body
<code>const char *(*hook_extension_address_type)(const char *, unsigned int);</code>	support x- and unregistered address-types
<code>const struct mparse_name_val *(*hook_extension_auto_submitted)(const char *, unsigned int);</code>	support x- and unregistered auto-submitted tokens
<code>const struct mparse_charset *(*hook_extension_charset)(const char *, unsigned int);</code>	support x- unregistered charsets
<code>const char *(*hook_extension_diagnostic_type)(const char *, unsigned int);</code>	support x- and unregistered diagnostic-types
<code>const struct mparse_disposition *(*hook_extension_disposition)(const char *, unsigned int);</code>	support x- and unregistered dispositions
<code>const struct mparse_name_val *(*hook_extension_disposition_modifier)(const char *, unsigned int);</code>	support x- and unregistered disposition modifiers
<code>const struct mparse_name_val *(*hook_extension_disposition_notification_option)(const char *, unsigned int);</code>	support x- and unregistered disposition notification options
<code>const struct mparse_language *(*hook_extension_language)(const char *, unsigned int);</code>	support x- and unregistered languages
<code>const struct mparse_subtype *(*hook_extension_media_subtype)(const struct mparse_type *, const char *, unsigned int);</code>	support x- and unregistered media type subtypes
<code>const struct mparse_type *(*hook_extension_media_type)(const char *, unsigned int);</code>	support x- and unregistered media types
<code>const char *(*hook_extension_numbering_plan)(const char *, unsigned int);</code>	support x- and unregistered numbering plans
<code>const struct mparse_encoding *(*hook_extension_transfer_encoding)(const char *, unsigned int);</code>	support x- and unregistered transfer encodings
<code>void (*hook_failure)(int, const struct mparse_field *, const struct mparse_token *);</code>	process RFC 2298, 3798 Failure field
<code>void (*hook_fcc)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Fcc field
<code>void (*hook_final_log_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Final-Log-ID field
<code>void (*hook_final_recipient)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process RFC 2298, 3798 final-recipient field
<code>void (*hook_followup_to)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Followup-To field
<code>void (*hook_free_entity_userptr)(const struct mparse_entity *, void *);</code>	free allocated userptr in entity structure
<code>void (*hook_free_field_userptr)(const struct mparse_field *, void *);</code>	free allocated userptr in field structure
<code>void (*hook_free_token_userptr)(const struct mparse_token *, void *);</code>	free allocated userptr in token structure
<code>void (*hook_from)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with From field
<code>void (*hook_gen_del_repl)(int, const struct mparse_field *);</code>	do something with Generate-Delivery-Report field
<code>void (*hook_field)(const struct mparse_field *);</code>	do something with logical field line (use next2 links)
<code>void (*hook_ihave)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process ihave control message (optional message-id series, system name)
<code>void (*hook_illegal_field)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Illegal-Field field
<code>void (*hook_illegal_object)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Illegal-Object field
<code>void (*hook_importance)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Importance field
<code>void (*hook_in_reply_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	process In-Reply-To field
<code>void (*hook_incomplete_copy)(int, const struct mparse_field *);</code>	do something with Incomplete-Copy field
<code>void (*hook_injection_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Injection-Date field

function (member of struct mparse_hooks)	description
<code>void (*hook_injector_info)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Injector-Info field
<code>void (*hook_keywords)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Keywords field
<code>void (*hook_last_attempt_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Last-Attempt-Date field
<code>void (*hook_latest_del_time)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Latest-Delivery-Time field
<code>void (*hook_lines)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Lines field
<code>void (*hook_list_archive)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process List-Archive field
<code>void (*hook_list_help)(int, const struct mparse_field *, const struct mparse_token *);</code>	process List-Help field
<code>void (*hook_list_id)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process List-ID field
<code>void (*hook_list_owner)(int, const struct mparse_field *, const struct mparse_token *);</code>	process List-Owner field
<code>void (*hook_list_post)(int, const struct mparse_field *, const struct mparse_token *);</code>	process List-Post field
<code>void (*hook_list_subscribe)(int, const struct mparse_field *, const struct mparse_token *);</code>	process List-Subscribe field
<code>void (*hook_list_unsubscribe)(int, const struct mparse_field *, const struct mparse_token *);</code>	process List-Unsubscribe field
<code>void (*hook_mail_from)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Mail-From field
<code>int (*hook_mailbox_domain_validate)(const struct mparse_field *, const struct mparse_token *);</code>	validate (0: OK) domain token list
<code>int (*hook_mailbox_validate)(const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	validate (0: OK) mailbox (local-part, domain)
<code>void (*hook_mdn_gateway)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process MDN-Gateway field
<code>void (*hook_media_accept_features)(int, const struct mparse_field *, const struct mparse_token *);</code>	process media-accept-features
<code>void (*hook_message_context)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Message-Context
<code>void (*hook_message_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Message-ID
<code>void (*hook_message_type)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Message-Type field
<code>void (*hook_mime_close_delimiter)(const struct mparse_entity *, const struct mparse_token *);</code>	process MIME close delimiter boundary
<code>void (*hook_mime_delimiter)(const struct mparse_entity *, const struct mparse_token *);</code>	process MIME delimiter boundary
<code>void (*hook_mime_encapsulation)(const struct mparse_entity *);</code>	set up child entity (hooks, etc.)
<code>void (*hook_mime_external_body)(const struct mparse_entity *);</code>	process (e.g. fetch, display) MIME external-body message
<code>void (*hook_mime_multipart)(const struct mparse_entity *);</code>	set up child entity (hooks, etc.)
<code>void (*hook_mime_version)(int, const struct mparse_field *, const struct mparse_token *);</code>	process MIME-version field
<code>void (*hook_mvgroup)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	process mvgroup control message (newsgroup names)
<code>void (*hook_newgroup)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process newgroup control message (newsgroup name, optional "moderated" keyword)
<code>void (*hook_newsgroups)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Newsgroups field
<code>int (*hook_newsgroup_validate)(const struct mparse_field *, const struct mparse_token *);</code>	validate (0: OK) newsgroup name
<code>void (*hook_obsoletes)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Obsoletes field
<code>void (*hook_organization)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Organization field
<code>void (*hook_orig_env_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Original-Envelope-ID field
<code>void (*hook_orig_ret_addr)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Originator-Return_Address field
<code>void (*hook_original_encoded_information_types)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Original-Encoded-Information-Types field
<code>void (*hook_original_message_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	process RFC 2298, 3798 original-message-id field
<code>void (*hook_original_recipient)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process RFC 2298, 3798 original-recipient field
<code>void (*hook_p1_content_type)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with P1-Content-Type field
<code>void (*hook_p1_message_id)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with P1-Message-ID field
<code>void (*hook_p1_recipient)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with P1-Recipient field
<code>void (*hook_path)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Path line

function (member of struct mparse_hooks)	description
<code>void (*hook_posting_version)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	process Posting-Version field
<code>void (*hook_prev_nondel_rep)(int, const struct mparse_field *);</code>	do something with Prevent-Nondelivery-Report field
<code>void (*hook_priority)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Priority field
<code>void (*hook_received)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Received field
<code>void (*hook_received_content_mic)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Received-content-MIC field
<code>int (*hook_received_by_domain_validate)(const struct mparse_field *, const struct mparse_token *);</code>	validate (0: OK) Received by domain
<code>int (*hook_received_from_domain_validate)(const struct mparse_field *, const struct mparse_token *);</code>	validate (0: OK) Received from domain
<code>void (*hook_rec_from_mta)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Received-From-MTA field
<code>void (*hook_references)(int, const struct mparse_field *, const struct mparse_token *);</code>	process References field
<code>void (*hook_relay_version)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	process Relay-Version field
<code>void (*hook_remote_mta)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Remote-MTA field
<code>void (*hook_reply_by)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Reply-By field
<code>void (*hook_reply_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Reply-To field
<code>void (*hook_reporting_mta)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Reporting-MTA field
<code>void (*hook_reporting_ua)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Reporting-UA field
<code>void (*hook_resent_bcc)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Resent-Bcc field
<code>void (*hook_resent_cc)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Resent-Cc field
<code>void (*hook_resent_date)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Resent-Date field
<code>void (*hook_resent_from)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Resent-From field
<code>void (*hook_resent_message_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Resent-Message-ID
<code>void (*hook_resent_reply_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with (deprecated) Resent-Reply-To field
<code>void (*hook_resent_sender)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Resent-Sender field
<code>void (*hook_resent_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Resent-To field
<code>void (*hook_return_path)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Return-Path field
<code>void (*hook_rmgroup)(int, const struct mparse_field *, const struct mparse_token *);</code>	process rmgroup control message (newsgroup name)
<code>void (*hook_sender)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Sender field
<code>void (*hook_sendme)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	process sendme control message (optional message-id series, system name)
<code>void (*hook_sendsys)(int, const struct mparse_field *);</code>	process sendsys control message
<code>void (*hook_sensitivity)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Sensitivity field
<code>void (*hook_separator)(const struct mparse_entity *);</code>	process separator between header fields and body section
<code>void (*hook_start_of_message)(const struct mparse_entity *);</code>	do something at start of message
<code>void (*hook_status)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Status field
<code>void (*hook_subject)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Subject field
<code>void (*hook_summary)(int, const struct mparse_field *, const struct mparse_token *);</code>	process Summary field
<code>void (*hook_supersedes)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Supersedes field
<code>void (*hook_to)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with To field
<code>void (*hook_ua_c_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with UA-Content-ID field
<code>void (*hook_undefined_control)(int, const struct mparse_field *, const struct mparse_token *);</code>	process undefined control message (mail to local "usenet" account?)
<code>void (*hook_user_agent)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *);</code>	do something with User-Agent field
<code>void (*hook_user_defined)(int, const struct mparse_field *, const struct mparse_token *);</code>	process user_defined (X-) mparse_field (contents)
<code>void (*hook_version)(int, const struct mparse_field *);</code>	process version control message
<code>void (*hook_warning)(int, const struct mparse_field *, const struct mparse_token *);</code>	process RFC 2298, 3798 Warning field
<code>void (*hook_will_retry_until)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with Will-Retry-Until field

function (member of struct <code>mparse_hooks</code>)	description
<code>void (*hook_x400_cont_id)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with X400-Content-Identifier field
<code>void (*hook_x400_cont_ret)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with X400-Content-Return field
<code>void (*hook_x400_content_type)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with X400-Content-Type field
<code>void (*hook_x400_mts_identifier)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with X400-MTS-Identifier field
<code>void (*hook_x400_originator)(int, const struct mparse_field *, const struct mparse_token *, int);</code>	do something with X400-Originator field
<code>void (*hook_x400_received)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with X400-Received field
<code>void (*hook_x400_recipients)(int, const struct mparse_field *, const struct mparse_token *, int);</code>	do something with X400-Recipients field
<code>void (*hook_x400_trace)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with X400-Trace field
<code>void (*hook_xref)(int, const struct mparse_field *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *, const struct mparse_token *);</code>	do something with Xref line
<code>void (*hook_x_archived_at)(int, const struct mparse_field *, const struct mparse_token *);</code>	do something with X-Archived-At field

Parser Initialization

The `mparse_message` structure should first be initialized to all zero values either by using `calloc(3)` or `memset(3)`. Flags, file pointers, and pointers to application-defined functions can then be set, prior to calling one of the `mparse` functions (described in detail below).

Message Parsing

The simplest call to `mparse` would be

```
mparse_parse(0, 0, 0);
```

which automatically allocates a `mparse_message` structure and associated `r_flex` structure, reads input from `stdin`, sends output to `stdout`, and error and warning output to `stderr`. Since no flags or function hooks are set in the allocated `mparse_message` structure, all processing is the default. In order to obtain behavior other than the default, the caller should provide a pointer to a `mparse_message` structure which has been initialized with the desired function hooks, modes, and flags. A typical method would be:

```
#include <mparse.h>

/* ... */
int ret;
struct mparse_message message;
struct mparse_debug debug;

memset(&message, 0, sizeof(struct mparse_message));
memset(&debug, 0, sizeof(struct mparse_debug));
message.dbg = &debug;
/* set function hooks */
/* set modes using mparse_add_mode(), etc. */
/* set flags, e.g.: */
message.no_copy = 1U;
ret = mparse_parse(&message, 0, 0);
```

Of course, it is also possible to pass other FILE pointers; this is done in the usual way by calling `fopen` or related functions. When called with an initialized `mparse_message` structure, `mparse_parse` will process input according to the flags, generate warning and error messages according to the modes, and call the specified functions at the appropriate points. Processing can be performed on-the-fly by functions called, or the calling application can defer processing until one or more of the `hook_end_of_...` functions is called, or a mix of both types of processing may be used.

It is also possible to parse a message held in a character array rather than a file; use

```
int mparse_parse_string(struct mparse_message *, const char *, FILE *);
```

Because C strings are terminated by an ASCII `NUL` character, `mparse_string` might fail if the message has an embedded ASCII `NUL`. In that case, one can use

```
int mparse_parse_buffer(struct mparse_message *, char *, unsigned int, FILE *);
```

where the message length in bytes is given by the unsigned integer argument. Refer to the *flex*

documentation for details on parsing input from such a *buffer*.

Token lists are generated for each field, each delimiter, each empty separator line, and each body (or preamble or epilogue) section. The delimiter and separator lists are referenced directly by the **delimiter** and **separator** pointers in the **mparse_entity** structure.

Termination

After **mparse_parse** returns, data (e.g. `ioerr`) in the **mparse_message** structure may be examined. If one wishes to examine parsed message structure, that should be done via the *hook_end_of_message* application hook described above, *i.e.* before `mparse` frees allocated structures and returns.

Support Functions

There are a number of support functions available for working with the **mparse_token** structure, which stores information about the token returned by the lexical analyzer, and provides pointers which comprise linked lists. One list links tokens in the order they are recognized, another is used to group tokens into a logical construct, such as an address, which may exclude some input tokens such as whitespace, line folding, and/or comments. Other pointers are used for navigating lists and bracketed constructs.

The **mparse_print_token** and **mparse_print_tokens** functions may be used to print a list of logically-grouped tokens (`mparse_print_token`) or a complete list of tokens (`mparse_print_tokens`) to the specified `FILE`. Character strings which precede and follow the tokens may be specified as the second and fourth arguments:

```
size_t mparse_print_token(FILE *, const char *, const struct mparse_token *, const char *);
```

```
size_t mparse_print_tokens(FILE *, const char *, const struct mparse_token *, const char *);
```

Both functions use **fwrite** to produce output, so there is no problem with binary data, including ASCII NUL. Byte-stuffing is included if specified for the message (if any) associated with the token or tokens. Each function returns the count of octets written. If a `NULL FILE *` argument is provided, no output is actually written, but the count returned is what would have been written.

Alternatively, the token lists may be expanded into an array of characters (with no provision for byte stuffing). The functions **mparse_token_string** and **mparse_tokens_string** may be used for this purpose:

```
size_t mparse_token_string(const struct mparse_token *, char *, size_t, unsigned int, unsigned int);
```

```
size_t mparse_tokens_string(const struct mparse_token *, char *, size_t, unsigned int, unsigned int);
```

The second argument points to the array of characters to be filled, and the third argument gives the size of the array. The return value gives the number of bytes written (not including the terminating '\0') if the buffer is large enough, or the required minimum size if the buffer is not large enough. To determine how large an array is needed, therefore, one can first call one of these functions with either a NULL character pointer or a zero size. The fourth argument provides information on the context of the first argument. It is constructed by bitwise ORing zero or more of the following:

symbolic name	description
MPARSE_CONTEXT_CFWS	token is part of CFWS
MPARSE_CONTEXT_DID_SPACE	whitespace exists at end of output buffer
MPARSE_CONTEXT_QUOTED	token is in a quoted-string

MPARSE_CONTEXT_DID_SPACE is primarily used for recursive calls and when **mparse_tokens_string** calls **mparse_token_string**. The fifth argument controls how comments, line folding, whitespace (collectively **CFWS**), how non-ASCII and ASCII control bytes are handled, and whether or not quoted and backslash-escaped sequences are canonicalized. It is constructed by bitwise ORing zero or more of the following:

symbolic name	description
MPARSE_STRING_MODE_CANONICALIZE	eliminate unnecessary quoting
MPARSE_STRING_MODE_COMMENT_SPACE	use a single space character in place of each comment
MPARSE_STRING_MODE_ENCODE_BITS	encode all non-ASCII octets using a quoted-printable-like encoding
MPARSE_STRING_MODE_ENCODE_CTLs	encode all ASCII control characters using a quoted-printable-like encoding
MPARSE_STRING_MODE_NORMALIZE_AT	use @ in place of RFC 733 " at "
MPARSE_STRING_MODE_REMOVE_WS	ignore all non-essential whitespace
MPARSE_STRING_MODE_SQUEEZE_WS	use a single space character to represent any run of whitespace
MPARSE_STRING_MODE_DELETE_TRAILING_WS	delete trailing whitespace
MPARSE_STRING_MODE_ENCODE_TRAILING_WS	encode trailing whitespace
MPARSE_STRING_MODE_UNFOLD	ignore line folding

There is interaction between these values; **MPARSE_STRING_MODE_UNFOLD** | **MPARSE_STRING_MODE_COMMENT_SPACE** | **MPARSE_STRING_MODE_SQUEEZE_WS** will replace all runs of CFWS with a single space character, and **MPARSE_STRING_MODE_UNFOLD** | **MPARSE_STRING_MODE_COMMENT_SPACE** | **MPARSE_STRING_MODE_REMOVE_WS** will ignore all non-essential CFWS.

Canonicalization of quoting may be useful when comparing local-parts or domains in addresses or message-ids; the following are all equivalent:

```
<foo.bar@[1.2.3.4]> (canonical form)
```

```
<"foo.bar"@[1\2.3.4]>
```

```
<"f\oo.bar"@[1\2.3.4]>
```

```
<"f\oo\bar"@[1\2.3.4]>
```

```
<"f\oo\bar"@[1\2.3.4]>
```

Normally, storage for tokens is allocated and released automatically. In some situations, it may be necessary to free space allocated for tokens which are no longer required, such as when replacing a body section. the functions **mparse_free_token** and **mparse_free_tokens** are used for this purpose:

```
void mparse_free_token(const struct mparse_message *, struct mparse_token *);  
void mparse_free_tokens(const struct mparse_message *, struct mparse_token *);
```

A token may be created with a call to

```
struct mparse_token *mparse_new_token(const struct mparse_message *, const char *, int, int, int,  
int);
```

The token created uses an allocated copy of the string pointed to by the second argument as the token **tok** member; the four integer arguments are used for the **len**, **col**, **type**, and **val** members, respectively. If **len** is zero, the appropriate length will be calculated using the usual C string rule that a zero valued character terminates the string.

A copy of an existing related set of tokens may be made by calling

```
struct mparse_token *mparse_token_copy(const struct mparse_message *, const struct mparse_field  
*, const struct mparse_token *);
```

or

```
struct mparse_token *mparse_tokens_copy(const struct mparse_message *, const struct mparse_field  
*, const struct mparse_token *, struct mparse_token **);
```

The former copies tokens linked by the **next** and **trailer** pointers, while the latter uses **trailer** and **next2** pointers. **tokens_copy** can also return the address of the last token copied by supplying a non-*NULL* pointer as the last argument. Both functions assign the copied tokens to the specified field structure.

A token or group of related tokens may be inserted into an existing linked structure of tokens using

```
void mparse_insert_tokens(struct mparse_token *, struct mparse_token *);
```

The first argument points to a token in an existing linked structure; the token or linked structure of tokens pointed to by the second argument will be inserted after the token pointed to by the first argument.

Various types of pointers may be established between tokens using

```
struct mparse_token *mparse_link_tokens(int, struct mparse_token *, struct mparse_token *);
```

The first argument specifies the types of pointer and may be bitwise ORed from the values given by the macros **MPARSE_LINK_NEXT**, **MPARSE_LINK_NEXT2**, **MPARSE_LINK_TRAILER**, and **MPARSE_LINK_CLOSE**. The remaining two arguments are pointers to **mparse_token** structures to be linked.

A pointer to the first token of a field body (i.e. after the field name, colon, and any CFWS) may be obtained by calling

```
struct mparse_token *mparse_field_body(const struct mparse_field *fld);
```

with a pointer to the *mparse_field* structure.

When token string text is changed, or tokens are inserted or deleted from a token stream, the column numbers associated with tokens may change. While the above functions automatically correct for that situation, it may sometimes be necessary for applications to make corrections. That may be accomplished by calling the function

```
int mparse_adjust_col(struct mparse_token *);
```

with the argument pointing to the last token known to have a correct column number. Subsequent tokens' column numbers through the next line ending are corrected. The return value is the column of the last token corrected, or a negative value if no adjustment could be made.

An instance of a particular field type in an entity may be located by calling the function

```
struct mparse_field *mparse_find_field(struct mparse_entity *, int, int, unsigned int);
```

where the first integer argument is either zero or the symbolic constant `MPARSE_RESENT_FIELD` to indicate whether an ordinary field or a Resent- version (if applicable) is to be returned. The second integer argument indicates the type of field, and is the token value in the associated `field_state` structure. The unsigned integer indicates how many instances should be skipped:

```
mparse_find_field(entity, 0, MPARSE_FIELD_RECEIVED, 2U);
```

returns a pointer to the 3rd Received field in the specified entity (two Received fields are skipped).

Given a token value for a field name, the function

```
const char *mparse_field_name(int);
```

returns a pointer to the canonical field name string.

The token value associated with a field name can also be used to return a pointer to a constant (read-only) `field_state` structure for that field name by calling

```
const struct mparse_field_state *mparse_field_state(int);
```

A copy of a field may be appended to the fields of an entity by calling the function

```
int mparse_copy_field(const struct mparse_field *, struct mparse_entity *, struct mparse_field *);
```

which allocates a copy of the field and inserts it in the fields in the specified entity, before the field (in the destination entity) specified by the last argument. It returns a negative value with *errno* set appropriately if bad arguments are supplied or in the event of a system error. It returns zero on success.

Message Body Decoding

MIME provides two methods of encoding body information for transport: base64 encoding and quoted-printable encoding as defined in RFC 2045. The functions `decode_b64` and `decode_qp` decode body sections which have been encoded by these methods, replacing the original (encoded) body by a sequence of tokens representing the decoded content.

```
void mparse_decode_b64(struct mparse_entity *);
```

```
void mparse_decode_qp(struct mparse_entity *);
```

Message bodies may contain other coded content which has been coded for reasons other than transport robustness, e.g. HTML or other page description languages. Such encoding is beyond the scope of what `mparse` has been designed to handle; `mparse` will collect the content and may be used to reverse transport encoding, but further processing (e.g. rendering HTML or another page description language to a display) must be performed by the calling application via the provided function hooks.

In addition to the transformations effected by base64 and quoted-printable encoding, MIME provides specification of the domain of body content via the Content-Transfer-Encoding field. The valid encoding types are documented in RFC 2045. The `mparse_encoding` structure holds information about the message body domain for an encoding:

member (struct mparse_encoding)	description
const char *encoding_name;	canonical encoding tag
unsigned int domain;	domain (7bit, 8bit, or binary)

Registered encodings are recognized automatically by `mparse`; extensions can be recognized via the hook described above.

Names of registered encodings may also be recognized by calling

```
const struct mparse_encoding *mparse_encoding_entry(const char *, unsigned int);
```

which returns a const `mparse_encoding` structure pointer for recognized encodings or a `NULL` pointer if the string is not a registered encoding name.

The standard encoding values are available by calling the following functions, each of which returns a pointer to a constant encoding structure:

```
const struct mparse_encoding *mparse_encoding_7bit(void);
const struct mparse_encoding *mparse_encoding_8bit(void);
const struct mparse_encoding *mparse_encoding_binary(void);
const struct mparse_encoding *mparse_encoding_base64(void);
const struct mparse_encoding *mparse_encoding_qp(void);
```

The function

```
const struct mparse_encoding *mparse_self_encoding(struct mparse_entity *);
```

returns a pointer to a constant encoding structure based on the domain of the actual content of the entity specified.

Message Generation

Mparse provides functions for generating low-level and high-level message components, including complete MIME messages. Recall that simple messages are stored in an **mparse_entity** structure; an empty one may be created by calling the function **mparse_new_entity**:

```
struct mparse_entity *mparse_new_entity(struct mparse_message *);
```

If it is desired to copy an **mparse_entity** structure, including content, **entity_copy** may be used:

```
struct mparse_entity *mparse_entity_copy(struct mparse_message *, const struct mparse_entity *);
```

The copied entity is assigned to the message specified, if not *NULL*, or to the same message as the original.

Having an **mparse_entity** structure, empty or otherwise, it is possible to add either header or body content. The **mparse_insert_field** function is provided for adding fields:

```
int mparse_insert_field(struct mparse_entity *, struct mparse_field *, const char *, int, const char *, int, size_t, const char *,
const char *, ...);
```

The **mparse_insert_field** function accepts a variable number of character string arguments and will generate a string in a character array. The first two arguments to **mparse_insert_field** are the **mparse_entity** structure which will contain the field and a pointer to the field before which the new field will be inserted. If a *NULL* pointer is given for the field, the new field is appended to any existing fields. The third and fourth arguments point to a string naming the charset used for the field contents and give the length of that string. A *NULL* pointer or non-positive length indicates the default US-ASCII charset (it is then an error if any non-ASCII characters are provided in the field strings). Language information is provided via the fifth argument. The sixth argument is a flag which indicates whether to use the canonical capitalization for the field name tag, etc. The seventh argument gives the maximum line length and controls folding of long lines. The string pointer supplied as the eighth argument is used as the field tag, and should not include a colon. Following strings are used to build the field body content; a zero (*NULL*) pointer must be passed as the last argument. An empty Bcc field can be appended with

```
mparse_insert_field(entity, 0, 0, 0, 0, 1, 78, "bCc", 0);
```

which will generate the field "Bcc:\r\n".

To insert a field at the start of the field list, use the **fields** pointer from the **mparse_entity** structure. For example:

```
int c;
char buf[64];
mparse_gen_date_local(buf, sizeof(buf));
c = mparse_insert_field(entity, entity->fields, 0, 0, 0, 1, 78, "Date", buf, 0);
```

will insert a Date field at the beginning of the list of fields in the **mparse_entity** structure pointed to by *entity* (**gen_date_local** will be described below).

A low-level method of creating a field with no content is

```
struct mparse_field *mparse_new_field(const struct mparse_entity *);
```

which allocates a structure and associates it with the specified entity, returning a pointer to the allocated structure.

```
int mparse_header_end(struct mparse_entity *entity);
```

should be called after all header fields have been generated. **mparse_header_end** summarizes information about the header fields and cross-checks fields for consistency and number. It returns a negative value if there is a very serious error, otherwise a count of the number of errors found.

A time-stamp line (a.k.a. Received field) may be inserted in a message by calling:

```
int mparse_time_stamp(struct mparse_message *message, struct sockaddr_in *peer, const char *name, int esmtp, const char *id);
```

where **mparse_message** points to the message, **peer** points to a *sockaddr_in* structure for the sender socket connection if available, otherwise **name** gives the domain name of the sender (preferably as determined by a reliable mechanism; the content of the HELO or EHLO SMTP command is unreliable (easily forged)), the **esmtp** flag indicates whether ESMTP protocol is being used (`esmtp > 0`), SMTP is being used (`esmtp == 0`), or neither protocol applies (`esmtp < 0`), and **id** gives a string for the optional id part of the field (NULL to omit the id, which will likely be elided in any case due to irreconcilable conflicts between the relevant RFCs). The generated time stamp line will be inserted at the beginning of the top-level message header fields as required by the SMTP RFCs.

SMTP servers insert a return path line when making "final" delivery. Such a line may be inserted in a message by calling:

```
int mparse_return_path(struct mparse_message *message, const char *aa);
```

where **mparse_message** points to the message and **aa** is a character string containing the angle-bracketed return path. The Return-Path field is prepended to the top-level message header fields, and if there are no errors any pre-existing Return-Path fields are removed.

SMTP MTAs supporting DSN extensions may need to insert an Original-Recipient field, which may be accomplished by calling:

```
int mparse_original_recipient(struct mparse_message *message, const char *addr_type, const char *or);
```

where, as above, **mparse_message** points to the message, **addr_type** points to a character string giving the (registered) address type, and **or** points to a character string giving the original recipient address. The field is added near the beginning of the top-level message header fields, between the Return-Path and Received fields. On success, any pre-existing Original-Recipient top-level header fields are removed, as described in RFC 2298 section 2.3 (also RFC 3298).

Body content may be appended to an **mparse_entity** structure using the **mparse_append_body_line**, **mparse_append_body_from_file**, or **mparse_append_body_from_buffer** functions:

```
int mparse_append_body_line(struct mparse_entity *, const char *, const char *, unsigned int, const char *, unsigned int, const char *, unsigned int, const char *, const char *, unsigned int, const char *, const char *, unsigned int);
```

```
int mparse_append_body_from_file(struct mparse_entity *, FILE *, const char *, unsigned int, const char *, unsigned int, const char *, unsigned int, const char *, const char *, unsigned int, const char *, const char *, unsigned int);
```

```
int mparse_append_body_from_buffer(struct mparse_entity *, char *, unsigned int, const char *, unsigned int, const char *, unsigned int, const char *, unsigned int, const char *, const char *, unsigned int);
```

Each function's first argument is the **mparse_entity** structure which is to contain the appended content. **mparse_append_body_line** takes a `const char *` as its second argument, while **mparse_append_body_from_file** takes a *stdio* **FILE ***, and **mparse_append_body_from_buffer** takes a pointer to a character array and an unsigned integer giving the number of octets of interest. The remaining arguments provide information which is used to properly tag the content, and consists of:

argument	description
<code>const char *type</code>	content media type
<code>unsigned int typelen</code>	length of type string
<code>const char *subtype</code>	content media subtype
<code>unsigned int subtypeplen</code>	length of subtype string
<code>const char *cst</code>	charset (if applicable)
<code>unsigned int cslen</code>	length of charset string
<code>const char *type_other_parameters</code>	any additional parameters for the Content-Type field (use <i>mparse_parameter_string()</i>)
<code>const char *disp</code>	disposition (if applicable)
<code>unsigned int displen</code>	length of disposition string
<code>const char *disp_other_parameters</code>	any additional parameters for the Content-Disposition field (use <i>mparse_parameter_string()</i>)
<code>const char *filename</code>	filename for Content-Disposition
<code>const char *languages</code>	content languages (if applicable)
<code>unsigned int linelen</code>	maximum line length for folding content

It is possible to build up a message body with multiple calls to **mparse_append_body_line**, or multiple lines may be appended at one time by separating them with `\r\n` in the string. The file pointer should be opened and positioned properly before calling **mparse_append_body_from_file**. This function makes it possible to read arbitrary binary content from files. To preserve binary content, open the file with a binary mode if your implementation of *fopen* supports it.

Body content may be copied from one entity to another with **int mparse_copy_body(const struct mparse_entity *src, struct mparse_entity *dst);**

which copies body fields from *src* to *dst*.

Application programmers who wish to enumerate standard disposition values (for example, to construct a menu) can call:

```
const struct mparse_disposition *mparse_disposition(int n);
```

sequentially with an integer argument beginning with 1. **mparse_disposition** will return a pointer to the structure in sequence according to the disposition value (see **mparse.h**), returning a *NULL* pointer when the end of the list is reached. Not all dispositions are appropriate for all messages; application authors may examine the disposition value and avoid presenting inappropriate values.

const struct mparse_disposition *mparse_disposition_entry(const char *, unsigned int);

validates a particular string with specified length as a known disposition keyword. A *NULL* pointer is returned if the string is not a valid disposition keyword, otherwise a pointer to a structure as noted above is returned.

There are also some low-level functions for putting content into an **mparse_field** structure, which can then be inserted in either the header or body section of an **entity** structure.

struct mparse_field *mparse_headerize_string(struct mparse_entity *, const char *, int, int);

returns a pointer to a newly allocated field structure which may be associated with the specified entity. Content is taken from the string pointed to by the second argument. The third argument indicates whether or not message content is to be canonicalized (non-zero to perform canonicalization), and the last argument indicates whether or not the content is part of the entity body (non-zero for body content).

Similarly, content can be taken from a stdio FILE using:

struct mparse_field *mparse_headerize_file(struct mparse_entity *, FILE *, int, int);

Finally, content may be taken from a character array with specified length:

struct mparse_field *mparse_headerize_buffer(struct mparse_entity *, char *, unsigned int, int, int);

The unsigned integer argument specifies the length in octets.

A field and its allocated storage may be discarded by calling

void mparse_free_field(const struct mparse_message *message, struct mparse_field *h)

void mparse_body_end(struct mparse_entity *entity);

should be called when all body content has been inserted. It consolidates multiple body **struct mparse_field** structures into a single structure, calls the user hooks for processing the body section, and checks for errors.

The above functions suffice for generating simple messages and discrete MIME messages. Composite MIME messages involve encapsulating messages in a MIME message entity or combining one or more simple body parts into a MIME multipart entity.

Sometimes it may be necessary to split a body into multiple **struct mparse_field** structures.

int mparse_split_body(struct mparse_entity *entity, unsigned int nlines);

splits the body into structures containing at most *nlines* lines.

It is possible to free the **struct mparse_field** structures associated with the body. That can be done by calling

void mparse_free_body(struct mparse_entity *);

Encapsulating a message can be performed with the function:

struct mparse_entity *mparse_encapsulate(struct mparse_entity *, const char *, const char *);

The three arguments are: the **mparse_entity** structure corresponding to the message to be encapsulated, a character string giving the message media subtype, and a character string giving any parameters (see **parameter_string** below). For example, to encapsulate a simple message as a MIME message/rfc822 type

```
mparse_encapsulate(entity, "rfc822", 0);
```

would return a pointer to the **mparse_entity** structure encapsulating the message.

```
mparse_encapsulate(entity, "external-body", ";access-type=local-file;name=foo");
```

is another example. Details of the message subtypes and **mparse_parameters** may be found in RFC 2046.

Application programmers who wish to enumerate standard external-body access-types (for example, to construct a menu) can call:

const struct mparse_name_val *mparse_access_type(int n);

sequentially with an integer argument beginning with 1. **mparse_access_type** will return a pointer to the structure in sequence according to the access-type name returning a *NULL* pointer when the end of the list is reached.

MIME multipart entities are initially constructed with the **mparse_create_multipart** function:

```
struct mparse_entity *mparse_create_multipart(struct mparse_entity *, const char *, const char *, const char *, const char *, const char *);
```

The first argument is a pointer to the **mparse_entity** structure representing the entity to be placed within a multipart enclosure. The second argument is a character string for the multipart media subtype. The third argument is for the *boundary* parameter which is required for all multipart entities. The fourth argument is for any additional parameters. The fifth argument is a string for the optional preamble, and the sixth argument is a string for the epilogue. For example:

```
mparse_create_multipart(entity, "mixed", "x", 0, "This is the preamble", "This is the epilogue.\r\n");
```

might have been used in the generation of the multipart example given in the earlier diagram.

Both **mparse_encapsulate** and **mparse_create_multipart** return a pointer to the enclosing **mparse_entity** structure.

Additional entities represented by **mparse_entity** structures may be inserted in existing encapsulations or enclosures with the **mparse_insert_entity** function:

```
int mparse_insert_entity(struct mparse_entity *, struct mparse_entity *, struct mparse_entity *, struct mparse_token *);
```

The first argument points to the **mparse_entity** structure representing the entity to be enclosed, while the second argument points to the **mparse_entity** structure of the enclosing composite entity. The third argument points to an entity within the enclosure; the new entity (specified by the first argument) will precede the one specified by the third argument (a third argument of zero places the new entity last). The fourth argument is for a delimiter and is intended for use in parsing text messages; it should be a zero (*NULL*) pointer when building messages using **mparse**.

A string specifying MIME parameters may be generated with a call to:

```
int mparse_parameter_string(const struct mparse_message *message, const char *attribute, const char *value, unsigned int vlen, const struct mparse_charset *cs, const unsigned char *language, char *buf, int sz);
```

which generates (in *buf*) a string of the form " ; attribute=value", taking into consideration the content of *value*, and any specified charset (required for non-ASCII content in *value*) and/or language. The generated string is compliant with RFC 2231. The value returned by **mparse_parameter_string** is -1 on error (e.g. non-ASCII content with no charset specified), the number of bytes written to *buf* (not including the terminating '\0') if *buf* is large enough (size specified by *sz*), or the required size of a buffer in bytes if the specified buffer is too small (or if a *NULL* pointer is given for *buf*). Any required encoding, quoting, or continuation of long *values* is handled automatically by **mparse_parameter_string**.

A string suitable for use as a phrase (such as appears in Keywords fields, and as the display name for a mailbox or named group) may be generated with a call to:

```
int mparse_phrase_string(const struct mparse_message *message, const char *phrase, unsigned int len, const struct mparse_charset *cs, const unsigned char *language, const unsigned char *encoding, char *buf, int sz);
```

which generates (in *buf*) a string for the phrase. taking into consideration the content of *phrase*, and any specified charset (required for non-ASCII and/or certain ASCII (NUL, lone CR or LF) content in *phrase*) and/or language, and encoding. The value returned by **mparse_phrase_string** is -1 on error (e.g. non-ASCII content with no charset specified), the number of bytes written to *buf* (not including the terminating `'\0'`) if *buf* is large enough (size specified by *sz*), or the required size of a buffer in bytes if the specified buffer is too small (or if a NULL pointer is given for *buf*). Any required encoding or quoting is handled automatically by **mparse_phrase_string**. If a language is specified, the entire phrase will be encoded as that is the only way that the entire phrase can be language-tagged. Encoding may be specified as one of the standard encoding types (B or Q), or a NULL pointer may be supplied, in which case **mparse_phrase_string** will select the encoding based on content.

A quoted-string may be generated with a call to:

```
int mparse_quote_string(const char *s, unsigned int len, char *buf, int sz);
```

which quotes the content in *s*, backslash-quoting any double quotes or backslashes. The value returned by **mparse_quote_string** is -1 on error (e.g. non-ASCII content), the number of bytes written to *buf* (not including the terminating `'\0'`) if *buf* is large enough (size specified by *sz*), or the required size of a buffer in bytes if the specified buffer is too small (or if a NULL pointer is given for *buf*).

Multiple strings can be concatenated into a single string via:

```
size_t mparse_build_string(char *, size_t, int, const char *, const char *, ...);
```

the first and second arguments give a buffer for the result and its size. The third argument specifies whether a MPARSE_TOKEN_CRLF pair is to be appended (if nonzero). The fourth argument is a string giving a field name, and is used when generating a field string. The field name is followed in the generated string by a colon and space (which should not be supplied in the strings). When not generating a field string, the fourth argument should be a NULL pointer. The remaining arguments are pointers to the strings to be concatenated (after the field name, colon, and space if generating a field), and must end with a NULL pointer.

Message Output and Processing

Just as it is possible to print a list of tokens, it is possible to print an entire message consisting of linked **mparse_entity** structures.

```
size_t mparse_print_message(FILE *, const char *, struct mparse_message *, const char *, FILE *);
```

will print the entire message referenced by the **mparse_message** structure to the specified *FILEs* (message proper to first *FILE*, errors and warnings to the second). **mparse_print_tokens** is called to output each part in turn, so the comments above regarding the advantages of **fwrite** still apply. If byte-stuffing is specified, it will be applied where appropriate and a line with a lone dot will be output at the end of the message. Any prefix string pointed to by the first *const char ** argument is printed before the message, and any suffix string pointed to by the second *const char ** argument is printed after the message (and byte-stuffing lone dot line). Both prefix and suffix are printed to the same *FILE* as the message. **mparse_print_message** returns the number of bytes written to the *FILE* used for the message output (including bytes for errors and warnings if the two *FILEs* are not *NULL* and are the same or if they use the same file descriptor). If a *FILE ** is *NULL*, no output is actually produced, but the byte counts are computed.

size_t mparse_print_entity(FILE *, const char *, const struct mparse_entity *, unsigned int, const char *, FILE *);

is similar to **mparse_print_message**, except that it outputs the contents of a single **mparse_entity** structure. The third argument controls which parts of the structure are output, and is constructed by bitwise OR-ing the following values according to the desired parts;

symbolic name	description
MPARSE_ENTITY_COMPONENT_DELIMITER	MIME delimiter
MPARSE_ENTITY_COMPONENT_HEADER	fields
MPARSE_ENTITY_COMPONENT_SEPARATOR	header/body separator (empty line)
MPARSE_ENTITY_COMPONENT_BODY	body

The symbolic value **MPARSE_ENTITY_COMPONENT_ALL** may be used to select all of the above parts.

The error and warning fields described above are also output by **mparse_print_message** and **mparse_print_entity** unless the **suppress_errors** and/or **suppress_warnings** flags are set in the **mparse_message** structure.

Likewise,

size_t mparse_print_field(FILE *, const char *, const struct mparse_field *, const char *, FILE *)

outputs a single field.

Generic processing of a message may be performed by calling

int mparse_process_message(int (*f)(struct mparse_entity *), struct mparse_message *, int);

passing a function which processes a single **mparse_entity** structure. The integer argument controls the order of processing of **mparse_entity** structures in the message:

argument	description
> 0	message's siblings (eldest first and including message in order) and their subtrees
< 0	starting with youngest descendant and working up through eldest sibling, reverse order
0	message and its subtree first, then siblings (eldest -> youngest) and their subtrees

The fields in an **mparse_entity** structure may be processed by calling

int mparse_process_header(int (*f) (struct mparse_field *), struct mparse_entity *);

Likewise, the body may be processed via

int mparse_process_body(int (*f) (struct mparse_field *), struct mparse_entity *);

And the tokens comprising a field may be processed by calling the function

int mparse_process_field(int (*f) (struct mparse_token *), struct mparse_field *);

If a copy of a message has been made, the specified application hooks can be called automatically and the allocated structures freed by calling:

int mparse_process_and_free_message(struct mparse_message *, int);

The integer argument, if non-zero, indicates abnormal termination and prevents message repair (if specified) and calls to the application-specified end-of-message hook. The return value is zero under normal conditions, but may be non-zero if some error occurred (e.g. if the supplied integer argument was non-zero, or if the end-of-message hook returned non-zero).

Message Modification

In addition to building up composite MIME entities, it is sometimes necessary to decompose them. Several functions are provided for this purpose:

int mparse_unlink_entity(struct mparse_entity *);

void mparse_free_entity(struct mparse_entity *);

```
int mparse_v_free_entity(struct mparse_entity *, va_list);
```

```
int mparse_collapse_composite(struct mparse_entity *);
```

The function **mparse_unlink_entity** removes the **mparse_entity** structure from its context, closing up any links where it was removed. The structure still exists and can be referenced, printed, etc. but it is isolated from all other structures. The function **mparse_free_entity** removes the structure from its context, deletes all content, and frees allocated space. When a composite MIME entity has a single body part (not counting multipart preamble and epilogue) **mparse_v_free_entity** is like *mparse_free_entity* but takes an unused *va_list*, and can therefore be used with *mparse_process_message* to clean up in the event of an error. **mparse_collapse_composite** may be called to remove the MIME enclosure, substituting the contents. While there may be some esoteric cases where a single-part "multipart" entity makes sense, usually it is better to present the content without the multipart wrapper.

Each of the above three functions returns zero on success. A non-zero return means that the structure could not be unlinked because that would make some other structure unreachable and would therefore lead to a memory leak.

The **mparse_free_entity** function isolates, deletes content, and frees allocated storage for the referenced **entity** structure and all structures linked via *child* and *next_sibling* links:

```
void mparse_free_entity(struct mparse_entity *);
```

A field may be removed by calling:

```
int mparse_delete_field(struct mparse_field *);
```

Message body content can be encoded for transport as described earlier. The following two functions perform the inverse of the two decoding functions:

```
int mparse_encode_b64(struct mparse_entity *);
```

```
int mparse_encode_qp(struct mparse_entity *);
```

and the function

```
int mparse_encode_body(struct mparse_entity *);
```

determines which encoding function produces the more compact encoding and applies that function.

Encoding can be applied to discrete media types within a message which require encoding by calling the function

```
int mparse_encode_message(struct mparse_message *);
```

Message Copies

Sometimes the original message should be kept intact, but it is desired to make a copy which can be modified or otherwise manipulated without affecting the original. The function **copy_message** makes a copy of a message, including content and linked structures:

```
struct mparse_entity *mparse_copy_message(struct mparse_message *, struct mparse_entity *);
```

The result is a copy of the tree of entities beginning with the specified entity, including content. Each **entity** in the copy points to the specified **mparse_message** structure.

Message Cleanup

After encoding or decoding a body section, or after enclosing an entity in a composite structure, it may be necessary to change the specified transport encoding of parent structures. Use:

```
struct mparse_encoding *mparse_adjust_encoding(struct mparse_entity *);
```

which revises the encoding in the **mparse_entity** structure and returns a pointer to the **mparse_encoding** structure (described above) which holds information about the encoding and its domain.

There are defaults for MIME media types, charset, and transport encoding. It is not always necessary to provide explicit fields for these. When an entity is enclosed in a composite structure or its transport encoding has changed, it may be possible to elide some MIME fields. The function

```
int mparse_minimize_mime_fields(struct mparse_entity *);
```

checks for this condition and removes unnecessary MIME fields. It also ensures that there is a MIME-Version field as required by RFC 2045 section 4.

Message Fragmentation and Reassembly

It may be necessary or desirable to split large messages into smaller pieces for transport and to reassemble the fragments into a complete message.

```
int mparse_fragment_message(struct mparse_message *message, unsigned int frag_sz, struct mparse_message ***pmsg_ptr_array)
```

takes an original *message* and splits it into fragments with size no greater than *frag_sz* octets in accordance with the rules in RFC 2046, allocating message structures as needed and placing an array of pointers to those message structures in allocated memory whose address is stored in the location given by *pmsg_ptr_array* (if not *NULL*), returning a negative value on error (with *errno* set appropriately), otherwise returning the number of fragment messages. It is the caller's responsibility to release allocated storage associated with the fragment messages, their content, and the array of pointers to the message structures.

An application author may reconstitute a fragmented message by calling

```
int mparse_combine_partial(struct mparse_message *message, unsigned int nmessages, struct mparse_message **messages);
```

supplying a pointer to an **mparse_message** structure for the reassembled message (which should have a *context* value including **MPARSE_SECONDARY_ROLE_TRANSFORM_REASSEMBLE**), a count of the number of fragment messages, and an array of pointers to the fragment messages. On error, a negative value is returned with *errno* set appropriately. Otherwise, zero is returned.

Building and Bursting Message Digests

The MIME media type multipart/digest is useful for transporting collections of messages. A digest may be created from a set of messages by calling the function

```
struct mparse_entity *mparse_build_digest(unsigned int nmessages, struct mparse_message **messages, const char *boundary, const char *other_parameters, const char *preamble, const char *epilogue);
```

supplying a count of the number of messages, an array of pointers to the messages, an optional message boundary delimiter string, optional additional *mparse_parameters*, optional preamble text, and optional epilogue text. On error, a *NULL* pointer is returned with *errno* set appropriately. Otherwise, a pointer to the multipart/digest wrapper entity is returned. That entity may be further wrapped in an enclosing media type (e.g. to provide a table of contents) or it may be inserted in a message structure.

A digest may be split into individual messages by calling the library function

```
int mparse_burst_digest(struct mparse_entity *entity, struct mparse_message ***pmsg_ptr_array);
```

supplying a pointer to the multipart/digest wrapper entity and the address of a location which can hold a pointer to an array of **mparse_message** structures. On error, a negative value is returned with *errno* set appropriately. On success, a count of the number of individual messages is returned. If *pmsg_ptr_array* is not *NULL*, The individual messages will have been placed in allocated message structures and a pointer to an array of **mparse_message** structure pointers will be placed in the location specified by *pmsg_ptr_array*. It is the caller's responsibility to free allocated storage for the individual messages and their contents and the array of pointers.

Low-level Message Component Generation

A number of functions are provided for generating RFC-compliant message components.

There are four variants which generate date-time components, according to whether UTC or local time plus offset is used, and whether or not the optional day-of-week is included. Each function takes a pointer to a character array for the result, and the size of the array in bytes. Return value is the number of bytes written (excluding the terminating '\0'), or, if the buffer is too small, the size of the buffer necessary to hold the result (including the terminating '\0'), or a negative value if a serious error occurred. A separate function is also provided which can be made equivalent to any of the other four by specifying zero-valued or non-zero arguments to specify inclusion of day-of-week and local time.

int mparse_gen_date(char *, int, unsigned int, unsigned int);

int mparse_gen_date_local(char *, int);

int mparse_gen_date_utc(char *, int);

int mparse_gen_dow_date_local(char *, int);

int mparse_gen_dow_date_utc(char *, int);

A domain literal for an Internet address may be obtained by a call to **mparse_gen_ip_literal**. The function takes a pointer to a character array for the result, and the size of the array in bytes. Return value is the number of bytes written (excluding the terminating '\0'), or, if the buffer is too small, the size of the buffer necessary to hold the result (including the terminating '\0'), or a negative value if a serious error occurred. The function **mparse_gen_ip_name** is similar, except it returns a domain name rather than a domain literal.

int mparse_gen_ip_literal(const struct sockaddr_in *, char *, int);

int mparse_gen_ip_name(const struct sockaddr_in *, char *, int);

The domain name for the host running **mparse** may be obtained by a call to the function **mparse_gen_hostname**. The function takes a pointer to a character array for the result, and the size of the array in bytes. Return value is the number of bytes written (excluding the terminating '\0'), or, if the buffer is too small, the size of the buffer necessary to hold the result (including the terminating '\0'), or a negative value if a serious error occurred. The function **mparse_gethostaddr** returns the current host's IP address in the structure pointed to by the first argument (a struct sockaddr), and the second argument gives its size. Preference is given to routable addresses on a host with multiple addresses.

int mparse_gen_hostname(char *, int);

int mparse_gethostaddr(struct sockaddr *, int *);

Some IP addresses are reserved for private use, examples, etc. and are not generally routable through the Internet. The function **mparse_is_routable** returns zero for such addresses, non-zero for routable addresses.

int mparse_is_routable(const struct sockaddr_in *);

A message-id component (including the angle brackets) can be generated by calling **mparse_gen_message_id**. The function takes a pointer to a character array for the result, and the size of the array in bytes. Return value is the number of bytes written (excluding the terminating '\0'), or, if the buffer is too small, the size of the buffer necessary to hold the result (including the terminating '\0'), or a negative value if a serious error occurred. The calling syntax is:

int mparse_gen_message_id(char *, int);

Header Fields and Information

The function

```
const struct mparse_field_state *mparse_field_entry(const char *, int);
```

returns a pointer to a structure defined in the header file **mparse.h**. The members of that structure are:

member (struct mparse_field_state)description

const char *field_name;	Canonical field name
int token;	integer value returned by lexical analyzer and stored in <i>type</i> member of mparse_token structure
int next_state;	used internally by lexical analyzer
unsigned int resent_ok : 1;	flag indicates whether Resent- version of field is legal
unsigned int mime_content : 1;	indicates whether field is a MIME Content- field
unsigned int cached : 1;	indicates whether information is cached in the mparse_cache structure
unsigned int defaultable : 1;	indicates whether a default value exists in the absence of the field
unsigned int multiple_ok : 1;	are multiple instances of the field permitted in an entity
unsigned int unstructured : 1;	field is unstructured (RFC 2047 encoding is permitted)
unsigned int usefor_inherit : 1;	"inheritable" field as defined in usefor draft
unsigned int usefor_variant : 1;	"variant" field as defined in usefor draft
unsigned int obsolete_name:1;	field name has been changed
unsigned int hook_tokens : 4;	number of token pointers (<= 15) passed to corresponding application hook function
unsigned int modes : 8;	bitmap of permitted entity types for this field

The arguments to **mparse_field_entry** are a (case-insensitive) character string and its length, for example as obtained from a **token** structure's *tok* and *len* members.

Application programmers who wish to enumerate known fields (for example, to construct a menu) can call:

```
const struct mparse_field_state *mparse_field(int n);
```

sequentially with an integer argument beginning with 1. **mparse_field** will return a pointer to the structure in case-insensitive alphabetic sequence, returning a *NULL* pointer when the end of the list is reached. Fields which are defined as having obsolete names are not included in the returned data. Note that not all fields are valid in all contexts; application authors may wish to examine the *modes* member of the **mparse_field_state** pointed to by the return value to determine whether or not to present a given field to the user.

RFC 2047 (as amended by RFC 2231 and errata) defines an encoded-word sequence which can represent human-readable text in character sets beyond those which may appear in message fields. The character set is specified and language may be specified. The function

```
int mparse_is_encoded_word(const unsigned char *s, int location);
```

is provided to indicate whether a character string matches the syntax for such an encoded-word. **location** specifies the context of the character string and is constructed from the following values:

symbolic name	description
MPARSE_ENCODED_WORD_LOCATION_UNKNOWN	location cannot be determined (used alone)
MPARSE_ENCODED_WORD_IN_COMMENT	string is part of a comment in a structured field
MPARSE_ENCODED_WORD_IN_PHRASE	string is a word in a phrase in a structured field
MPARSE_ENCODED_WORD_IN_UTEXT	string is part of unstructured text

Note that a comment might occur within a phrase (in which case **MPARSE_ENCODED_WORD_IN_COMMENT** suffices); otherwise the **location** values above are mutually exclusive. If the string matches the syntax for an encoded-word, the function returns a non-zero value.

It is possible that a token matching the syntax for an encoded-word is not a valid encoded-word. The function **int mparse_bad_word(struct mparse_entity *entity, struct mparse_token *t)**; performs more stringent tests on a token and returns a non-zero value if the token is not a valid encoded-word, setting appropriate errors in the process. Context is determined automatically from the token.

An encoded word may be decoded by calling the function

```
int mparse_decode_encoded_word(struct mparse_entity *entity, struct mparse_token *t, unsigned char *buf, int len, const struct mparse_charset **pcs, const unsigned char **plang, unsigned int *planglen, const unsigned char **penc, unsigned int *penclen);
```

where *buf* and *len* provide a buffer of length *len* for the decoded text (*buf* may be *NULL* to determine a suitable length), and *entity* and *t* point to the structures containing the encoded-word token. The function returns the length of the decoded text (not including the terminating '\0' character unless the supplied buffer is too small). If *pcs* is not *NULL*, a pointer to the **mparse_charset** structure will be stored. Likewise, if *plang* and *planglen* are not *NULL* and a language is specified, the start of the language tag and its length will be stored. Finally, if *penc* and *penclen* are not *NULL* the start of the encoding tag and its length will be stored.

Application programmers who wish to enumerate standard MIME-compatible charsets (for example, to construct a menu) can call:

```
const struct mparse_charset *mparse_charset(int n);
```

sequentially with an integer argument beginning with 1. **mparse_charset** will return a pointer to the structure in MIB enum sequence (see the IANA character-sets registry) returning a *NULL* pointer when the end of the list is reached. Only MIME-compatible charsets are listed, and only the structure for the preferred name for each distinct charset is returned.

The charset tag given by string *s* having length *len* may be validated by calling the function

```
int mparse_bad_charset(struct mparse_message *message, struct mparse_token *t, const char *s, size_t len, const struct mparse_charset **pcs, unsigned int errors, unsigned int remedy int rfc, const char *ref, const char *sect, int x, const char *str);
```

which returns non-zero if the charset is not valid. In that case it may also use the *struct mparse_token *t* to indicate the offending token (*t* will usually be the token containing *s*). Because a charset may be specified in a number of contexts (in an encoded-word, in an extended-initial-value in a parameter, or as the value corresponding to a "charset" parameter in a Content-Type header field), **mparse_bad_charset** has provision for specifying the relevant RFC etc. for a possible error message. If the charset is valid and *const struct mparse_charset **pcs* is not *NULL*, **mparse_bad_charset** will store a pointer to the relevant *struct mparse_charset* there.

The function

```
const struct mparse_charset *mparse_charset_entry(const char *, unsigned int);
```

returns a pointer to a structure (defined in *mparse.h*) for a charset name with a specified length if the charset is registered, or a *NULL* pointer if there is no registered charset with the specified name.

The default charset in messages is US-ASCII. A pointer to the **mparse_charset** structure corresponding to US-ASCII may be obtained by calling the function

```
const struct mparse_charset *mparse_ascii_charset(void);
```

Charsets may have a number of aliases. The function

```
int mparse_charset_is_ascii(const struct mparse_charset *);
```

returns a positive integer if the specified charset is equivalent to US-ASCII, zero if it is not, and a negative integer if there is some error.

There are a number of restrictions on charsets which may be used with certain media types.

```
int mparse_incompatible_charset(const struct mparse_entity *, const struct mparse_charset *);
```

returns a positive integer if the specified charset is incompatible with the specified entity, zero if there is no incompatibility, and a negative value in the event of an error in arguments.

The function

```
const struct mparse_charset *mparse_preferred_charset(const struct mparse_charset *);
```

returns a pointer to the preferred **mparse_charset** structure for a given charset, if there is one.

A language tag may be validated by calling

```
int mparse_bad_language(struct mparse_message *message, struct mparse_token *t, const char *s, size_t len);
```

A language tag (as defined in RFC 3066) may consist of a raw ISO 3166 language tag possibly followed by a hyphen separator and an ISO 639 country code. IANA-registered language tags are also defined. **mparse_bad_language** checks these if present. It is also possible to check them separately by calling one of:

```
const struct mparse_language *mparse_language_entry(const char *str, unsigned int len);
```

```
const struct mparse_country *mparse_country_entry(const char *str, unsigned int len);
```

which return a *NULL* pointer if the string pointed to by the first argument (with length given by the second argument) is not a valid code of the corresponding type. Otherwise it returns a pointer to a structure defined in **mparse.h**. Each structure holds a pointer to the code tag, a pointer to the English name(s) corresponding to the tag, a sequence number according to alphabetic order by English name, a pointer to the French name(s) corresponding to the tag, and a sequence number according to alphabetic order by French name (English and French being the languages of the ISO documents). [The IANA-registered language tags have descriptions in English only, but these appear in the English and French members.]

Application programmers who wish to enumerate the country or language codes (for example, to construct a menu) can call one of:

```
const struct mparse_country *mparse_country_en(int n);
```

```
const struct mparse_country *mparse_country_fr(int n);
```

```
const struct mparse_language *mparse_language_en(int n);
```

```
const struct mparse_language *mparse_language_fr(int n);
```

sequentially with an integer argument beginning with 1. Each function will return a pointer to the structure in the corresponding alphabetic sequence, returning a *NULL* pointer when the end of the list is reached. The IANA-registered tags appear after the last ISO tag.

MIME Boundaries and Other Parameters

There are restrictions on the length and character set for MIME boundaries. The function

```
int mparse_boundary_ok(const char *);
```

returns 0 if the string is not suitable as a boundary, 1 if it is suitable as supplied, and 2 if it must be quoted when supplied as a parameter value in a Content-Type field.

RFC 2231 provides MIME extensions for parameter continuation and for specifying parameter charset and language. MIME parameter information is stored in a **mparse_parameter** structure having the following members:

member (struct mparse_parameter)description

struct mparse_parameter *initial;	head of singly-linked list
struct mparse_parameter *next;	singly-linked list
struct mparse_token *attribute;	parameter name
struct mparse_token *value;	value
struct mparse_token *importance;	importance (disposition-notification-options parameters)
const struct mparse_charset *charset;	charset for encoding
const struct mparse_language *language;	language for tagging
unsigned int section;	continuation fragment number
unsigned int extended : 1;	parameter is extended
unsigned int quoted : 1;	value is quoted

MIME parameters can be extracted from a parameter list by a call to **mparse_get_parameter**:

```
int mparse_get_parameter(const struct mparse_message *, const struct mparse_token *, const char *, char *, size_t, struct mparse_parameter **);
```

The arguments are: a pointer to the current **mparse_message** structure, a pointer to the **mparse_token** list comprising the MIME parameters, (usually the *content_parameters* member of the **mparse_entity** structure) a character string corresponding to the desired parameter name (e.g. "boundary"), a pointer to a character array where the corresponding parameter value is to be written, and the size of the array, and a pointer to a **mparse_parameter** structure pointer which will be updated to point to the *initial mparse_parameter* structure (for charset and language information; a NULL pointer may be passed to **mparse_get_parameter** if this information is not required). The return value from **mparse_get_parameter** is the number of characters returned (excluding the terminating '\0'), or, if the buffer is too small, the size of the buffer (including terminating '\0') required to hold the parameter value string.

The integer returned by

```
int mparse_parameters(const struct mparse_token *);
```

is a count of the number of parameters in the **mparse_token** list comprising the MIME parameters linked mparse_token structure which is supplied as an argument (usually the *content_parameters* member of the **mparse_entity** structure).

Convenience and Utility Functions

The following function provides information regarding a **message**:

```
int mparse_ends_with_crlf(struct mparse_message *);
```

returns non-zero if the message ends with a CRLF. MIME multipart messages do not necessarily end with a CRLF; the closing boundary delimiter itself does not end with CRLF, and the optional epilogue following the close delimiter also might not end with CRLF,

The following functions provide quick tests for information regarding an **mparse_entity** structure:

```
int mparse_has_close_delimiter(const struct mparse_entity *);
```

returns non-zero for the last part of a multipart entity.

```
int mparse_is_application(const struct mparse_entity *);
```

returns non-zero for an application MIME media type entity.

```
int mparse_is_audio(const struct mparse_entity *);
```

returns non-zero for an audio MIME media type entity.

int mparse_is_composite(const struct mparse_entity *);

returns non-zero for a multipart or message composite MIME entity.

int mparse_is_dsn(const struct mparse_entity *);

returns non-zero for a Delivery Status Notification message. Return value is 1 for the per-message fields structure, and 2 for the per-recipient fields structures. Note that 0 is returned for the enclosing message/delivery-status structure, which is not considered part of the DSN *per se*.

int mparse_is_epilogue(const struct mparse_entity *);

returns non-zero for the last part of a multipart entity if an epilogue is present.

int mparse_is_external(const struct mparse_entity *);

returns non-zero for a message/external-body MIME media type.

int mparse_is_mdn(const struct mparse_entity *);

returns non-zero for a Message Disposition Notification message (not its enclosure).

int mparse_is_mdn_report(const struct mparse_entity *);

returns non-zero for a multipart/report holding a report-type of disposition-notification.

int mparse_is_message_type(const struct mparse_entity *);

returns non-zero for a MIME message composite media type.

int mparse_is_multipart(const struct mparse_entity *);

returns non-zero for a MIME multipart composite media type.

int mparse_is_plain(const struct mparse_entity *);

returns non-zero for a text/plain media type.

int mparse_is_preamble(const struct mparse_entity *);

returns non-zero for the first part of a multipart entity if a preamble is present.

int mparse_is_report(const struct mparse_entity *);

returns non-zero for a MIME multipart/report media type.

int mparse_is_rfc822(const struct mparse_entity *entity)

returns non-zero if the **entity** structure is encapsulated in a message/rfc822 encapsulation or equivalent (e.g. the obsolete "message/news" media type).

int mparse_is_signed(const struct mparse_entity *);

returns non-zero for any media type enclosed (at any level) as the first part of a multipart/signed entity. These may not be modified as that would prevent signature verification. Application code should operate on a copy of such an **mparse_entity** structure if it is necessary to modify it (e.g. decode transfer encoding).

int mparse_is_text(const struct mparse_entity *);

returns non-zero for any text media type.

int mparse_is_mtsn(const struct mparse_entity *);

returns non-zero for a Message Tracking Status Notification message. Return value is 1 for the per-message fields structure, and 2 for the per-recipient fields structures. Note that 0 is returned for the enclosing message/tracking-status structure, which is not considered part of the MTSN *per se*.

int mparse_is_multiheader(const struct mparse_entity *);

returns non-zero for an entity which consists of one of a set of multiple fields (DSNs and MTSNs).

int mparse_part(const struct mparse_entity *);

returns -1 if the type is not message/partial. Returns 0 (N.B.) for the encapsulated part of the first part of a partial message series (the encapsulation of the first part yields a return of 1), and returns an integer corresponding to the part number of subsequent parts.

int mparse_report_part(const struct mparse_entity *);

returns 0 if the structure pointed to is not part of a multipart/report. Returns a negative number if the multipart/report is malformed (fewer than two or more than three parts). Returns 1 for the first part (intended to be human readable text; see RFC 1892). Returns 2 for the machine-readable part (DSN or MDN). Returns 3 for the optional returned message or returned header fields. Note that zero is returned for the multipart/report enclosure.

The following function provides information regarding a field:

int mparse_in_body(const struct mparse_field *);

returns 1 if the field is part of an **entity** body, 0 if it is not.

The following functions provide information regarding a token:

int mparse_is_ws(const struct mparse_token *);

returns 1 if the token is whitespace, 0 if it is not.

int mparse_has_whitespace(const struct mparse_token *);

returns 1 if the logical token contains any whitespace tokens, 0 if there are none.

int mparse_is_fws(const struct mparse_token *);

returns 1 if the token is whitespace or the MPARSE_TOKEN_CRLF which is part of line folding, 0 if it is not.

int mparse_has_folding(const struct mparse_token *);

returns 1 if the logical token contains any line folding tokens, 0 if there are none.

int mparse_is_cfws(const struct mparse_token *);

returns 1 if the token is part of CFWS, 0 if it is not.

int mparse_has_comment(const struct mparse_token *);

returns 1 if the logical token contains any comments, 0 if there are none.

int mparse_is_trailing_ws(const struct mparse_token *);

returns 1 if the token is whitespace at the end of a line or at the end of a message body, 0 if it is not.

A string with specified length may be characterized with the following functions:

int mparse_is_atom(const unsigned char *c, size_t len);

returns 1 if the string is a valid atom, 0 if it is not.

int mparse_is_qtext(const unsigned char *c, size_t len);

returns 1 if the string is valid qtext, 0 if not.

int mparse_is_mime_charset(const unsigned char *c, unsigned int len);

returns 1 if the string is a valid name for a MIME charset, else it returns 0.

The following functions return information about a single character, much like the C library *ctype* functions:

int mparse_isspecial(int c);

returns nonzero if the character is a special character as defined in RFCs 822 and 2822.

int mparse_isaspecial(int c);

returns nonzero if the character is an aspecial other than the percent sign, %.

int mparse_istspecial(int c);

returns nonzero if the character is a tspecial (i.e. must be quoted if used in a parameter value).

int mparse_isldh(int c);

returns non-zero for any letter, digit, or hyphen (i.e. valid characters for a domain name component).

int mparse_islwsp(int c);

returns nonzero for linear whitespace characters.

int mparse_is_ew_char(int c, unsigned int context);

returns nonzero if the character is valid in an encoded-word in the given context.

int mparse_isnewsgc(int c);

returns nonzero if the character is valid in a newsgroup or distribution name component.

int mparse_istokenc(int c);

returns nonzero if the character is valid in a MIME token.

int mparse_isboundaryc(int c);

returns nonzero if the character is valid in a MIME boundary string.

int mparse_is_uri_reserved(int c);

returns nonzero if the character is a URI reserved character (RFC 2396).

int mparse_is_uri_excluded(int c);

returns nonzero if the character is not permitted unescaped in a URI (RFC 2396).

int mparse_is_uri_authority_reserved(int c);

returns nonzero if the character is a URI reserved character in the authority URI component (RFC 2396).

int mparse_is_uri_path_reserved(int c);

returns nonzero if the character is a URI reserved character in the path URI component (RFC 2396).

int mparse_is_uri_query_reserved(int c);

returns nonzero if the character is a URI reserved character in the query URI component (RFC 2396).

The following functions provide access to base64 encoding and decoding translation:

unsigned char mparse_translate_b64(int);

returns the base64 character corresponding to the integer argument (range 0 to 63).

int mparse_encode_b64_word(const unsigned char *, unsigned int, unsigned int, unsigned int, char *, int);

encodes a word pointed to by the first argument, with length given by the second argument into a buffer with size given by the last two arguments. The two other unsigned integer arguments specify the number of leading and trailing space characters to be encoded with the word. The return value is the number of octets in the result, or if the buffer is too small, the number of octets necessary to hold the result.

int mparse_decode_b64_word(struct mparse_entity *, struct mparse_token *, char *, int, const char *, int);

Given an optional entity and token (for error reporting) and a buffer and its length (first four arguments), the base64-encoded text with length given by the last two arguments is decoded into the buffer. The return value is the number of octets in the result, or if the buffer is too small, the number of octets necessary to hold the result.

There is a required **micalg** parameter used with the multipart/signed media type. It is also used in conjunction with the Received-content-MIC header field.

Application programmers who wish to enumerate standard micalg values (for example, to construct a menu) can call:

```
const struct mparse_name_val *mparse_micalg(int n);
```

sequentially with an integer argument beginning with 1. **mparse_micalg** will return a pointer to the structure in sequence according to the micalg name returning a *NULL* pointer when the end of the list is reached.

IANA has a registry of context values for the Message-Context header field.

Application programmers who wish to enumerate standard context values (for example, to construct a menu) can call:

```
const struct mparse_name_val *mparse_context(int n);
```

sequentially with an integer argument beginning with 1. **mparse_context** will return a pointer to the structure in sequence according to the context value name, returning a *NULL* pointer when the end of the list is reached.

Mailboxes and Message-IDs

```
int mparse_count_mailboxes(const struct mparse_token *list);
```

returns the number of mailboxes (or message-ids) in *list*, which can be a series or delimited list or a single item.

```
int mparse_mailbox_components(const struct mparse_token *list, int n, const struct mparse_token **display_name, const struct mparse_token **bracket, const struct mparse_token **route, const struct mparse_token **local_part, const struct mparse_token **at, const struct mparse_token **domain);
```

finds the *n*th mailbox (or message-id) in *list* and sets pointers to the components (if they exist): display name, opening angle bracket, route (which may be an RFC 822 list or an RFC 733 string of domains), local-part, @ delimiter, and domain. It returns -1 on serious errors with *errno* set, and returns *n* on success. If there are fewer than *n* mailboxes (or message-ids), it returns the number present in *list*.

int mparse_interpolate_components(struct mparse_token *t, int n, char *buf, int sz, unsigned int components);

may be used to interpolate mailbox (or message-id) components into a buffer *buf*. The *components* argument determines which components of the *n*th mailbox are used according to the following values (defined in *mparse.h*) which should be bitwise ORed:

symbolic name	description
MPARSE_INTERPOLATE_DISPLAY_NAME	the display name (if present)
MPARSE_INTERPOLATE_BRACKETS	angle brackets
MPARSE_INTERPOLATE_ROUTE	RFC 822 source route or extra domains (RFC 733)
MPARSE_INTERPOLATE_LOCAL_PART	the local-part of the address
MPARSE_INTERPOLATE_AT_DELIMITER	the @ delimiter between local-part and domain
MPARSE_INTERPOLATE_DOMAIN	the domain part of the address
MPARSE_INTERPOLATE_INCLUDE_COMMENTS	any comments associated with the mailbox components listed above
MPARSE_INTERPOLATE_EXCESS_WS	any excess whitespace (otherwise compressed to a single space character outside brackets or eliminated within brackets including within local-part, domain, and around the @)

It is possible to get nonsense by specifying certain combinations of values; normally the values (MPARSE_INTERPOLATE_BRACKETS | MPARSE_INTERPOLATE_LOCAL_PART | MPARSE_INTERPOLATE_AT_DELIMITER | MPARSE_INTERPOLATE_DOMAIN) would be specified to obtain a full address for use with a transport protocol such as SMTP, MPARSE_INTERPOLATE_DOMAIN alone to obtain the domain name for use with DNS (e.g. for MX lookups), and (MPARSE_INTERPOLATE_DISPLAY_NAME | MPARSE_INTERPOLATE_BRACKETS | MPARSE_INTERPOLATE_LOCAL_PART | MPARSE_INTERPOLATE_AT_DELIMITER | MPARSE_INTERPOLATE_DOMAIN) for use in header fields. MPARSE_INTERPOLATE_DISPLAY_NAME alone (or possibly with MPARSE_INTERPOLATE_INCLUDE_COMMENTS) might be used by user agents to extract the display name for presentation.

The return value is the number of octets in the result, or if the buffer is too small, the number of octets necessary to hold the result.

Replies and Follow-ups

Sometimes one would like to know if a message is or is not a reply to another message. The function

int mparse_is_reply(struct mparse_message *message);

returns 0 if *message* is definitely not a reply, returns 1 if *message* is definitely a reply to another message, and returns -1 if it is not possible to definitively determine if *message* is or is not a reply.

Likewise

int mparse_is_auto_reply(struct mparse_message *message);

similarly indicates whether *message* is an automatic reply (not counting DSNs and MDNs).

The following functions may be used to extract addresses for replies to a message or for follow-ups to newsgroups:

int mparse_envelope_sender(struct mparse_message *message, char *buf, int sz, unsigned int mode);

places the envelope sender address (from the Return-Path field of *message*) in *buf*, returning the number of characters copied if *buf* is sufficiently large (as specified by *sz*), or the required size if *buf* is a *NULL* pointer or is too small, or a negative value (with *errno* set) on error. The *mode* argument provides some control over interpolation of the address (see the description for the *mparse_tokens_string* function), though unfolding and normalization of "at" are always performed.

int mparse_error_addresses(struct mparse_message *message, char *buf, int sz, unsigned int mode);

places the address for an error reply in *buf*, as above. The error reply address is the envelope sender address if present, the Sender field if there is no envelope sender address, or the From field if there is no envelope sender or Sender.

int mparse_sender_mailbox(struct mparse_message *message, char *buf, int sz, unsigned int mode);

copies the sender mailbox to *buf*, as above. The sender mailbox is the mailbox specified in the Sender field if present or the From field if there is no Sender field.

int mparse_author_mailboxes(struct mparse_message *message, char *buf, int sz, unsigned int mode);

copies the authors' mailboxes to *buf*, as above. The authors' mailboxes are the mailboxes specified in the From field if present.

int mparse_reply_addresses(struct mparse_message *message, char *buf, int sz, unsigned int mode);

copies normal reply primary addresses to *buf*, as above. Normal replies go to the addresses specified in the Reply-To field if present, the From field if there is no Reply-To field.

int mparse_cc_mailboxes(struct mparse_message *message, char *buf, int sz, unsigned int mode);

copies the Cc field mailboxes to *buf*, as above.

int mparse_to_mailboxes(struct mparse_message *, char *, int, unsigned int);

copies the To field mailboxes to *buf*, as above.

int mparse_mdn_addresses(struct mparse_message *message, char *buf, int sz, unsigned int mode);

extracts the addresses from the Disposition-Notification-To field into *buf*, as above.

int mparse_auto_response_address(struct mparse_message *message, char *buf, int sz, int is_service, unsigned int mode);

uses the Return-Path address (if present), primarily for use when generating reply fields for use with an Auto-Submitted field. If *is_service* is non-zero, The From field will be used if it contains a single mailbox and there is no Return-Path. If there are multiple mailboxes in the From field, *errno* is set to **MPARSE_ERRNO_EMULTIADDRESS**, the addresses are placed in *buf* (if not NULL and if sufficiently large) and a negative value (-1 times the number of characters) is returned.

int mparse_list_post_uri(struct mparse_message *message, char *buf, int sz, unsigned int mode, unsigned int nskip);

places a list-post URI in *buf*, as above. The unsigned integer argument *nskip* indicates how many bracketed URIs should be skipped, starting at the beginning of the list in the List-Post field, if present. Zero is returned if no bracketed URI is available (including if there is no List-Post field).

int mparse_followup_newsgroups(struct mparse_message *message, char *buf, int sz, unsigned int mode);

places a list of followup newsgroup names in *buf*, as above. There are none if the message contains a Followup-To field with only the "poster" keyword. If there is a Followup-To field it specifies the followup newsgroups. If there is no Followup-To field, the followup newsgroups are taken from the Newsgroups field if present. In this case, default processing (extendable via *mode*) includes replacement of each comment with a space character.

int mparse_followup_addresses(struct mparse_message *message, char *buf, int sz, unsigned int mode);

copies followup addresses to *buf*, as above. Addresses are taken from the Followup-To field if present and if it contains addresses, and/or addresses are taken from the Reply-To or From fields if the Followup-To field contains the "poster" keyword. If no Followup-To field is present, reply addresses are taken from Reply-To or From fields as described above for reply addresses.

```
int mparse_message_identifier(struct mparse_message *message, char *buf, int sz, unsigned int mode);
```

copies the message identifier from the Message-ID header field to *buf*, as above.

Repair

When generating messages and when the context `MPARSE_SECONDARY_ROLE_REPAIR` flag is in effect, some errors may be repaired. There may be some errors which are beyond the ability to repair. Others can be repaired, and the following functions initiate repairs on message components:

```
struct mparse_token *mparse_fix_token_errors(struct mparse_entity *, struct mparse_field *, struct mparse_token *, struct mparse_token **, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int, unsigned int);
```

repairs certain errors and/or eliminates warnings recorded for a specified token. The entity and field for the token should be specified along with the token. A pointer to a token structure pointer must be supplied, if the line containing the token must be folded, the start of the following line will be at the token pointed to by the pointer on successful return. The charset and language tag string may be supplied to indicate charset and/or language in case some content must be encoded. The final four unsigned integer arguments indicate, in turn, whether the token is in a message body part (non-zero), the `MPARSE_FIX_*` values to be repaired, whether to repair items for which warnings (but no errors) have been issued (if non-zero), and whether to consider issues relative to all RFCs (if non-zero) or only the RFC modes in effect for the message (if zero). Because a token might be deleted in the process of making repairs, the return value is a pointer to the next physical token after the specified token after repairs have been made.

```
int mparse_fix_field_errors(struct mparse_entity *, struct mparse_field *, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int, unsigned int);
```

operates similarly for fields (including handling all tokens in the field). The entity and field are specified, followed by a maximum line length for line folding. The remaining arguments (beginning with charset) are as above.

```
int mparse_fix_entity_field_errors(struct mparse_entity *, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int);
```

makes repairs for all fields in a given entity (and including all tokens in all fields). The entity, line length limit, etc. are specified as above.

```
int mparse_fix_entity_body_errors(struct mparse_entity *, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int);
```

does the same for body lines and tokens within an entity.

```
int mparse_fix_entity_misc_errors(struct mparse_entity *, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int);
```

makes repairs for entity delimiter and separator lines, fields, tokens, and other entity error conditions (e.g. missing fields).

```
int mparse_fix_entity_errors(struct mparse_entity *, int, const struct mparse_charset *, const char *, unsigned int, unsigned int, unsigned int);
```

repairs all of the above types of errors in an entire entity.

It may be desirable to locate an **mparse_error** structure corresponding to some remedy which should be applied.

```
struct mparse_error *mparse_find_field_remedy(const struct mparse_message *, const struct mparse_field *, unsigned int, unsigned int, int);
```

returns a pointer to such a structure corresponding to a specified field if one exists. The first unsigned integer argument is constructed from the `MPARSE_FIX*` values to be considered. The other unsigned integer arguments determines whether to consider issues relative to all RFCs (if non-zero) or only the RFC modes in effect for the message (if zero). The last argument sets a minimum severity (`MPARSE_REQUIREMENTS_SHOULD`, `MPARSE_REQUIREMENTS_MUST`, etc.) to be considered.

```
int mparse_field_needs_remedy(const struct mparse_message *, const struct mparse_field *, unsigned int, unsigned int, int);
```

simply returns a positive non-zero value if there exists an **mparse_error** structure corresponding to the arguments as detailed above. It returns zero if there is no such structure and a negative value if the supplied arguments are not valid.

The functions

```
struct mparse_error *mparse_find_token_remedy(const struct mparse_message *, const struct mparse_token *, unsigned int, unsigned int, int);
```

and

```
int mparse_token_needs_remedy(const struct mparse_message *, const struct mparse_token *, unsigned int, unsigned int, int);
```

operate similarly for tokens.

Miscellany

Some fields may contain Uniform Resource Identifiers (URIs) as defined in RFC 2396. RFC 2396 Appendix B describes parsing a URI into component parts, which may be performed by calling:

```
int mparse_parse_uri(const char *s, char *pscheme, size_t *pscheme_sz, char *pauthority, size_t *pauthority_sz, char *ppath, size_t *ppath_sz, char *pquery, size_t *pquery_sz, char *pfragment, size_t *pfragment_sz);
```

The URI is supplied as a nul-terminated character string *s*. Component strings are copied to corresponding `char *` arguments (if not NULL), up to the size given by the corresponding `size_t *` arguments (which must not be NULL and into which the length (not including terminating `'\0'`) is written if the supplied size was sufficient; if insufficient, the required size (including space for a terminating `'\0'`) is written). Return value is -1 on error, with *errno* set, zero if all sizes are adequate, and a positive value if one or more sizes are insufficient, or if there is an error involving regular expressions for parsing. N.B. path may include path parameters.

There are functions for conversion between messages and *mailto* URIs as defined in RFC 2368:

```
int mparse_instantiate_mailto(const char *s, struct mparse_message *message, unsigned int is_html);
```

turns a *mailto* URI in character string *s* into message content in **message**. If the *mailto* URI has been taken from HTML context, **is_html** should be set to a non-zero value. The return value is negative, with *errno* set appropriately in the event of an error. A zero return value indicates that all went well with the conversion, while a positive return value indicates that some content in the resulting message has generated at least one error or warning with respect to the RFCs enabled via the **modes** set in **message**.

int mparse_prepare_mailto(const struct mparse_message *message, char *buf, size_t sz, unsigned int is_html);

generates a *mailto* URI in **buf** from the message content in **message**. If the *mailto* URI will be used in an HTML context, **is_html** should be set to a non-zero value. The return value is negative with *errno* set appropriately in the event of an error. Otherwise the number of characters placed into **buf** is returned (not including the terminating ASCII NUL if the buffer is sufficiently large); if that value is greater than the supplied **sz** value, the buffer is too small to hold the *mailto* URI, and the return value indicates the minimum buffer size that will accommodate the URI (including the terminating ASCII NUL).

Application programmers who wish to enumerate message MIME parts may use

int mparse_part_string(const struct mparse_entity *, char *, size_t);

which places an RFC 3501 IMAP-compatible part number for the **struct mparse_entity** pointed to by the first argument in the buffer pointed to by the second argument, and with size given by the third argument. The return value is the number of characters in the string, or the required buffer size if the buffer is too small.

Various registered protocol element names can be tested, and in some cases enumerated, by calling functions similar in operation to those already described for specific protocol elements. The functions described below are automatically called during parsing of messages (including when fields are generated via *mparse_insert_field*) and need not be called by application authors. These functions are provided:

const char *mparse_MTA_name_typ_entry (register const char *str, register unsigned int len);

validates *str* (with length *len*) as a registered MTA-name-type (RFC 3464). It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

MIME message/external-body provides for access-types. A potential access-type can be validated by calling

const struct mparse_name_val *mparse_access_entry(const char *, unsigned int);

It returns a structure which holds the canonical name and some additional information (see *mparse.h*).

const struct mparse_name_val *mparse_access_type(int n);

may be used to enumerate access-types.

Message Disposition Notifications (RFC 3798) provides for action-modes in the Disposition MDN field.

const char *mparse_act_mode_entry(const char *, unsigned int);

may be called to determine if a string of a given length is a valid action-mode value. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

There are registered action-value names associated with the Action per-recipient field of Delivery Status Notifications (RFC 3464) and Message Tracking Status Notifications (RFC 3XXX).

const struct mparse_name_val *mparse_action_entry(const char *, unsigned int);

validates strings as action-value names. It returns a structure which holds the canonical name and some additional information (see *mparse.h*). A program may enumerate those action-value names (e.g. to create a menu) by calling the function

const struct mparse_type *mparse_action(int n);

with successive integer arguments beginning with 1. Each call will return a pointer to a *mparse_name_val* structure which represents an action-value; when all action-values have been enumerated, **mparse_action** returns a NULL pointer.

DSN and MTSN Original-Recipient and Final-Recipient fields use an address-type value.

const char *mparse_address_type_entry(const char *, unsigned int);

validates strings as address-type names. It returns a pointer to the canonical name. There is no corresponding enumeration function.

The Autosubmitted header field defined in RFC 2156 contains an autosubmitted value.

const struct mparse_name_val *mparse_autosub_entry(const char *, unsigned int);

validates a string with a given length as an autosubmitted value name. It returns a structure which holds the canonical name and some additional information (see `mparse.h`). There is no corresponding enumeration function.

Boolean values, i.e. the strings "true" and "false", may appear in some contexts. The function

const struct mparse_name_val *mparse_boolean_entry(const char *, unsigned int);

recognizes those strings and returns a structure which holds the canonical name and some additional information (see `mparse.h`). There is no corresponding enumeration function.

RFC 3458 defines a Message-Context header field which can contain registered context values.

const struct mparse_name_val *mparse_commparse_text_entry(const char *, unsigned int);

recognizes those strings and returns a structure which holds the canonical name and some additional information (see `mparse.h`).

const struct mparse_name_val *mparse_context_type(int);

is the corresponding enumeration function.

Usenet control messages use a control keyword.

const struct mparse_control *mparse_control_entry(const char *, unsigned int);

recognizes those keywords and returns a structure which holds the canonical name and some additional information (see `mparse.h`).

const char *mparse_control_name(int);

is the corresponding enumeration function.

Specification of dates using the date-time production of RFC 822 and RFC 2822 grammar provides for optional specification of the day-of-week.

const struct mparse_name_val *mparse_day_entry(const char *, unsigned int);

recognizes valid day-of-week strings and returns a structure which holds the canonical name and some additional information (see `mparse.h`). While there is no corresponding enumeration function, the global character string array `mparse_dows` provides access to the canonical strings (index 0 corresponds to Sunday, etc.).

DSN Diagnostic-Code fields use a diagnostic-type value.

const char *mparse_diagnostic_t_entry(const char *, unsigned int);

may be called to determine if a string of a given length is a valid diagnostic-type value. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

Message Disposition Notifications (RFC 3798) provides for disposition-modifiers in the Disposition MDN field.

const struct mparse_name_val *mparse_disp_mod_entry(const char *, unsigned int);

may be called to determine if a string of a given length is a valid disposition-modifier value. It returns a structure which holds the canonical name and some additional information (see `mparse.h`). There is no corresponding enumeration function.

The MDN RFC also provides for Disposition-Notification-Options parameters which are registered with IANA.

const struct mparse_name_val *mparse_disp_not_opt_entry(const char *, unsigned int);

may be called to determine if a string of a given length is a valid Disposition-Notification-Options parameter. It returns a structure which holds the canonical name and some additional information (see mparse.h). There is no corresponding enumeration function.

The MDN RFC provides for Disposition types which are registered with IANA.

const char *mparse_dtype_entry(const char *, unsigned int);

validates a string against known disposition types. It returns a pointer to the canonical form of the name.

The keywords WHERE and END may be used in constructing filters (RFC 2533)

const struct mparse_name_val *mparse_filter_entry(const char *, unsigned int);

recognizes these keywords. It returns a structure which holds the canonical name and some additional information (see mparse.h). There is no corresponding enumeration function.

Media feature tags are also defined in RFC 2533 and are registered by IANA.

const char *mparse_ftag_entry(const char *, unsigned int);

recognizes registered media feature keywords. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

Disposition-Notification-Options mparse_parameters use "importance" keywords to distinguish required and optional support for parameters.

const char *mparse_importance_entry(const char *, unsigned int);

recognizes valid importance keywords. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

The MIXER RFCs defined an Importance header field.

const struct mparse_name_val *mparse_importance2_entry(const char *, unsigned int);

recognizes valid keywords used with that field. It returns a structure which holds the canonical name and some additional information (see mparse.h). There is no corresponding enumeration function.

The MIXER RFCs also defined a Message-Type header field.

const struct mparse_name_val *mparse_message_type_entry(const char *, unsigned int);

recognizes valid phrases used with that field. It returns a structure which holds the canonical phrase and some additional information (see mparse.h). There is no corresponding enumeration function.

RFC 3335 defines a Received-content-MIC header field which contains a message integrity check (MIC) algorithm (micalg) name. These micalg names may also appear in a Disposition-Notification-Options header field which is used in conjunction with MDNs for secure business data interchange.

const struct mparse_name_val *mparse_micalg_entry(const char *, unsigned int);

recognizes valid micalg names. It returns a structure which holds the canonical name and some additional information (see mparse.h).

const struct mparse_name_val *mparse_micalg(int);

is the corresponding enumeration function.

Standardized alphabetic names are used to designate months in date-time specifications.

const struct mparse_name_val *mparse_month_entry(const char *, unsigned int);

recognizes valid month names and returns a structure which holds the canonical name and some additional information (see `mparse.h`). While there is no corresponding enumeration function, the global character string array `mparse_mons` provides access to the canonical strings (index 0 corresponds to January, etc.).

Newsgroup names have certain restrictions on allowable newsgroup components, and there are some keywords that may be used in place of newsgroup names in some contexts.

const struct mparse_name_val *mparse_newsgroups_entry(const char *, unsigned int);

determines if a string is such a keyword or forbidden name or component. It returns a structure which holds the canonical name and some additional information (see `mparse.h`). There is no corresponding enumeration function.

The MIXER RFCs defined a Priority header field.

const struct mparse_name_val *mparse_priority_entry(const char *, unsigned int);

recognizes valid names used with that field. It returns a structure which holds the canonical phrase and some additional information (see `mparse.h`). There is no corresponding enumeration function.

The MIXER RFCs also defined several header fields which are used to indicate whether or not some end-to-end signaling feature is to be prohibited.

const struct mparse_name_val *mparse_prohibition_entry(const char *, unsigned int);

recognizes valid names used with those fields. It returns a structure which holds the canonical name and some additional information (see `mparse.h`). There is no corresponding enumeration function.

The original SMTP RFC, RFC 788, specified several protocol names to be used with the Mail-From header field.

const char *mparse_protocol_entry(const char *, unsigned int);

recognizes valid protocol names for that context. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

Uniform Resource Indicators (URIs) specify a particular access scheme. Such schemes are registered by IANA.

const char *mparse_schemes_entry(const char *, unsigned int);

recognizes registered scheme names. It returns a pointer to the canonical form of the name.

const char *mparse_scheme(int);

is the corresponding enumeration function.

Message Disposition Notifications (RFC 3798) provides for sending-modes in the Disposition MDN field.

const char *mparse_sending_entry(const char *, unsigned int);

may be called to determine if a string of a given length is a valid sending-mode value. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

The MIXER RFCs defined a Sensitivity header field.

const struct mparse_name_val *mparse_sensitivity_entry(const char *, unsigned int);

recognizes valid names used with that field. It returns a structure which holds the canonical phrase and some additional information (see `mparse.h`). There is no corresponding enumeration function.

The MIXER RFCs also defined an X400-Received header field.

const struct mparse_name_val *mparse_x400act_entry(const char *, unsigned int);

recognizes valid mparse_action names used with that field. It returns a structure which holds the canonical phrase and some additional information (see mparse.h). There is no corresponding enumeration function.

The MIXER RFCs also defined an X400-Content-Type header field.

const char *mparse_x400ct_entry(const char *, unsigned int);

recognizes the sole valid name used with that field. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

The MIXER RFCs defined encoded-information-type keywords.

const struct mparse_name_val *mparse_x400eit_entry(const char *, unsigned int);

recognizes valid encoded-information-type keyword names. It returns a structure which holds the canonical name and some additional information (see mparse.h). There is no corresponding enumeration function.

Numeric time zone offsets (from UTC) and standardized alphabetic names are used in date-time specifications.

const struct mparse_name_val *mparse_zones_entry(const char *, unsigned int);

recognizes valid offsets and standardized names used in date-time specifications. It returns a structure which holds the canonical name and some additional information (see mparse.h). There is no corresponding enumeration function.

Calling Line Identification for Voice Mail Messages (RFC 3939) provides for a numbering plan option in the Caller-ID field.

const char *mparse_numbering_entry(const char *str, unsigned int len);

validates str (with length len) as a registered numbering plan. It returns a pointer to the canonical form of the name. There is no corresponding enumeration function.

Two functions are provided for interpolating numbers into a buffer without the performance overhead of *snprintf*.

size_t mparse_n_print(char *, size_t, int);

puts a string representation of a decimal integer into the character array given by the first two arguments (start and length).

size_t mparse_un_print(char *, size_t, unsigned int);

Does the same for unsigned integers.

const char mparse_digits[];

is a global character array where each element is the ASCII decimal digit corresponding to the array index.

Hacking

Suppose that you wish to add support for some new field. The first thing to do is to determine if that's really wise. There is a tendency for novices to approach any messaging issue with "let's invent a header field to do...". There are frequently better approaches. The second step is to determine whether to provide support directly in `mparse`, or at the application level via the hooks for extension and user-defined fields. Unless there is a high probability that the field will be standardized, it may be best to use the hooks rather than hack `mparse`. Assuming that one decides to proceed, the next thing required is a syntax specification for the field in a form similar to RFC 2822's ABNF. Additional required information is an understanding of where and how often the field may appear (e.g. it may be a mandatory DSN per-recipient field). Other field characteristics, such as whether or not a Resent- form of the field is to be recognized (see the `mparse_field_state` structure flags in `mparse.h`) should also be determined. Having collected the necessary information, one can proceed to modify the related files:

- `mparse.h.in`: Processing hooks may be added; these are called when the field is processed. The hook function should take an integer (indication of errors in the field as parsed) as its first argument, a pointer to the relevant `mparse_field` structure as the second argument, and may have additional arguments as pointers to interesting `mparse_token` structures,
- `hooktest.c`: Arrangements should be made to recognize added hooks and to generate some appropriate output (often simply echoing the field).
- `mparse.y`: Grammar rules for the field need to be added, and should be referenced by an entry in the list of fields which appears prior to the individual field grammar rules. See the existing field grammar rules for examples. Typically there should be an error-handling rule and one rule or more for normal field recognition. Each of the rules will typically call the function `mparse_install_and_process_field` with appropriate arguments (see the function definition in the file `parse2.c` or the prototype in the header file `mparse.h`). Ideally, the grammar should not include a lot of C code in the corresponding rule action; the idea is to put processing code in `parse2.c` and reserve `mparse.y` for grammar rules per se. Of course, there are exceptional cases where some detailed action code may be necessary. There are a number of non-terminal productions defined in the grammar file; these should be used where applicable rather than attempting to reinvent the wheel. Another principle to bear in mind when formulating the grammar rules is "In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior" (RFC 791). In `mparse`, this is achieved by making the grammar as liberal as practical and detecting and flagging errors and dubious constructs (which can be repaired during field generation).
- `parse2.c`: Detailed field-specific actions, error-checking, etc. are handled in the function `mparse_install_and_process_field`. See the existing code there for standard fields for examples of the type of processing performed. A check for proper context of the field may be added to the `check_field_context` function, and field count checks and cross-checks may be added to the `mparse_header_end` function.
- `fields.gperf`: The data corresponding to the `mparse_field_state` structure is entered on a line. Lexical analyzer start condition is determined by the field grammar; see the existing field definitions and the lexical analyzer rules (`mlex.l`).
- `mparse.0`: The hooks and any new functions related to the field should be documented.
- `fix_fold.c`: Handling of the field during repair should be incorporated into the `fix_*` functions.

If there are any potential errors that should be flagged, it may be necessary to provide error message strings and/or references to a particular section of a particular standards document. Macros used to reference internationalized strings are found in file `msg.h`; the macros expand to an index integer which is used to retrieve the corresponding text from the appropriate language variant of the messages stored in `msg_lang.gperf` for text that may appear in various languages.

In rare cases, it may be necessary to modify or extend the lexical analyzer (`mlex.l`).

`mparse` should be rebuilt and the regression tests run after modifying the relevant files. That is accom-

plished by entering the command **make regression**. The program **hooktest** should be used to test proper recognition of the field (including detection of and suitable diagnostics for illegal input).

Debugging

```
void mparse_dump_token(FILE *, const struct mparse_token *);
```

may be used to print information about a physical token and its attributes to the specified stdio FILE.

```
void mparse_dump_tokens(FILE *, const struct mparse_token *);
```

prints information for a logical token series.

Maintenance

IANA and other authorities maintain sets of keywords, language tags, charset names, etc. which may be periodically updated. **Mparse** can be rebuilt to incorporate such updates. The *makefile* (specifically the file *makefile* which contains the recipes for building **mparse**) automates much of the update, provided suitable tools are available. The high-performance keyword hash tables which are generated from the reference sets are regenerated. Any differences due to updates are displayed and the rebuild of **mparse** stops. That is intentional so that the differences can be reviewed. Occasionally a typographical error is introduced into the reference information, and some of the tables incorporate ancillary information (e.g. flags indicating required Content-Type parameters associated with media types) which cannot be automatically generated from the reference information (somebody needs to manually review the defining registration information and determine whether flags need to be updated). When the files automatically generated from the reference information have been reviewed (and manually updated if necessary), the build process may be restarted to continue the update.

EXAMPLE

The following example demonstrates construction of a delivery status notification:

```
/* Description: demonstration program: generate a Delivery Status Notification message
*/
#define optind EFFIN_GCC_NONSENSE      /* work around gcc silliness */

#include <mparse.h>

#include <errno.h>          /* errno */
#include <string.h>        /* strerror, strlen */
#include <stdlib.h>        /* atoi */
#include <unistd.h>        /* access, R_OK */

#undef optind

#define OPTSTRING "a:r:s:t:"
#define USAGE "-r recipient -a action -s status -t to [file]"

struct counts {
    unsigned int errcount;
    unsigned int warncount;
};

static char *setopt(char *s, char **argv, int *poptind, unsigned int *perr,
    char **popt, const char *name)
{
    if (!*++s)
        s = argv[++*poptind];
    if (!s)
        (*perr)++;
    else {
        if (*popt) {
            (void)fprintf(stderr, "%s: %s already specified: %s0, argv[0],
                name, *popt);
            (*perr)++;
        } else
            *popt = s;
        for (; *s; s++)
            s--;
    }
    return s;
}

static int field_err(struct mparse_entity *entity, FILE * s, int line)
{
    if (entity) {
        struct mparse_field *f = entity->last_field;
```

```

    struct mparse_message *message = entity->message;

    (void)fprintf(s, "bad %s:0,
        f->state ? f->state->field_name : f->tokens->tok);
    message->suppress_errors = message->suppress_warnings = 0U;
    (void)mparse_field_error_messages(f, mparse_fwrite_wrapper, s);
    (void)mparse_process_and_free_message(message, -1);
}
return line;
}

static int header_err(struct mparse_entity *entity, FILE * s, int line)
{
    if (entity) {
        struct mparse_message *message = entity->message;

        (void)fprintf(s, "bad fields:0);
        message->suppress_errors = message->suppress_warnings = 0U;
        (void)mparse_entity_header_error_messages(entity, mparse_fwrite_wrapper,
            s);
        (void)mparse_process_and_free_message(message, -1);
    }
    return line;
}

static int copy_err(struct mparse_entity *entity, FILE * s, int line)
{
    (void)fprintf(s, "message copy error0);
    if (entity)
        (void)mparse_process_and_free_message(entity->message, -1);
    return line;
}

static int insertion_err(struct mparse_entity *entity1,
    struct mparse_entity *entity2, FILE * s, int line)
{
    (void)fprintf(s, "insertion error0);
    if (entity1)
        (void)mparse_process_and_free_message(entity1->message, -1);
    if (entity2)
        (void)mparse_process_and_free_message(entity2->message, -1);
    return line;
}

static int hook_end_of_message(struct mparse_message *message)
{
    struct mparse_entity *cpy, *msg, *top =
        (struct mparse_entity *) (message->userptr);

    cpy = mparse_copy_message(message, message->top);
    if (!cpy)
        return copy_err(top, stderr, __LINE__);
    else if (mparse_insert_entity(msg =
        mparse_encapsulate(cpy, "rfc822", 0), top, 0, 0))
        return insertion_err(msg, top, stderr, __LINE__);
    return 0;
}

/* args: -r recipient, -a action, -s status, -t to, [file] */
int main(int argc, char **argv)
{
    char buf[1024], buf2[128], *act, *recipient, *st, *to, *s;
    int c, optind;
    FILE *in;
    struct counts counts;
    struct mparse_field *h;
    struct mparse_message message;
    struct mparse_entity *p, *q, *r;
    struct mparse_debug debug;

    setvbuf(stdout, NULL, _IOLBF, 0);
    memset(&message, 0, sizeof(struct mparse_message));
    memset(&debug, 0, sizeof(struct mparse_debug));
    message.dbg = &debug;
    memset(&counts, 0, sizeof(struct counts));
    act = recipient = st = to = 0;
    /* parse option arguments; emulate getopt (too many implementation differences to rely on it) */
    for (optind = 1; (optind < argc) && (argv[optind][0] == '-'); optind++) {
        if (!strcmp(argv[optind], "--")) {
            optind++;
            break;
        }
    }
}

```

```

    }
    for (s = argv[optind] + 1; s && ((c = *s) != ' '); s++)
        switch (c) {
            case 'a':
                s = setopt(s, argv, &optind, &counts.errcount, &act,
                    "action");
                break;
            case 'r':
                s = setopt(s, argv, &optind, &counts.errcount, &recipient,
                    "recipient");
                break;
            case 's':
                s = setopt(s, argv, &optind, &counts.errcount, &st,
                    "status");
                break;
            case 't':
                s = setopt(s, argv, &optind, &counts.errcount, &to, "to");
                break;
            default:
                counts.errcount++;
                break;
        }
    }
    /* make sure mandatory arguments have been specified */
    if (!act || !recipient || !st || !to)
        counts.errcount++;
    else if (argc - optind > 1) { /* check file argument(s) */
        (void)fprintf(stderr, "%s: too many file arguments0, argv[0]");
        counts.errcount++;
    } else
        for (c = optind; c < argc; c++) { /* pass 0: quick check for readable input file */
            if (!strcmp(argv[c], "-"))
                continue;
            errno = 0;
            if (access(argv[c], R_OK)) {
                (void)fprintf(stderr, "%s: can't read %s: %s0, argv[0],
                    argv[c], strerror(errno));
                counts.errcount++;
            }
        }
    if (counts.errcount) {
        (void)fprintf(stderr, "%s: usage: %s %s0, argv[0], argv[0], USAGE);
        return __LINE__;
    }
    s = getenv("YYDEBUG"); /* same */
    if (s) /* as */
        message.gdebug = (atoi(s) ? 1U : 0U); /* Berkeley yacc */
    message.no_copy = 1U; /* don't echo: will print_message after assembly */
    message.suppress_errors = message.suppress_warnings = 1U;
    message.linelen = MPARSE_LIMIT_RECLINELEN;
    message.context = MPARSE_PRIMARY_ROLE_GENERATION;
    q = mparse_new_entity(&message);
    p = mparse_encapsulate(q, "delivery-status", 0); /* encapsulate before adding fields */
    mparse_gen_hostname(buf, sizeof(buf));
    if (mparse_insert_field(q, 0, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN,
        "Reporting-MTA", " dns", "; ", buf, 0))
        return field_err(q, stderr, __LINE__);
    return header_err(q, stderr, __LINE__);
    /* could add more per-message fields here */
    if (mparse_header_end(q))
        return header_err(q, stderr, __LINE__);
    q = mparse_new_entity(&message);
    if (mparse_insert_entity(q, p, 0, 0)) /* insert in encapsulation before adding fields */
        return insertion_err(q, p, stderr, __LINE__);
    if (mparse_insert_field(q, 0, 0, 0, 0, 1, MPARSE_LIMIT_ENCLINELEN,
        "Final-Recipient", " ", "rfc822", " ; ", recipient, 0))
        return field_err(q, stderr, __LINE__);
    if (mparse_insert_field(q, 0, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN, "Action",
        " ", act, 0))
        return field_err(q, stderr, __LINE__);
    if (mparse_insert_field(q, 0, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN, "Status",
        " ", st, 0))
        return field_err(q, stderr, __LINE__);
    /* could add more per-recipient fields here */
    if (mparse_header_end(q))
        return header_err(q, stderr, __LINE__);
    q = mparse_new_entity(&message);
    (void)mparse_append_body_line(q, "This is a DSN example.", "text", 4U,
        "plain", 5U, "us-ascii", 8U, 0, 0, 0U, 0, 0, 0,
        MPARSE_LIMIT_RECLINELEN);
    mparse_body_end(q);
    (void)mparse_parameter_string(&message, "report-type", "delivery-status",

```

```

    15U, 0, 0, buf2, sizeof(buf2));
r = mparse_create_multipart(p, "report", "report delimiter", buf2, 0, 0);
if (mparse_insert_entity(q, r, p, 0))
    return insertion_err(q, r, stderr, __LINE__);
/* fields for report */
if (mparse_insert_field(r, r->fields, 0, 0, 0, 1, MPARSE_LIMIT_ENCLINELEN,
    "From", " postmaster@", buf, 0))
    return field_err(r, stderr, __LINE__);
h = r->fields->next;
if (mparse_insert_field(r, h, 0, 0, 0, 1, MPARSE_LIMIT_ENCLINELEN, "To",
    " ", to, 0))
    return field_err(r, stderr, __LINE__);
mparse_gen_date_local(buf, sizeof(buf));
if (mparse_insert_field(r, h, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN, "Date",
    " ", buf, 0))
    return field_err(r, stderr, __LINE__);
if (mparse_insert_field(r, h, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN,
    "Subject", " Delivery Status Notification: ", act, 0))
    return field_err(r, stderr, __LINE__);
mparse_gen_message_id(buf, sizeof(buf));
if (mparse_insert_field(r, h, 0, 0, 0, 1, MPARSE_LIMIT_RECLINELEN,
    "message-id", " ", buf, 0))
    return field_err(r, stderr, __LINE__);
if (optind < argc) {
    struct mparse_message orig;
    struct mparse_hooks hooks;

    /* read and parse input file */
    memset(&orig, 0, sizeof(struct mparse_message));
    orig.context = MPARSE_PRIMARY_ROLE_ACCESS;
    orig.dbg = &debug;
    orig.no_copy = 1U; /* don't echo; will print_message after assembly */
    orig.suppress_errors = orig.suppress_warnings = 1U;
    orig.userptr = r;
    orig.timeout = 10.0;
    memset(orig.hooks = &hooks, 0, sizeof(struct mparse_hooks));
    hooks.hook_end_of_message = hook_end_of_message;
    if (!strcmp(argv[optind], "-"))
        in = stdin;
    else {
        in = fopen(argv[optind], "rb");
        if (!in) {
            (void)fprintf(stderr, "%s: can't open %s: %s0, argv[0],
                argv[optind], strerror(errno));
            (void)mparse_process_and_free_message(&message, -1);
            return __LINE__;
        }
    }
    (void)mparse_parse(&orig, in, stderr);
    if (in != stdin)
        fclose(in);
}
if (!message.top)
    return __LINE__;
(void)mparse_adjust_encoding(r);
(void)mparse_minimize_mime_fields(r);
if (mparse_header_end(r))
    return header_err(q, stderr, __LINE__);
(void)mparse_print_message(stdout, 0, &message, 0, stderr);
return mparse_process_and_free_message(&message, 0);
}

```

RETURN VALUE

mparse_parse returns zero unless a serious parsing error occurred. Examine **ioerr** in the **mparse_message** structure to determine whether an I/O error such as a timeout was encountered while processing the message.

BUGS and CAVEATS

There are some conflicts between RFCs 821, 822, 2821, 2822, 1034, 1123, 1036, 2045, 2425, etc. and quite a few gray areas. These may be addressed as the standards are revised.

RFC 2822 is rather generous in allowing certain constructs. This parser warns about invalid old dates, invalid time zones, domain names and domain literals that are syntactically legal per RFC 2822, but which are meaningless. These warnings are associated with an RFC number of zero. Printing such warnings via the demonstration program is enabled via the **-0** (zero) command-line option, which sets RFC **mode 0**.

RFC 2822 eliminated the distinction between RFC 822 extension fields and user-defined (those beginning

with "X-") fields. The RFC 822 scheme reduced the likelihood of namespace clashes. Mparse does distinguish between the two types, although they are treated similarly (some differences are mandated by RFC 2047). Separate user hooks are provided for the two types.

RFC 2184 (obsoleted by RFC 2231) used the number 1 for the first part of a parameter continuation. RFC 2231 uses 0. RFC 2184 (in effect from August to November 1997) parameter continuation numbering is not supported. This is a conflict between the two RFCs which cannot be readily resolved.

RFC 2821 permits a mix of angle bracketed addresses and mailbox specifications in a Received field *for* component, as well as permitting multiple mailboxes there. RFC 2822 does not permit a mix or multiple addr-specs. The mix and multiple mailboxes permitted by the 2821 specification lead to parsing conflicts and are therefore not supported.

The rules in RFC 2046 for message/partial reassembly require that an original message's Date field be placed in the initial piece message header when fragmenting in order to preserve that field through the fragmentation/reassembly process. That conflicts with RFC 2822 section 3.6.1 regarding the Date field semantics. **Mparse** preserves the Date information from the original message in the initial piece message header when generating that piece.

RFCs 850 and 1036 have some sections which effectively impose structure on the (RFC 822) unstructured Subject field. That contradicts RFC 822, and as both RFCs explicitly state that RFC 822 has precedence in the case of conflicts, the purported structural suggestions are not supported.

A number of extension fields have been defined in several RFCs. The following are not yet fully supported, or are supported only to the extent of non-conflicting specifications.

field	description	resolution
Received (RFCs 821, 822, 886, 2821, 2822)	RFC 886 advocates using unregistered values in the 'with' component, contrary to the other RFCs and RFC 1958 which require an IANA-registered protocol.	Unregistered values are considered errors.
Delivery-Report-Content-Reported-Recipient-Info (RFC 987)	Use of *text in the case of drc-failure leads to ambiguities that cannot be resolved without extensive lookahead.	A phrase_list is recognized where RFC 987 specifies *text.
Content-Location (RFC 2557, 2616, 2912), also Content-Base (RFC 2110)	These can't be unambiguously parsed as currently defined in the RFCs. Also, encoding of URIs and handling of long URIs are not appropriately defined. No provision for #fragments.	Comments disallowed (parentheses are part of URI). URI encoding should be used if required. Long URIs assembled from portions separated by FWS. #fragments parsed per RFC 2396.
Reporting-UA (RFCs 2298, 3798)	Specification is ambiguous.	<i>ua-name</i> parsed as RFC 2822 phrase not *text.
Received-content-MIC (RFC 3335)	Syntax uses MIME parameter value specification outside of a parameter. Permissible CFWS locations not specified. Min/max number of fields not specified.	Parsed as specified, but unlikely to be what the authors intended as the value might include RFC 2231 charset, language, and/or encoding. Liberal acceptance of CFWS. No min/max count enforced.

A number of additional MIME types have been defined in multiple RFCs. At least the following may have parsing implications which are not fully supported, or are supported only to the extent of non-conflicting specifications:

media type	description	resolution
application/dicom (RFC 3240)	the required id parameter may be optional according to RFC 3240	parameters are not checked
multipart/voice-message (RFCs 2421, 2423)	Requires text/directory be present. Requires certain content in text/directory. This poses a couple of problems for mparse : the text/directory, being contained within the multipart/voice-message wrapper, will not have been detected when the multipart/voice-message Content-Type field is seen, and mparse does not attempt to interpret text body content as that is a display issue.	Content is not checked. Application code requiring conformance with the relevant RFCs should provide checks. N.B. RFC 3801 (obsoletes 2421) has dropped the text/directory requirements.
text/directory (RFCs 2425, 2927)	Requires charset parameter. Default encoding 8bit (illegal per 2045 [6.1]; encoding always defaults to 7bit in the absence of a Content-Transfer-Encoding field).	charset parameter is checked for presence. RFC 2045 encoding rules are used. Content is uninterpreted.

Many field definitions have since been obsoleted (e.g. due to name changes from RFC 1327 to 2156). **Mparse** recognizes the obsoleted names as equivalent to the current names.

It should also be noted that undefined fields (*i.e.* those not defined in a stable, public document like an RFC) cannot be checked for syntax since there is no official syntax definition. They are treated as extension fields which may contain arbitrary text. Applications which are intended to support user-defined or undefined fields are responsible for syntax checks, field count checks, and recognizing the field from among other user-defined or extension fields.

SEE ALSO

The file **generating** for guidelines on generating RFC-compliant messages.

AUTHOR

Bruce Lilly <blilly@erols.com>