Mailfromd mail filter

version 9.2, 31 October 2026

Sergey Poznyakoff.

Copyright © 2005–2026 Sergey Poznyakoff

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Dedico aquest treball a Lluis Llach, per obrir els nous horitzons.

Short Contents

Preface 1
1 Introduction to `mailfromd` 5
2 Building the Package 9
3 Tutorial 13
4 Mail Filtering Language 53
5 The MFL Library Functions 125
6 Using the GNU Emacs MFL Mode 215
7 Configuring `mailfromd` 219
8 `Mailfromd` Command Line Syntax 233
9 Using `mailfromd` with Various MTAs 243
10 `calloutd` 249
11 `mfdbtool` 259
12 `mtasim` — a testing tool 261
13 Pmilter multiplexer program. 271
14 How to Report a Bug 279
A Gacopyz 281
B Time and Date Formats 283
C Upgrading 287
D GNU Free Documentation License 303
Concept Index 311

Table of Contents

Preface **1**
Short history of `mailfromd` 1
Acknowledgments 3

1 Introduction to `mailfromd` **5**
1.1 Typographical conventions 5
1.2 Overview of Mailfromd 6
1.3 Sender Address Verification 6
1.3.1 Limitations of Sender Address Verification 7
1.4 Controlling Mail Sending Rate 8
1.5 SPF, DKIM, and others 8

2 Building the Package **9**

3 Tutorial **13**
3.1 Start Up 13
3.2 Simplest Configurations 15
3.3 Conditional Execution 16
3.4 Functions and Modules 17
3.5 Domain Name System 18
3.6 Checking Sender Address 18
3.7 SMTP Timeouts 19
3.8 Avoiding Verification Loops 20
3.9 Using Variables 22
3.10 SMTP RSET and Milter Abort Handling 22
3.11 Controlling Number of Recipients 23
3.12 Sending Rate 24
3.13 Greylisting 27
3.14 Local Account Verification 29
3.15 Databases 30
3.15.1 Database Formats 31
3.15.2 Basic Database Operations 32
3.15.3 Database Maintenance 33
3.16 Testing Filter Scripts 33
3.17 Run Mode 35
3.17.1 The Top of a Script File 36
3.17.2 Parsing Command Line Arguments 37
3.18 Examining Default Values 40
3.19 Logging and Debugging 43
3.20 Runtime Errors 48
3.21 Notes and Cautions 51

4 Mail Filtering Language 53
4.1 Comments 53
4.2 #include and #include_once 53
4.3 #line 54
4.4 #warning and #error 54
4.5 Pragmatic comments 55
4.5.1 Pragma prereq 55
4.5.2 Pragma stacksize 55
4.5.3 Pragma regex 56
4.5.4 Pragma dbprop 57
4.5.5 Pragma greylist 57
4.5.6 Pragma miltermacros 58
4.5.7 Pragma provide-callout 58
4.6 Data Types 58
4.7 Numbers 58
4.8 Literals 59
4.9 Here Documents 60
4.10 Sendmail Macros 61
4.11 Constants 62
4.11.1 Built-in constants 63
4.12 Variables 65
4.12.1 Predefined Variables 66
4.13 Back references 69
4.14 Handlers 70
4.14.1 Multiple Handler Definitions 75
4.15 Initialization and Cleanup Handlers 75
4.15.1 The '`begin`' and '`end`' special handlers 76
4.15.2 Global startup and shutdown handlers. 77
4.15.3 Action Hook 77
4.16 Functions 78
4.16.1 Some Useful Functions 83
4.17 Expressions 85
4.17.1 Constant Expressions 85
4.17.2 Function Calls 85
4.17.3 Concatenation 85
4.17.4 Arithmetic Operations 85
4.17.5 Bitwise shifts 86
4.17.6 Relational Expressions 86
4.17.7 Special Comparisons 86
4.17.8 Boolean Expressions 87
4.17.9 Operator Precedence 87
4.17.10 Type Casting 88
4.18 Variable and Constant Shadowing 89
4.19 Statements 92
4.19.1 Action Statements 92
4.19.2 Variable Assignments 94

4.19.3 The `pass` statement ... 94
4.19.4 The `echo` statement ... 95
4.20 Conditional Statements ... 95
4.21 Loop Statements ... 96
4.22 Exceptional Conditions ... 99
4.22.1 Built-in Exceptions ... 99
4.22.2 User-defined Exceptions ... 101
4.22.3 Exception Handling ... 101
4.23 Sender Verification Tests ... 104
4.24 Modules ... 108
4.24.1 Declaring Modules ... 108
4.24.2 Scope of Visibility ... 109
4.24.3 Require and Import ... 109
4.25 Dynamically Loaded Modules ... 111
4.25.1 Loadable Library ... 111
4.25.2 Interface Module ... 114
4.25.3 Creating a Mfmod Structure ... 115
4.25.3.1 mfmodnew invocation ... 116
4.26 MFL Preprocessor ... 116
4.26.1 Preprocessor Configuration ... 117
4.26.2 Preprocessor Usage ... 117
4.26.3 Preprocessor Macros ... 118
4.27 Example of a Filter Script File ... 120
4.28 Reserved Words ... 122

5 The MFL Library Functions ... 125

5.1 Sendmail Macro Access Functions ... 125
5.2 `tr`, `dc`, and `sq` functions ... 126
5.3 The `sed` function ... 128
5.4 String Manipulation Functions ... 130
5.5 String formatting ... 133
5.6 Character Type ... 136
5.7 I/O functions ... 137
5.8 Filtering functions ... 142
5.8.1 Filters and Filter Pipes ... 143
5.9 Email processing functions ... 147
5.10 Envelope Modification Functions ... 147
5.11 Header Modification Functions ... 148
5.12 Body Modification Functions ... 150
5.13 Message Modification Queue ... 151
5.14 Mail Header Functions ... 152
5.15 Mail Body Functions ... 153
5.16 EOM Functions ... 153
5.17 Current Message Functions ... 154
5.18 Mailbox Functions ... 154
5.19 Message Functions ... 156

5.19.1 Header functions ... 157
5.19.2 Message body functions ... 157
5.19.3 MIME functions ... 158
5.19.4 Message digest functions ... 160
5.20 Quarantine Functions ... 161
5.21 SMTP Callout Functions ... 161
5.22 Compatibility Callout Functions ... 162
5.23 Internet address manipulation functions ... 164
5.24 DNS Functions ... 166
5.24.1 dns_query ... 166
5.24.2 Simplified DNS functions ... 168
5.25 Geolocation functions ... 173
5.26 Database Functions ... 175
5.27 Control Database ... 179
5.28 System functions ... 180
5.29 System User Database ... 182
5.30 Sieve Interface ... 183
5.31 Interfaces to Third-Party Programs ... 185
5.31.1 SpamAssassin ... 186
5.31.2 ClamAV ... 188
5.32 Rate limiting functions ... 189
5.33 Greylisting functions ... 190
5.34 Special Test Functions ... 190
5.35 Mail Sending Functions ... 191
5.36 Blacklisting Functions ... 194
5.37 SPF Functions ... 195
5.38 DKIM ... 199
5.38.1 Setting up a DKIM record ... 205
5.39 Sockmap Functions ... 206
5.40 National Language Support Functions ... 207
5.41 Syslog Interface ... 209
5.42 Debugging Functions ... 209
5.43 Informative Functions ... 212
5.44 Mfmod Interface Functions ... 213

6 Using the GNU Emacs MFL Mode ... 215

7 Configuring `mailfromd` ... 219

7.1 Special Configuration Data Types ... 220
7.2 Base Mailfromd Configuration ... 220
7.3 Preprocessor Configuration ... 221
7.4 DNS Resolver Configuration ... 222
7.5 Server Configuration ... 223
7.6 Milter Connection Configuration ... 225
7.7 Logging and Debugging configuration ... 225
7.8 Timeout Configuration ... 226

7.9 Call-out Configuration 227
7.10 Privilege Configuration 229
7.11 Database Configuration 229
7.12 Runtime Constants Configuration 231
7.13 Standard Mailutils Statements 231

8 `Mailfromd` Command Line Syntax 233

8.1 Command Line Options. 233
8.1.1 Operation Modifiers 233
8.1.2 General Settings 234
8.1.3 Preprocessor Options 236
8.1.4 Timeout Control 236
8.1.5 Logging and Debugging Options 236
8.1.6 Informational Options 239
8.2 Starting and Stopping 239

9 Using `mailfromd` with Various MTAs 243

9.1 Using `mailfromd` with Sendmail. 243
9.2 Using `mailfromd` with MeTA1. 244
9.3 Using `mailfromd` with Postfix 246

10 `calloutd` 249

10.1 Calloutd Configuration 249
10.1.1 `calloutd` General Setup 250
10.1.2 The `server` statement 250
10.1.3 `calloutd` logging 251
10.2 Calloutd Command-Line Options 252
10.3 The Callout Protocol 254

11 `mfdbtool` 259

11.1 Invoking `mfdbtool` 259
11.2 Configuring `mfdbtool` 260

12 `mtasim` — a testing tool 261

12.1 `mtasim` interactive mode mode 261
12.2 `mtasim` expect commands 265
12.3 Trace Files 266
12.4 Daemon Mode 266
12.5 Summary of the `mtasim` Administrative Commands 266
12.6 `mtasim` command line options 268

13 Pmilter multiplexer program. 271
13.1 Pmult Configuration 271
13.1.1 Multiplexer Configuration. 272
13.1.2 Translating MeTA1 macros. 272
13.1.3 Pmult Client Configuration. 275
13.1.4 Debugging Pmult 276
13.2 Pmult Example 277
13.3 Pmult Invocation 277

14 How to Report a Bug 279

Appendix A Gacopyz 281

Appendix B Time and Date Formats 283

Appendix C Upgrading 287
C.1 Upgrading from 9.1 to 9.2 287
C.2 Upgrading from 8.17 to 9.0 287
C.3 Upgrading from 8.16 to 8.17 288
C.4 Upgrading from 8.14 to 8.15 288
C.5 Upgrading from 8.13 to 8.14 290
C.6 Upgrading from 8.7 to 8.8 291
C.7 Upgrading from 8.5 to 8.6 291
C.8 Upgrading from 8.2 to 8.3 (or 8.4) 291
C.9 Upgrading from 7.0 to 8.0 292
C.10 Upgrading from 6.0 to 7.0 293
C.11 Upgrading from 5.x to 6.0 294
C.12 Upgrading from 5.0 to 5.1 295
C.13 Upgrading from 4.4 to 5.0 296
C.14 Upgrading from 4.3.x to 4.4 297
C.15 Upgrading from 4.2 to 4.3.x 297
C.16 Upgrading from 4.1 to 4.2 298
C.17 Upgrading from 4.0 to 4.1 298
C.18 Upgrading from 3.1.x to 4.0 298
C.19 Upgrading from 3.0.x to 3.1 299
C.20 Upgrading from 2.x to 3.0.x 300
C.21 Upgrading from 1.x to 2.x 300

Appendix D GNU Free Documentation License 303
D.1 ADDENDUM: How to use this License for your documents 309

Concept Index 311

Preface

Simple Mail Transfer Protocol (SMTP) which is the standard for email transmissions across the Internet was designed in the good old days when nobody could even think of the possibility of e-mail being abused to send tons of unsolicited messages of dubious contents. Therefore it lacks mechanisms that could have prevented this abuse (*spamming*), or at least could have made it difficult. Attempts to introduce such mechanisms (such as SMTP-AUTH extension (`http://tools.ietf.org/html/rfc2554`)) are being made, but they are not in wide use yet and, probably, their introduction will not be enough to stop the e-mail abuse. Spamming is today's grim reality and developers spend lots of time and efforts designing new protection measures against it. `Mailfromd` is one of such attempts.

The package is designed to work with any MTA supporting '`Milter`' or '`Pmilter`' protocol, such as '`Sendmail`', '`MeTA1`' or '`Postfix`'. It allows you to:

- Control whether messages come from trustworthy senders, using so called *callout* or *Sender Address Verification* (see Section 1.3 [SAV], page 6) mechanism.
- Prevent emails coming from forged addresses by use of SPF mechanism (see Section 5.37 [SPF Functions], page 195).
- Limit connection and/or sending rates (see Section 1.4 [Rate Limit], page 8).
- Use *black-*, *white-* and *greylisting* techniques.
- Invoke external programs or other mail filters.

Short history of `mailfromd`.

The idea of the utility appeared in 2005, and its first version appeared soon afterward. Back then it was a simple implementation of Sender Address Verification (see Section 1.3 [SAV], page 6) for '`Sendmail`' (hence its name – `mailfromd`) with rudimentary tuning possibilities.

After a short run on my mail servers, I discovered that the utility was not flexible enough. It took less than a month to implement a configuration file that allowed the user to control program and data flow during the '`envfrom`' SMTP state. The new version, 1.0, appeared in June, 2005.

Next major release, 1.2 (1.1 contained mostly bugfixes), appeared two months later, and introduced *mail sending rate* control (see Section 1.4 [Rate Limit], page 8).

The program evolved during the next year, and the version 2.0 was released in September, 2006. This version was a major change in the main idea of the program. Configuration file become a flexible filter script allowing the operator to control almost all SMTP states. The program supplied in the script file was compiled into a pseudo-code at startup, this code being subsequently evaluated each time the filter was invoked. This caused a considerable speed-up in comparison with the previous versions, where the run-time evaluator was traversing the parse tree. This version also introduced (implicitly, at the time), two separate data types for the entities declared in the script, which also played its role in the speed improvement (in the previous versions all data were considered strings). Lots of improvements were made in the filter language (see Chapter 4 [MFL], page 53) itself, such as user-defined functions, the `switch` statement, the `catch` statement for handling run-time errors, etc. The set of built-in functions extended considerably. A testsuite (using *DejaGNU*) was introduced in this version.

During this initial development period the limitations imposed by `libmilter` implementation became obvious. Finally, I felt they were stopping further development, and decided that `mailfromd` should use its own '`Milter`' implementation. This new library, `libgacopyz` was the main new feature of the 3.0 release, which was released in November, 2006. Another major feature was the `--dump-macros` option and the `macros` subcommand to `rc.mailfromd` script, that were intended to facilitate configuration on the '`Sendmail`' side.

The development of 3.*x* (more properly, 3.1.*x*) series concentrated mainly on bug-fixes, while the main development was done on the next branch.

The version 4.0 appeared on May 12, 2007. A full list of changes in this release is more than 500 lines long, so it is impractical to list them here. In particular, this version introduced lots of new features in MFL syntax and the library of useful MFL functions. The runtime engine was also improved, in particular, stack space become expandable which eliminated many run-time errors. This version also provided a foundation for MFL module system. The code generation was re-implemented to facilitate introduction of object files in future versions. Another new features in this release include SPF support and `mtasim` utility — an MTA simulator designed for testing `mailfromd` scripts (see Chapter 12 [mtasim], page 261). The test suite in this version was made portable by rewriting it in *Autotest*.

Another big leap forward was the 5.0 release, which appeared on December 26, 2008. It largely enriched a set of available functions (61 new functions were introduced, which amounts to 41% of all the available functions in 5.0 release) and introduced several improvements in the MFL itself. Among others, function aliases and optional arguments in user-defined functions were introduced in this release. The new "run operation mode" allowed to execute arbitrary MFL functions from the command line. This release also raised the Mailutils version requirements to at least 2.0.

Version 6.0, which was released in on 12 December, 2009, introduced a full-fledged modular system, akin to that of Python, and quite a few improvements to the language. such as explicit type casts, concatenation operator, static variables, etc.

Starting from version 7.0, the focus of further development of `mailfromd` has shifted. While previously it had been regarded as a mail-filtering server, since then it was developed as a system for extending MTA functionality in the broad sense, mail filtering being only one of features it provides.

Version 7.0 makes the MFL syntax more consistent and the language itself more powerful. For example, it is no longer necessary to use prefixes before variables to dereference them. The new '`try--catch`' construct allows for elegant handling of exceptions and errors. User-defined exceptions provide a way for programming complex loops and recursions with non-local exits.

This version introduces a concept of dedicated callout server. This allows `mailfromd` to defer verifications for a later time if the remote server does not response within a reasonably short period of time (see Section 3.7 [SMTP Timeouts], page 19).

Six years later the version 8.0 was released. This version was a major rewrite of the mailfromd codebase. It introduced a separate callout daemon that made it possible to separate the mailfromd server machine from machines performing callout checks. The MFL language was extended by a number of built-in functions.

Since version 8.3 (2017-11-02) `mailfromd` uses '`adns`'[1] for DNS queries.

The version 8.7 released in July, 2020 introduced DKIM support.

The version 8.15 (2022-12-11) introduced dynamically loaded MFL modules. These modules use dynamically loaded libraries to extend the program functionality without having to modify its code. Several such modules were provided as separate projects. See Section 4.25 [mfmod], page 111.

The version 9.0 released in January, 2024 introduced full IPv6 support, separated module and search paths and added several new MFL functions.

Acknowledgments

Many people need to be thanked for their assistance in developing and debugging `mailfromd`. After S. C. Johnson, I can say that this program "*owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right.*"

A real test for a program like `mailfromd` cannot be done but in conditions of production environment. A decision to try it in these conditions is by no means an easy one, it requires courage and good faith in the intentions and abilities of the author. To begin with, I would like to thank my contributors for these virtues.

Jan Rafaj has intrepidly been using `mailfromd` since its early releases and invested lots of efforts in improving the program and its documentation. He is the author of many of the MFL library functions, shipped with the package. Some of his ideas are still waiting in my implementation queue, while new ones are consistently arriving.

Peter Markeloff patiently tested every `mailfromd` release and helped discover and fix many bugs.

Zeus Panchenko contributed many ideas and gave lots of helpful comments. He offered invaluable help in debugging and testing `mailfromd` on FreeBSD platform.

Sergey Afonin proposed many improvements and new ideas. He also invested a lot of his time in finding bugs and testing bugfixes.

John McEleney and Ben McKeegan contributed the token bucket filter implementation (see [TBF], page 25).

Con Tassios helped to find and fix various bugs and contributed the new implementation of the `greylist` function (see [greylisting types], page 28).

The following people (in alphabetical order) provided bug reports and helpful comments for various versions of the program: Alan Dobkin, Brent Spencer, Jeff Ballard, Nacho González López, Phil Miller, Simon Christian, Thomas Lynch.

[1] `https://www.gnu.org/software/adns`

1 Introduction to mailfromd

Mailfromd is a general-purpose mail filtering daemon and a suite of accompanying utilities for Sendmail[1], MeTA1[2], Postfix[3] or any other MTA that supports Milter (or Pmilter) protocol. It is able to filter both incoming and outgoing messages using a filter program, written in *mail filtering language* (MFL). The daemon interfaces with the MTA using Milter protocol.

The name mailfromd can be thought of as an abbreviation for '*Mail F*iltering and *R*untime *M*odification' *D*aemon, with an 'o' for itself. Historically, it stemmed from the fact that the original implementation was a simple filter implementing the *sender address verification* technique. Since then the program has changed dramatically, and now it is actually a language translator and run-time evaluator providing a set of built-in and library functions for filtering electronic mail.

The first part of this manual is an overview, describing the features mailfromd offers in general.

The second part is a tutorial, which provides an introduction for those who have not used mailfromd previously. It moves from topic to topic in a logical, progressive order, building on information already explained. It offers only the principal information needed to master basic practical usage of mailfromd, while omitting many subtleties.

The other parts are meant to be used as a reference for those who know mailfromd well enough, but need to look up some notions from time to time. Each chapter presents everything that needs to be said about a specific topic.

The manual assumes that the reader has a good knowledge of the SMTP protocol and the mail transport system he uses (Sendmail , Postfix or MeTA1).

1.1 Typographical conventions

This manual is written using Texinfo, the GNU documentation formatting language. The same set of Texinfo source files is used to produce both the printed and online versions of the documentation. Because of this, the typographical conventions may be slightly different than in other books you may have read.

Examples you would type at the command line are preceded by the common shell primary prompt, '$'. The command itself is printed *in this font*, and the output it produces 'in this font', for example:

```
$ mailfromd --version
mailfromd (mailfromd 9.2)
```

In the text, the command names are printed like this, command line options are displayed in this font. Some notions are emphasized *like this*, and if a point needs to be made strongly, it is done **this way**. The first occurrence of a new term is usually its *definition* and appears in the same font as the previous occurrence of "definition" in this sentence. File names are indicated like this: /path/to/ourfile.

[1] See http://www.sendmail.org

[2] See http://www.meta1.org

[3] See http://www.postfix.org

The variable names are represented *like this*, keywords and fragments of program text are written in `this font`.

1.2 Overview of Mailfromd

In contrast to the most existing milter filters, `mailfromd` does not implement any default filtering policies. Instead, it depends entirely on a *filter script*, supplied to it by the administrator. The script, written in a specialized and simple to use language, called MFL (see Chapter 4 [MFL], page 53), is supposed to run a set of tests and to decide whether the message should be accepted by the MTA or not. To perform the tests, the script can examine the values of `Sendmail` macros, use an extensive set of built-in and library functions, and invoke user-defined functions.

1.3 Sender Address Verification.

Sender address verification, or *callout*, is one of the basic mail verification techniques, implemented by `mailfromd`. It consists in probing each MX server for the given address, until one of them gives a definite (positive or negative) reply. Using this technique you can block a sender address if it is not deliverable, thereby cutting off a large amount of spam. It can also be useful to block mail for undeliverable recipients, for example on a mail relay host that does not have a list of all the valid recipient addresses. This prevents undeliverable junk mail from entering the queue, so that your MTA doesn't have to waste resources trying to send '`MAILER-DAEMON`' messages back.

Let's illustrate how it works on an example:

Suppose that the user '`<jsmith@somedomain.net>`' is trying to send mail to one of your local users. The remote machine connects to your MTA and issues `MAIL FROM: <jsmith@somedomain.net>` command. However, your MTA does not have to take its word for it, so it uses `mailfromd` to verify the sender address validity. `Mailfromd` strips the domain name from the address ('`somedomain.net`') and queries DNS about '`MX`' records for that domain. Suppose, it receives the following list

```
10          relay1.somedomain.net
20          relay2.somedomain.net
```

It then connects to first MX server, using SMTP protocol, as if it were going to send a message to '`<jsmith@somedomain.net>`'. This is called sending a *probe message*. If the server accepts the recipient address, the `mailfromd` accepts the incoming mail. Otherwise, if the server rejects the address, the mail is rejected as well. If the MX server cannot be connected, `mailfromd` selects next server from the list and continues this process until it finds the answer or the list of servers is exhausted.

The *probe message* is like a normal mail except that no data are ever being sent. The probe message transaction in our example might look as follows ('`S:`' meaning messages sent by remote MTA, '`C:`' meaning those sent by `mailfromd`):

```
C: HELO mydomain.net
S: 220 OK, nice to meet you
C: MAIL FROM: <>
S: 220 <>: Sender OK
C: RCPT TO: <jsmith@somedomain.net>
```

```
S: 220 <jsmith@remote.net>: Recipient OK
C: QUIT
```

Probe messages are never delivered, deferred or bounced; they are always discarded.

The described method of address verification is called a *standard* method throughout this document. `Mailfromd` also implements a method we call *strict*. When using strict method, `mailfromd` first resolves IP address of sender machine to a fully qualified domain name. Then it obtains '`MX`' records for this machine, and then proceeds with probing as described above.

So, the difference between the two methods is in the set of '`MX`' records that are being probed: standard method queries '`MX`'s based on the sender email domain, strict method works with '`MX`'s for the sender IP address.

Strict method allows to cut off much larger amount of spam, although it does have many drawbacks. Returning to our example above, consider the following situation: '`<jsmith@somedomain.net>`' is a perfectly normal address, but it is being used by a spammer from some other domain, say '`otherdomain.com`'. The standard method is not able to cope with such cases, whereas the strict one is.

An alert reader will ask: what happens if `mailfromd` is not able to get a definite answer from any of MX servers? Actually, it depends entirely on how you will instruct it to act in this case, but the general practice is to return temporary failure, which will urge the remote party to retry sending their message later.

After receiving a definite answer, `mailfromd` will cache it in its database, so that next time your MTA receives a message from that address (or from the sender IP/email address pair, for strict method), it will not waste its time trying to reach MX servers again. The records remain in the cache database for a certain time, after which they are discarded.

1.3.1 Limitations of Sender Address Verification

Before deciding whether and how to use sender address verification, you should be aware of its limitations.

Both standard and strict methods suffer from the following limitations:

- The sender verification methods will perform poorly on highly loaded sites. The traffic and/or resource usage overhead may not be feasible for you. However, you may experiment with various `mailfromd` options to find an optimal configuration.
- Some sites may blacklist your MTA if it probes them too often. `Mailfromd` eliminates this drawback by using a *cache database*, which keeps results of the recent callouts.
- When verifying the remote address, no attempt to actually deliver the message is made. If MTA accepts the address, `mailfromd` assumes it is OK. However in reality, a mail for a remote address can bounce *after* the nearest MTA accepts the recipient address.

  This drawback can often be avoided by combining sender address verification with greylisting (see Section 3.13 [Greylisting], page 27).
- If the remote server rejects the address, no attempt is being made to discern between various reasons for rejection (client rejected, '`HELO rejected`', '`MAIL FROM` rejected, etc.)
- Some major sites such as '`yahoo.com`' do not reject unknown addresses in reply to the '`RCPT TO`' command, but report a delivery failure in response to end of '`DATA`' after a

message is transferred. Of course, sender address verification does not work with such sites. However, a combination of address verification and greylisting (see Section 3.13 [Greylisting], page 27) may be a good choice in such cases.

In addition, strict verification breaks forward mail delivery. This is obvious, since mail forwarding is based on delivering unmodified message to another location, so the sender address domain will most probably not be the same as that of the MTA doing the forwarding.

1.4 Controlling Mail Sending Rate.

Mail Sending Rate for a given identity is defined as the number of messages with this identity received within a predefined interval of time.

MFL offers a set of functions for limiting mail sending rate (see Section 5.32 [Rate limiting functions], page 189), and for controlling broader rate aspects, such as data transfer rates (see [TBF], page 25).

1.5 SPF, DKIM, and others

Sender Policy Framework, or SPF for short, is an extension to SMTP protocol that allows to identify forged identities supplied with the `MAIL FROM` and `HELO` commands. The framework is explained in detail in RFC 4408 (`http://tools.ietf.org/html/rfc4408`) and on the SPF Project Site (`http://www.openspf.org/`).

Mailfromd provides a set of functions for using SPF to control mail flow. These are described in Section 5.37 [SPF Functions], page 195.

DomainKeys Identified Mail (DKIM) is an email authentication method designed to detect forged sender addresses in emails. Mailfromd supports both DKIM signing and verification. See Section 5.38 [DKIM], page 199, for a detailed description of these features.

Mailfromd also provides support for some third-party spam-abatement programs, namely `SpamAssassin`, and `ClamAV`. These are discussed in Section 5.31 [Interfaces to Third-Party Programs], page 185.

2 Building the Package

This chapter contains a detailed list of steps you need to undertake in order to configure and build the package.

1. Make sure you have the necessary software installed.

   To build `mailfromd` you will need to have following packages on your machine:

   A. GNU mailutils version 3.3 or newer.

      GNU mailutils is a general-purpose library for handling electronic mail. It is available from `http://mailutils.org`.

   B. GNU adns library, version 1.5.1 or newer.

      GNU adns is an advanced DNS client library. The recent version can be downloaded from `http://www.chiark.greenend.org.uk/~ian/adns/adns.tar.gz`. Visit `http://www.gnu.org/software/adns`, for more information.

   C. A DBM library. `Mailfromd` is able to link with any flavor of DBM supported by GNU mailutils. As of version 9.2 it will refuse to build without DBM. By default, `configure` will try to find the best implementation installed on your machine (preference is given to Berkeley DB) and will use it. You can, however, explicitly specify which implementation you want to use. To do so, use the `--with-dbm` configure option. Its argument specifies the *type* of database to use. It must be one of the types supported by GNU mailutils. At the time of this writing, these are:

bdb	Berkeley DB (versions 2 to 6).
gdbm	GNU DBM.
kc	Kyoto Cabinet
tc	Tokyo Cabinet
ndbm	NDBM

      To check what database types are supported by your version of mailutils, run the following command:

      ```
      $ mailutils dbd gdbm kc tc ndbm
      ```

      For backward compatibility, `configure` accepts the following two options:

      `--with-gdbm`
      : Same as `--with-dbm=gdbm`.

      `--with-berkeley-db`
      : Same as `--with-dbm=bdb`.

      For `Sendmail` users, it often makes sense to configure `mailfromd` to use the same database flavor as `sendmail`. The following table will help you do that. The column '`DB type`' lists types of DBM databases supported by `mailfromd`. The column '`confMAPDEF`' lists the value of `confMAPDEF` Sendmail configuration macro corresponding to that database type. The column '`configure option`' contains the corresponding option to configure.

DB type	confMAPDEF	configure option
NDBM	-NNDBM	--with-dbm=ndbm
Berkeley DB	-NNEWDB	--with-dbm=bdb
GDBM	N/A	--with-dbm=gdbm

2. Decide what user privileges will be used to run `mailfromd`

   After startup, the program drops root privileges. By default, it switches to the privileges of user '`mail`', group '`mail`'. If there is no such user on your system, or you wish to use another user account for this purpose, override it using *DEFAULT_USER* environment variable. For example for `mailfromd` to run as user '`nobody`', use

   ```
   ./configure DEFAULT_USER=nobody
   ```

   The user name can also be changed at run-time (see [–user], page 235).

3. Decide where to install `mailfromd` and where its filter script and data files will be located.

   As usual, the default value for the installation prefix is `/usr/local`. If it does not suit you, specify another location using `--prefix` option, e.g.: '`--prefix=/usr`'.

   During installation phase, the build system will install several files. These files are:

   prefix`/sbin/mailfromd`
   : Main daemon. See Chapter 8 [mailfromd], page 233.

   prefix`/etc/mailfromd.mfl`
   : Default main filter script file. It is installed only if it is not already there. Thus, if you are upgrading to a newer version of `mailfromd`, your old script file will be preserved with all your changes.

     See Chapter 4 [MFL], page 53, for a description of the mail filtering language.

   prefix`/share/mailfromd/9.2/*.mfl`
   : MFL modules. See Section 4.24 [Modules], page 108.

   prefix`/share/mailfromd/9.2/include/pp-setup`
   : Default preprocessor setup file. See Section 4.26 [Preprocessor], page 116.

   prefix`/info/mailfromd.info*`
   : Documentation files.

   prefix`/bin/mtasim`
   : MTA simulator program for testing `mailfromd` scripts. See Chapter 12 [mtasim], page 261.

   prefix`/sbin/pmult`
   : Pmilter multiplexor for `MeTA1`. See Chapter 13 [pmult], page 271. It is build only if `MeTA1` version '`PreAlpha29.0`' or newer is installed on the system. You may disable it by using the `--disable-pmilter` command line option.

     When testing for `MeTA1` presence, `configure` assumes its default location. If it is not found there, inform `configure` about its actual location by using the following option:

     ```
     --enable-pmilter=prefix
     ```

where *prefix* stands for the `MeTA1` installation prefix.

It is advisable to use the same settings for file name prefixes as those you used when configuring `mailutils`. In particular, try to use the same `--sysconfdir`, since it will facilitate configuring the whole system.

Another important point is location of *local state directory*, i.e. a directory where `mailfromd` keeps its data files (e.g. communication socket, PID-file and database files). By default, its full name is *`localstatedir`*`/mailfromd`. You can change it by setting `DEFAULT_STATE_DIR` configuration variable. This value can be changed at runtime using the `state-directory` configuration statement (see Section 7.2 [conf-base], page 220). To inspect the actual value of the local state directory, run

```
mailfromd --no-config --show-defaults | grep '^statedir:'
```

See Section 3.18 [Examining Defaults], page 40, for information about the `--show-defaults` option.

4. Select default communication socket. This is the socket used to communicate with MTA, in the usual `Milter` port notation (see [milter port specification], page 220). If the socket name does not begin with a protocol or directory separator, it is assumed to be a UNIX socket, located in the local state directory. The default value is `mailfrom`, which is equivalent to `unix:`*`localstatedir`*`/mailfromd/mailfrom`.

   To alter this, use `DEFAULT_SOCKET` environment variable, e.g.:

   ```
   ./configure DEFAULT_SOCKET=inet:999@localhost
   ```

   The communication socket can be changed at run time using `--port` command line option (see [–port], page 235) or the `listen` configuration statement (see Section 7.5 [conf-server], page 223).

5. Select default expiration interval. *Expiration interval* defines the period of time during which a record in the `mailfromd` database is considered valid. It is described in more detail in Section 3.15 [Databases], page 30. The default value is 86400 seconds, i.e. 24 hours. It is OK for most sites. If, however, you wish to change it, use *DEFAULT_EXPIRE_INTERVAL* environment variable.

   The `DEFAULT_EXPIRE_RATES_INTERVAL` variable sets default expiration time for mail rate database (see Section 5.32 [Rate limiting functions], page 189).

   Expiration settings can be changed at run time using `database` statement in the `mailfromd` configuration file (see Section 7.11 [conf-database], page 229).

6. Select a `syslog` implementation to use.

   `Mailfromd` uses `syslog` for diagnostics output. The default `syslog` implementation on most systems (most notably, on GNU/Linux) uses blocking `AF_UNIX SOCK_DGRAM` sockets. As a result, when an application calls `syslog()`, and `syslogd` is not responding and the socket buffers get full, the application will hang.

   For `mailfromd`, as for any daemon, it is more important that it continue to run, than that it continue to log. For this purpose, `mailfromd` is shipped with a non-blocking `syslog` implementation by Simon Kelley. This implementation, instead of blocking, buffers log lines in memory. When the buffer log overflows, some lines are lost, but the daemon continues to run. When lines are lost, this fact is logged with a message of the form:

   ```
   async_syslog overflow: 5 log entries lost
   ```

To enable this implementation, configure the package with `--enable-syslog-async` option, e.g.:

```
./configure --enable-syslog-async
```

Additionally, you can instruct `mailfromd` to use asynchronous syslog by default. To do so, set `DEFAULT_SYSLOG_ASYNC` to 1, as shown in example below:

```
./configure --enable-syslog-async DEFAULT_SYSLOG_ASYNC=1
```

You will be able to override these defaults at run-time by using the `--logger` command line option (see Section 3.19 [Logging and Debugging], page 43).

7. Run `configure` with all the desired options.

   For example, the following command:

   ```
   ./configure DEFAULT_SOCKET=inet:999@localhost --with-berkeley-db=3
   ```

   will configure the package to use Berkeley DB database, version 2, and '`inet:999@localhost`' as the default communication socket.

   At the end of its run `configure` will print a concise summary of its configuration settings. It looks like that (with the long lines being split for readability):

   ```
   ****************************************************************
   Mailfromd configured with the following settings:

   Mailutils version..................... 3.15
   External preprocessor................. /usr/bin/m4 -s
   DBM version........................... Berkeley DB v. 3
   Default user.......................... mail
   State directory.......................
             $(localstatedir)/$(PACKAGE)
   Socket................................ mailfrom
   Expiration interval................... 86400
   Compile asynchronous syslog........... no
   Readline (for mtasim)................. yes
   Documentation rendition type.......... PROOF
   Enable pmilter support................ no
   Enable GeoIP2 support................. no
   Enable DKIM support................... yes
   IPv6 support.......................... yes
   ****************************************************************
   ```

   Make sure these settings satisfy your needs. If they do not, reconfigure the package with the right options.

8. Run `make`.
9. Run `make` install.
10. Make sure *`localstatedir`*`/mailfromd` has the right owner and mode.
11. Examine filter script file (*`sysconfdir`*`/mailfromd.mfl`) and edit it, if necessary.
12. If you are upgrading from an earlier release of Mailfromd, refer to Appendix C [Upgrading], page 287, for detailed instructions.

3 Tutorial

This chapter contains a tutorial introduction, guiding you through various `mailfromd` configurations, starting from the simplest ones and proceeding up to more advanced forms. It omits most complicated details, concentrating mainly on the common practical tasks.

If you are familiar with `mailfromd`, you can skip this chapter and go directly to the next one (see Chapter 4 [MFL], page 53), which contains detailed discussion of the mail filtering language and `mailfromd` interaction with the Mail Transport Agent.

3.1 Start Up

The `mailfromd` utility runs as a standalone *daemon* program and listens on a predefined communication channel for requests from the *Mail Transfer Agent* (MTA, for short). When processing each message, the MTA installs communication with `mailfromd`, and goes through several states, collecting the necessary data from the sender. At each state it sends the relevant information to `mailfromd`, and waits for it to reply. The `mailfromd` filter receives the message data through *Sendmail macros* and runs a *handler program* defined for the given state. The result of this run is a *response code*, that it returns to the MTA. The following response codes are defined:

`continue` Continue message processing from next milter state.

`accept` Accept this message for delivery. After receiving this code the MTA continues processing this message without further consulting `mailfromd` filter.

`reject` Reject this message. The message processing stops at this stage, and the sender receives the reject reply ('`5xx`' reply code). No further `mailfromd` handlers are called for this message.

`discard` Silently discard the message. This means that MTA will continue processing this message as if it were going to deliver it, but will discard it after receiving. No further interaction with `mailfromd` occurs.

`tempfail` Temporarily reject the message. The message processing stops at this stage, and the sender receives the '`temporary failure`' reply ('`4xx`' reply code). No further `mailfromd` handlers are called for this message.

The instructions on how to process the message are supplied to `mailfromd` in its *filter script file*. It is normally called `/usr/local/etc/mailfromd.mfl` (but can be located elsewhere, see Chapter 8 [Invocation], page 233) and contains a set of *milter state handlers*, or subroutines to be executed in various SMTP states. Each interaction state can be supplied its own handling procedure. A missing procedure implies `continue` response code.

The filter script can define up to nine *milter state handlers*, called after the names of milter states: '`connect`', '`helo`', '`envfrom`', '`envrcpt`', '`data`', '`header`', '`eoh`', '`body`', and '`eom`'. The '`data`' handler is invoked only if MTA uses Milter protocol version 3 or later. Two special handlers are available for initialization and clean-up purposes: '`begin`' is called before the processing starts, and '`end`' is called after it is finished. The diagram below shows the control flow when processing an SMTP transaction. Lines marked with `C:` show SMTP commands issued by the remote machine (the *client*), those marked with '⇒' show called

handlers with their arguments. An '[R]' appearing at the start of a line indicates that this part of the transaction can be repeated any number of times:

```
⇒ begin()
⇒ connect(hostname, family, port, 'IP address')
C: HELO domain
helo(domain)
for each message transaction
do
        C: MAIL FROM sender
        ⇒ envfrom(sender)

[R]     C: RCPT TO recipient
        ⇒ envrcpt(recipient)

        C: DATA
        ⇒ data()
[R]     C: header: value
        ⇒ header(header, value)

        C:
        ⇒ eoh()

[R]     C: body-line
        ⇒ /* Collect lines into blocks blk of
        ⇒  * at most len bytes and for each
        ⇒  * such block call:
        ⇒  */
        ⇒ body(blk, len)

        C: .
        ⇒ eom()
done
⇒ end()
```

Figure 3.1: Mailfromd Control Flow

This control flow is maintained for as long as each called handler returns `continue` (see Section 4.19.1 [Actions], page 92). Otherwise, if any handler returns `accept` or `discard`, the message processing continues, but no other handler is called. In the case of `accept`, the MTA will accept the message for delivery, in the case of `discard` it will silently discard it.

If any of the handlers returns `reject` or `tempfail`, the result depends on the handler. If this code is returned by `envrcpt` handler, it causes this particular recipient address to be rejected. When returned by any other handler, it causes entire message to be rejected.

The `reject` and `tempfail` actions executed by `helo` handler do not take effect immediately. Instead, their action is deferred until the next SMTP command from the client, which is usually `MAIL FROM`.

3.2 Simplest Configurations

The `mailfromd` script file contains a series of *declarations* of the handler procedures. Each declaration has the form:

```
prog name
do
  ...
done
```

where `prog`, `do` and `done` are the *keywords*, and *name* is the state name for this handler. The dots in the above example represent the actual *code*, or a set of commands, instructing `mailfromd` how to process the message.

For example, the declaration:

```
prog envfrom
do
  accept
done
```

installs a handler for '`envfrom`' state, which always approves the message for delivery, without any further interaction with `mailfromd`.

The word `accept` in the above example is an *action*. *Action* is a special language statement that instructs the run-time engine to stop execution of the program and to return a response code to the `Sendmail`. There are five actions, one for each response code: `continue`, `accept`, `reject`, `discard`, and `tempfail`. Among these, `reject` and `discard` can optionally take one to three arguments. There are two ways of supplying the arguments.

In the first form, called *literal* or *traditional* notation, the arguments are supplied as additional words after the action name, separated by whitespace. The first argument is a three-digit RFC 2821 reply code. It must begin with '`5`' for `reject` and with '`4`' for `tempfail`. If two arguments are supplied, the second argument must be either an *extended reply code* (RFC 1893/2034) or a textual string to be returned along with the SMTP reply. Finally, if all three arguments are supplied, then the second one must be an extended reply code and the third one must supply the textual string. The following examples illustrate all possible ways of using the `reject` statement in literal notation:

```
reject
reject 503
reject 503 5.0.0
reject 503 "Need HELO command"
reject 503 5.0.0 "Need HELO command"
```

Please note the quotes around the textual string.

Another form for these action is called *functional* notation, because it resembles the function syntax. When used in this form, the action word is followed by a parenthesized group of exactly three arguments, separated by commas. The meaning and ordering of the argument is the same as in literal form. Any of three arguments may be absent, in which case it will be replaced by the default value. To illustrate this, here are the statements from the previous example, written in functional notation:

```
reject(,,)
reject(503,,)
reject(503, 5.0.0)
reject(503,, "Need HELO command")
reject(503, 5.0.0, "Need HELO command")
```

3.3 Conditional Execution

Programs consisting of a single action are rarely useful. In most cases you will want to do some checking and decide whether to process the message depending on its result. For example, if you do not want to accept messages from the address '`<badguy@some.net>`', you could write the following program:

```
prog envfrom
do
  if $f = "badguy@some.net"
    reject
  else
    accept
  fi
done
```

This example illustrates several important concepts. First or all, `$f` in the third line is a *Sendmail macro reference*. Sendmail macros are referenced the same way as in `sendmail.cf`, with the only difference that curly braces around macro names are optional, even if the name consists of several letters. The value of a macro reference is always a string.

The equality operator ('`=`') compares its left and right arguments and evaluates to true if the two strings are exactly the same, or to false otherwise. Apart from equality, you can use the regular relational operators: '`!=`', '`>`', '`>=`', '`<`' and '`<=`'. Notice that string comparison in `mailfromd` is always case sensitive. To do case-insensitive comparison, translate both operands to upper or lower case (See [tolower], page 132, and see [toupper], page 132).

The `if` statement decides what actions to execute depending on the value its condition evaluates to. Its usual form is:

```
if expression then-body [else else-body] fi
```

The *then-body* is executed if the *expression* evaluates to `true` (i.e. to any non-zero value). The optional *else-body* is executed if the *expression* yields `false` (i.e. zero). Both *then-body* and *else-body* can contain other `if` statements, their nesting depth is not limited. To facilitate writing complex conditional statements, the `elif` keyword can be used to introduce alternative conditions, for example:

```
prog envfrom
do
  if $f = "badguy@some.net"
    reject
  elif $f = "other@domain.com"
    tempfail 470 "Please try again later"
  else
    accept
  fi
done
```

See [switch], page 95, for more elaborate forms of conditional branching.

3.4 Functions and Modules

As any programming language, MFL supports a concept of *function*, i.e. a body of code that is assigned a unique name and can be invoked elsewhere as many times as needed.

All functions have a *definition* that introduces types and names of the formal parameters and the result type, if the function is to return a meaningful value (function definitions in MFL are discussed in detail in see [User-defined], page 78).

A function is invoked using a special construct, a *function call*:

name (*arg-list*)

where *name* is the function name, and *arg-list* is a comma-separated list of expressions. Each expression in *arg-list* is evaluated, and its type is compared with that of the corresponding formal argument. If the types differ, the expression is converted to the formal argument type. Finally, a copy of its value is passed to the function as a corresponding argument. The order in which the expressions are evaluated is not defined. The compiler checks that the number of elements in *arg-list* match the number of mandatory arguments for function *name*.

If the function does not deliver a result, it should only be called as a statement.

Functions may be recursive, even mutually recursive.

`Mailfromd` comes with a rich set of predefined functions for various purposes. There are two basic function classes: *built-in* functions, that are implemented by the MFL runtime environment in `mailfromd`, and *library* functions, that are implemented in MFL. The built-in functions are always available and no preparatory work is needed before calling them. In contrast, the library functions are defined in *modules*, special MFL source files that contain functions designed for a particular task. In order to access a library function, you must first *require* a module it is defined in. This is done using `require` statement. For example, the function `hostname` looks up in the DNS the name corresponding to the IP address specified as its argument. This function is defined in module `dns.mfl`, so before calling it you must require this module:

```
require dns
```

The `require` statement takes a single argument: the name of the requested module (without the '`.mfl`' suffix). It looks up the module on disk and loads it if it is available.

For more information about the module system See Section 4.24 [Modules], page 108.

3.5 Domain Name System

Site administrators often do not wish to accept mail from hosts that do not have a proper reverse delegation in the Domain Name System. In the previous section we introduced the library function `hostname`, that looks up in the DNS the name corresponding to the IP address specified as its argument. If there is no corresponding name, the function returns its argument unchanged. This can be used to test if the IP was resolved, as illustrated in the example below:

```
require 'dns'

prog envfrom
do
  if hostname($client_addr) = $client_addr
    reject
  fi
done
```

The `#require dns` statement loads the module `dns.mfl`, after which the definition of `hostname` becomes available.

A similar function, `resolve`, which resolves the symbolic name to the corresponding IP address is provided in the same `dns.mfl` module.

3.6 Checking Sender Address

A special language construct is provided for verification of sender addresses (*callout*):

```
on poll $f do
when success:
  accept
when not_found or failure:
  reject 550 5.1.0 "Sender validity not confirmed"
when temp_failure:
  tempfail 450 4.1.0 "Try again later"
done
```

The `on poll` construct runs standard verification (see [standard verification], page 6) for the email address specified as its argument (in the example above it is the value of the Sendmail macro '`$f`'). The check can result in the following conditions:

`success` The address exists.

`not_found`
: The address does not exist.

`failure` Some error of permanent nature occurred during the check. The existence of the address cannot be verified.

`temp_failure`
: Some temporary failure occurred during the check. The existence of the address cannot be verified at the moment.

The `when` branches of the `on poll` statement introduce statements, that are executed depending on the actual return condition. If any condition occurs that is not handled

within the `on` block, the run-time evaluator will signal an *exception*[1] and return temporary failure, therefore it is advisable to always handle all four conditions. In fact, the condition handling shown in the above example is preferable for most normal configurations: the mail is accepted if the sender address is proved to exist and rejected otherwise. If a temporary failure occurs, the remote party is urged to retry the transaction some time later.

The `poll` statement itself has a number of options that control the type of the verification. These are discussed in detail in [poll], page 107.

It is worth noticing that there is one special email address which is always available on any host, it is the *null address* '`<>`' used in error reporting. It is of no use verifying its existence:

```
prog envfrom
do
  if $f == ""
    accept
  else
    on poll $f do
    when success:
      accept
    when not_found or failure:
      reject 550 5.1.0 "Sender validity not confirmed"
    when temp_failure:
      tempfail 450 4.1.0 "Try again later"
    done
  fi
done
```

3.7 SMTP Timeouts

When using polling functions, it is important to take into account possible delays, which can occur in SMTP transactions. Such delays may be due to low network bandwidth or high load on the remote server. Some sites impose them willingly, as a spam-fighting measure.

Ideally the callout verification should use the timeout values defined in the RFC 2822, but this is impossible in practice, because it would cause a *timeout escalation*, which consists in propagating delays encountered in a callout SMTP session back to the remote client whose session initiated the callout.

Consider, for example, the following scenario. An MFL script performs a callout on '`envfrom`' stage. The remote server is overloaded and delays heavily in responding, so that the initial response arrives 3 minutes after establishing the connection, and processing the '`EHLO`' command takes another 3 minutes. These delays are OK according to the RFC, which imposes a 5 minute limit for each stage, but while waiting for the remote reply our SMTP server remains in the '`envfrom`' state with the client waiting for a response to its '`MAIL`' command more than 6 minutes, which is intolerable, because of the same 5 minute limit. Thus, the client will almost certainly break the session.

[1] For more information about exceptions and their handling, please refer to Section 4.22 [Exceptions], page 99.

To avoid this, `mailfromd` uses a special instance, called *callout server*, which is responsible for running callout SMTP sessions asynchronously. The usual sender verification is performed using so-called *soft* timeout values, which are set to values short enough to not disturb the incoming session (e.g. a timeout for '`HELO`' response is 3 seconds, instead of 5 minutes). If this verification yields a definite answer, that answer is stored in the cache database and returned to the calling procedure immediately. If, however, the verification is aborted due to a timeout, the caller procedure is returned an '`e_temp_failure`' exception, and the callout is scheduled for processing by a callout server. This exception normally causes the milter session to return a temporary error to the sender, urging it to retry the connection later.

In the meantime, the callout server runs the sender verification again using another set of timeouts, called *hard* timeouts, which are normally much longer than '`soft`' ones (they default to the values required by RFC 2822). If it gets a definitive result (e.g. '`email found`' or '`email not found`'), the server stores it in the cache database. If the callout ends due to a timeout, a '`not_found`' result is stored in the database.

Some time later, the remote server retries the delivery, and the `mailfromd` script is run again. This time, the callout function will immediately obtain the already cached result from the database and proceed accordingly. If the callout server has not finished the request by the time the sender retries the connection, the latter is again returned a temporary error, and the process continues until the callout is finished.

Usually, callout server is just another instance of `mailfromd` itself, which is started automatically to perform scheduled SMTP callouts. It is also possible to set up a separate callout server on another machine. This is discussed in Chapter 10 [calloutd], page 249.

For a detailed information about callout timeouts and their configuration, see Section 7.8 [conf-timeout], page 226.

For a description of how to configure `mailfromd` to use callout servers, see Section 7.5 [conf-server], page 223.

3.8 Avoiding Verification Loops

An `envfrom` program consisting only of the `on poll` statement will work smoothly for incoming mails, but will create infinite loops for outgoing mails. This is because upon sending an outgoing message `mailfromd` will start the verification procedure, which will initiate an SMTP transaction with the same mail server that runs it. This transaction will in turn trigger execution of `on poll` statement, etc. *ad infinitum*. To avoid this, any properly written filter script should not run the verification procedure on the email addresses in those domains that are relayed by the server it runs on. This can be achieved using `relayed` function. The function returns `true` if its argument is contained in one of the predefined *domain list* files. These files correspond to `Sendmail` plain text files used in `F` class definition forms (see *Sendmail Installation and Operation Guide*, chapter 5.3), i.e. they contain one domain name per line, with empty lines and lines started with '`#`' being ignored. The domain files consulted by `relayed` function are defined in the `relayed-domain-file` configuration file statement (see Section 7.2 [conf-base], page 220):

```
relayed-domain-file (/etc/mail/local-host-names,
                     /etc/mail/relay-domains);
```

or:

```
relayed-domain-file /etc/mail/local-host-names;
relayed-domain-file /etc/mail/relay-domains;
```

The above example declares two domain list files, most commonly used in `Sendmail` installations to keep hostnames of the server[2] and names of the domains, relayed by this server[3].

Given all this, we can improve our filter program:

```
require 'dns'

prog envfrom
do
  if $f == ""
    accept
  elif relayed(hostname(${client_addr}))
    accept
  else
    on poll $f do
    when success:
      accept
    when not_found or failure:
      reject 550 5.1.0 "Sender validity not confirmed"
    when temp_failure:
      tempfail 450 4.1.0 "Try again later"
    done
  fi
done
```

If you feel that your Sendmail's relayed domains are not restrictive enough for `mailfromd` filters (for example you are relaying mails from some third-party servers), you can use a database of trusted mail server addresses. If the number of such servers is small enough, a single '`or`' statement can be used, e.g.:

```
elif ${client_addr} = "10.10.10.1"
     or ${client_addr} = "192.168.11.7"
  accept
...
```

otherwise, if the servers' IP addresses fall within one or several CIDRs, you can use the `match_cidr` function (see Section 5.23 [Internet address manipulation functions], page 164), e.g.:

```
elif match_cidr (${client_addr}, "199.232.0.0/16")
  accept
...
```

or combine both methods. Finally, you can keep a DBM database of relayed addresses and use `dbmap` or `dbget` function for checking (see Section 5.26 [Database functions], page 175).

[2] class '`w`', see *Sendmail Installation and Operation Guide*, chapter 5.2.

[3] class '`R`'

```
elif dbmap("%__statedir__/relay.db", ${client_addr})
  accept
...
```

3.9 Using Variables

As any programming language, MFL has a concept of *variables*. A variable is declared using the following syntax:

```
type name
```

where *variable* is the variable name and *type* defines its type: either `string`, or `number`.

A variable is assigned a value using the `set` statement:

```
set name expr
```

where *expr* is any valid MFL expression.

The `set` statement can occur within handler or function declarations as well as outside of them.

A variable need not be declared prior to assigning a value: a `set` statement with *name* referring to a variable that has not been declared, declares it implicitly, deducing its type from that of the *expr*.

There are two kinds of `Mailfromd` variables: *global variables*, that are visible to all handlers and functions, and *automatic variables*, that are available only within the handler or function where they are declared. Any variable declared outside of a `prog` block is a global variable. Global variables are used to pass information between the handlers. For example, the following code fragment uses global variable `helohost` to store the argument to the `helo` handler and to use it later on to decide whether to accept the message.

```
# Declare the helohost variable
string helohost

prog helo
do
  # Save the host name for further use
  set helohost $1
done

prog envfrom
do
  # Reject hosts claiming to be localhost
  if helohost = "localhost"
    reject 570 "Please specify real host name"
  fi
done
```

3.10 SMTP RSET and Milter Abort Handling

In previous section we have used a global variable to hold certain information and share it between handlers. In the majority of cases, such information is session specific, and becomes invalid if the remote party issues the SMTP `RSET` command. Therefore, `mailfromd` clears

all global variables when it receives a Milter '`abort`' request, which is normally generated by this command.

However, you may need some variables that retain their values even across SMTP session resets. In `mailfromd` terminology such variables are called *precious*. Precious variables are declared by prefixing their declaration with the keyword `precious`. Consider, for example, this snippet of code:

```
precious number rcpt_counter

prog envrcpt
do
  set rcpt_counter rcpt_counter + 1
done
```

Here, the variable '`rcpt_counter`' is declared as precious and its value is incremented each time the '`envrcpt`' handler is called. This way, '`rcpt_counter`' will keep the total number of SMTP `RCPT` commands issued during the session, no matter how many times it was restarted using the `RSET` command.

3.11 Controlling Number of Recipients

Any MTA provides a way to limit the number of recipients per message. For example, in `Sendmail` you may use the `MaxRecipientsPerMessage` option[4]. However, such methods are not flexible, so you are often better off using `mailfromd` for this purpose.

`Mailfromd` keeps the number of recipients collected so far in variable `rcpt_count`, which can be controlled in `envrcpt` handler as shown in the example below:

```
prog envrcpt
do
  if rcpt_count > 10
    reject 550 5.7.1 "Too many recipients"
  fi
done
```

This filter will accept no more than 10 recipients per message. You may achieve finer granularity by using additional conditions. For example, the following code will allow any number of recipients if the mail is coming from a domain relayed by the server, while limiting it to 10 for incoming mail from other domains:

```
prog envrcpt
do
  if not relayed(hostname($client_addr)) and rcpt_count > 10
    reject 550 5.7.1 "Too many recipients"
  fi
done
```

There are three important features to notice in the above code. First of all, it introduces two *boolean* operators: `and`, which evaluates to `true` only if both left-side and right-side expressions are `true`, and `not`, which reverses the value of its argument.

[4] *Sendmail (tm) Installation and Operation Guide*, chapter 5.6, '`O -- Set Option`'.

Secondly, the scope of an operation is determined by its *precedence*, or *binding strength*. `Not` binds more tightly than `and`, so its scope is limited by the next expression between it and `and`. Using parentheses to underline the operator scoping, the above `if` condition can be rewritten as follows:

```
if (not (relayed(hostname($client_addr)))) and (%rcpt_count > 10)
```

Finally, it is important to notice that all boolean expressions are computed using *shortcut evaluation*. To understand what it is, let's consider the following expression: *x* `and` *y*. Its value is `true` only if both *x* and *y* are `true`. Now suppose that we evaluate the expression from left to right and we find that *x* is false. This means that no matter what the value of *y* is, the resulting expression will be `false`, therefore there is no need to compute *y* at all. So, the boolean shortcut evaluation works as follows:

x `and` *y* If *x* ⇒ `false`, do not evaluate *y* and return `false`.

x `or` *y* If *x* ⇒ `true`, do not evaluate *y* and return `true`.

Thus, in the expression `not relayed(hostname($client_addr)) and rcpt_count > 10`, the value of the `rcpt_count` variable will be compared with '`10`' only if the `relayed` function yielded `false`.

To further enhance our sample filter, you may wish to make the `reject` output more informative, to let the sender know what the recipient limit is. To do so, you can use the *concatenation operator* '`.`' (a dot):

```
set max_rcpt 10
prog envrcpt
do
  if not relayed(hostname($client_addr)) and rcpt_count > 10
    reject 550 5.7.1 "Too many recipients, max=" . max_rcpt
  fi
done
```

When evaluating the third argument to `reject`, `mailfromd` will first convert `max_rcpt` to string and then concatenate both strings together, producing string '`Too many recipients, max=10`'.

3.12 Sending Rate

We have introduced the notion of mail sending rate in Section 1.4 [Rate Limit], page 8. `Mailfromd` keeps the computed rates in the special `rate` database (see Section 3.15 [Databases], page 30). Each record in this database consists of a `key`, for which the rate is computed, and the rate value, in form of a double precision floating point number, representing average number of messages per second sent by this `key` within the last sampling interval. In the simplest case, the sender email address can be used as a `key`, however we recommend to use a conjunction *email-sender_ip* instead, so the actual *email* owner won't be blocked by actions of some spammer abusing his/her address.

Two functions are provided to control and update sending rates. The `rateok` function takes three mandatory arguments:

```
bool rateok(string key, number interval, number threshold)
```

The *key* meaning is described above. The *interval* is the sampling interval, or the number of seconds to which the actual sending rate value is converted. Remember that it is

stored internally as a floating point number, and thus cannot be directly used in `mailfromd` filters, which operate only on integer numbers. To use the rate value, it is first converted to messages per given interval, which is an integer number. For example, the rate `0.138888` brought to 1-hour interval gives `500` (messages per hour).

When the `rateok` function is called, it recomputes rate record for the given *key*. If the new rate value converted to messages per given *interval* is less than *threshold*, the function updates the database and returns `True`. Otherwise it returns `False` and does not update the database.

This function must be *required* prior to use, by placing the following statement somewhere at the beginning of your script:

```
require rateok
```

For example, the following code limits the mail sending rate for each '`email address`'-'`IP`' combination to 180 per hour. If the actual rate value exceeds this limit, the sender is returned a temporary failure response:

```
require rateok

prog envfrom
do
  if not rateok($f . "-" . ${client_addr}, 3600, 180)
    tempfail 450 4.7.0 "Mail sending rate exceeded.  Try again later"
  fi
done
```

Notice argument concatenation, used to produce the key.

It is often inconvenient to specify intervals in seconds, therefore a special `interval` function is provided. It converts its argument, which is a textual string representing time interval in English, to the corresponding number of seconds. Using this function, the function invocation would be:

```
rateok($f . "-" . ${client_addr}, interval("1 hour"), 180)
```

The `interval` function is described in [interval], page 130, and time intervals are discussed in [time interval specification], page 220.

The `rateok` function begins computing the rate as soon as it has collected enough data. By default, it needs at least four mails. Since this may lead to a big number of false positives (i.e. overestimated rates) at the beginning of sampling interval, there is a way to specify a minimum number of samples `rateok` must collect before starting to actually compute rates. This number of samples is given as the optional fourth argument to the function. For example, the following call will always return `True` for the first 10 mails, no matter what the actual rate:

```
rateok($f . "-" . ${client_addr}, interval("1 hour"), 180, 10)
```

The `tbf_rate` function allows to exercise more control over the mail rates. This function implements a *token bucket filter* (TBF) algorithm.

The token bucket controls when the data can be transmitted based on the presence of abstract entities called *tokens* in a container called *bucket*. Each token represents some amount of data. The algorithm works as follows:

- A token is added to the bucket at a constant rate of 1 token per t microseconds.

- A bucket can hold at most *m* tokens. If a token arrives when the bucket is full, that token is discarded.
- When *n* items of data arrive (e.g. *n* mails), *n* tokens are removed from the bucket and the data are accepted.
- If fewer than *n* tokens are available, no tokens are removed from the bucket and the data are not accepted.

This algorithm allows to keep the data traffic at a constant rate *t* with bursts of up to *m* data items. Such bursts occur when no data was being arrived for *m***t* or more microseconds.

`Mailfromd` keeps buckets in a database '`tbf`'. Each bucket is identified by a unique *key*. The `tbf_rate` function is defined as follows:

```
bool tbf_rate(string key, number n, number t, number m)
```

The *key* identifies the bucket to operate upon. The rest of arguments is described above. The `tbf_rate` function returns '`True`' if the algorithm allows to accept the data and '`False`' otherwise.

Depending on how the actual arguments are selected the `tbf_rate` function can be used to control various types of flow rates. For example, to control mail sending rate, assign the arguments as follows: *n* to the number of mails and *t* to the control interval in microseconds:

```
prog envfrom
do
  if not tbf_rate($f . "-" . $client_addr, 1, 10000000, 20)
    tempfail 450 4.7.0 "Mail sending rate exceeded.  Try again later"
  fi
done
```

The example above permits to send at most one mail each 10 seconds. The burst size is set to 20.

Another use for the `tbf_rate` function is to limit the total delivered mail size per given interval of time. To do so, the function must be used in `prog eom` handler, because it is the only handler where the entire size of the message is known. The *n* argument must contain the number of bytes in the email (or email bytes * number of recipients), and the *t* must be set to the number of bytes per microsecond a given user is allowed to send. The *m* argument must be large enough to accommodate a couple of large emails. E.g.:

```
prog eom
do
  if not tbf_rate("$f-$client_addr",
                  message_size(current_message()),
                  10240*1000000,  # At most 10 kb/sec
                  10*1024*1024)
    tempfail 450 4.7.0 "Data sending rate exceeded.  Try again later"
  fi
done
```

See Section 5.32 [Rate limiting functions], page 189, for more information about `rateok` and `tbf_rate` functions.

3.13 Greylisting

Greylisting is a simple method of defending against the spam proposed by Evan Harris. In few words, it consists in recording the '`sender IP`'-'`sender email`'-'`recipient email`' triplet of mail transactions. Each time the unknown triplet is seen, the corresponding message is rejected with the `tempfail` code. If the mail is legitimate, this will make the originating server retry the delivery later, until the destination eventually accepts it. If, however, the mail is a spam, it will probably never be retried, so the users will not be bothered by it. Even if the spammer will retry the delivery, the *greylisting period* will give spam-detection systems, such as DNSBLs, enough time to detect and blacklist it, so by the time the destination host starts accepting emails from this triplet, it will already be blocked by other means.

You will find the detailed description of the method in The Next Step in the Spam Control War: Greylisting (`http://projects.puremagic.com/greylisting/whitepaper.html`), the original whitepaper by Evan Harris.

The `mailfromd` implementation of greylisting is based on `greylist` function. The function takes two arguments: the `key`, identifying the greylisting triplet, and the `interval`. The function looks up the key in the *greylisting database*. If such a key is not found, a new entry is created for it and the function returns `true`. If the key is found, `greylist` returns `false`, if it was inserted to the database more than `interval` seconds ago, and `true` otherwise. In other words, from the point of view of the greylisting algorithm, the function returns `true` when the message delivery should be blocked. Thus, the simplest implementation of the algorithm would be:

```
prog envrcpt
do
 if greylist("${client_addr}-$f-${rcpt_addr}", interval("1 hour"))
   tempfail 451 4.7.1 "You are greylisted"
 fi
done
```

However, the message returned by this example, is not informative enough. In particular, it does not tell when the message will be accepted. To help you produce more informative messages, `greylist` function stores the number of seconds left to the end of the greylisting period in the global variable `greylist_seconds_left`, so the above example could be enhanced as follows:

```
prog envrcpt
do
  set gltime interval("1 hour")
  if greylist("${client_addr}-$f-${rcpt_addr}", gltime)
    if greylist_seconds_left = gltime
      tempfail 451 4.7.1
         "You are greylisted for %gltime seconds"
    else
      tempfail 451 4.7.1
         "Still greylisted for %greylist_seconds_left seconds"
    fi
  fi
done
```

In real life you will have to avoid greylisting some messages, in particular those coming from the '`<>`' address and from the IP addresses in your relayed domain. It can easily be done using the techniques described in previous sections and is left as an exercise to the reader.

`Mailfromd` provides two implementations of greylisting primitives, which differ in the information stored in the database. The one described above is called *traditional*. It keeps in the database the time when the greylisting was activated for the given key, so the `greylisting` function uses its second argument (`interval`) and the current timestamp to decide whether the key is still greylisted.

The second implementation is called by the name of its inventor *Con Tassios*. This implementation stores in the database the time when the greylisting period is set to expire, computed by the `greylist` when it is first called for the given key, using the formula '`current_timestamp + interval`'. Subsequent calls to `greylist` compare the current timestamp with the one stored in the database and ignore their second argument. This implementation is enabled by one of the following pragmas:

```
#pragma greylist con-tassios
```

or

```
#pragma greylist ct
```

When Con Tassios implementation is used, yet another function becomes available. The function `is_greylisted` (see Section 5.33 [is_greylisted], page 190) returns '`True`' if its argument is greylisted and '`False`' otherwise. It can be used to check for the greylisting status without actually updating the database:

```
if is_greylisted("${client_addr}-$f-${rcpt_addr}")
  ...
fi
```

One special case is *whitelisting*, which is often used together with greylisting. To implement it, `mailfromd` provides the function `dbmap`, which takes two mandatory arguments: `dbmap(`*`file, key`*`)` (it also allows an optional third argument, see [dbmap], page 176, for more information on it). The first argument is the name of the DBM file where to search for the key, the second one is the key to be searched. Assuming you keep your whitelist database in file `/var/run/whitelist.db`, a more practical example will be:

```
prog envrcpt
do
  set gltime interval("1 hour")

  if not ($f = "" or relayed(hostname(${client_addr}))
         or dbmap("/var/run/whitelist.db", ${client_addr}))
    if greylist("${client_addr}-$f-${rcpt_addr}", gltime)
      if greylist_seconds_left = gltime
        tempfail 451 4.7.1
           "You are greylisted for %gltime seconds"
      else
        tempfail 451 4.7.1
           "Still greylisted for %greylist_seconds_left seconds"
      fi
    fi
  fi
done
```

3.14 Local Account Verification

In your filter script you may need to verify if the given user name is served by your mail server, in other words, to verify if it represents a *local account*. Notice that in this context, the word *local* does not necessarily mean that the account is local for the server running `mailfromd`, it simply means any account whose mailbox is served by the mail servers using `mailfromd`.

The `validuser` function may be used for this purpose. It takes one argument, the user name, and returns `true` if this name corresponds to a local account. To verify this, the function relies on `libmuauth`, a powerful authentication library shipped with GNU `mailutils`. More precisely, it invokes a list of *authorization* functions. Each function is responsible for looking up the user name in a particular source of information, such as system `passwd` database, an SQL database, etc. The search is terminated when one of the functions finds the name in question or the list is exhausted. In the former case, the account is local, in the latter it is not. This concept is discussed in detail in see Section "Authorization and Authentication Principles" in *GNU Mailutils Manual*). Here we will give only some practical advices for implementing it in `mailfromd` filters.

The actual list of available authorization modules depends on your `mailutils` installation. Usually it includes, apart from traditional UNIX `passwd` database, the functions for verifying PAM, RADIUS and SQL database accounts. Each of the authorization methods is configured using special configuration file statements. For the description of the Mailutils configuration files, See Section "Mailutils Configuration File" in *GNU Mailutils Manual*. You can obtain the template for `mailfromd` configuration by running `mailfromd --config-help`.

For example, the following `mailfromd.conf` file:

```
auth {
  authorization pam:system;
}

pam {
  service mailfromd;
}
```

sets up the authorization using PAM and system `passwd` database. The name of PAM service to use is '`mailfromd`'.

The function `validuser` is often used together with `dbmap`, as in the example below:

```
#pragma dbprop /etc/mail/aliases.db null

if dbmap("/etc/mail/aliases.db", localpart($rcpt_addr))
   and validuser(localpart($rcpt_addr))
  ...
fi
```

For more information about `dbmap` function, see [dbmap], page 176. For a description of `dbprop` pragma, see Section 5.26 [Database functions], page 175.

3.15 Databases

Some `mailfromd` functions use DBM databases to save their persistent state data. Each database has a unique *identifier*, and is assigned several pieces of information for its maintenance: the database *file name* and the *expiration period*, i.e. the time after which a record is considered expired.

To obtain the list of available databases along with their preconfigured settings, run *`mailfromd --show-defaults`* (see Section 3.18 [Examining Defaults], page 40). You will see an output similar to this:

```
version:             9.2
script file:         /etc/mailfromd.mfl
preprocessor:        /usr/bin/m4 -s
user:                mail
statedir:            /var/run/mailfromd
socket:              unix:/var/run/mailfromd/mailfrom
pidfile:             /var/run/mailfromd/mailfromd.pid
default syslog:          blocking
supported databases:     gdbm, bdb
default database type:   bdb
optional features:   DKIM GeoIP2 STARTTLS
greylist database:      /var/run/mailfromd/greylist.db
greylist expiration:    86400
tbf database:        /var/run/mailfromd/tbf.db
tbf expiration:      86400
rate database:      /var/run/mailfromd/rates.db
rate expiration:    86400
cache database:      /var/run/mailfromd/mailfromd.db
cache positive expiration: 86400
cache negative expiration: 43200
```

The text below '`optional features`' line describes the available built-in databases. Notice that the '`cache`' database, in contrast to the rest of databases, has two expiration periods associated with it. This is explained in the next subsection.

3.15.1 Database Formats

The version 9.2 runs the following database types (or *formats*):

'`cache`' *Cache database* keeps the information about external emails, obtained using sender verification functions (see Section 3.6 [Checking Sender Address], page 18). The key entry to this database is an email address or *email*:*sender-ip* string, for addresses checked using strict verification. The data its stores for each key are:

1. Address validity. This field can be either `success` or `not_found`, meaning the address is confirmed to exists or it is not.
2. The time when the entry was entered into the database. It is used to check for expired entries.

The '`cache`' database has two expiration periods: a *positive expiration* period, that is applied to entries with the first field set to `success`, and a *negative expiration* period, applied to entries marked as `not_found`.

'`rate`' The mail sending rate data, maintained by `rate` function (see Section 5.32 [Rate limiting functions], page 189). A record consists of the following fields:

timestamp The time when the entry was entered into the database.

interval Interval during which the rate was measured (seconds).

count Number of mails sent during this interval.

'`tbf`' This database is maintained by `tbf_rate` function (see [TBF], page 25). Each record represents a single bucket and consists of the following keys:

timestamp Timestamp of most recent token, as a 64-bit unsigned integer (microseconds resolution).

expirytime
Estimated time when this bucket expires (seconds since epoch).

tokens Number of tokens in the bucket (`size_t`).

'`greylist`'
This database is maintained by `greylist` function (see Section 3.13 [Greylisting], page 27). Each record holds only the timestamp. Its semantics depends on the greylisting implementation in use (see [greylisting types], page 28). In traditional implementation, it is the time when the entry was entered into the database. In Con Tassios implementation, it is the time when the greylisting period expires.

3.15.2 Basic Database Operations

The `mfdbtool` utility is provided for performing various operations on the `mailfromd` database.

To list the contents of a database, use `--list` option. When used without any arguments it will list the '`cache`' database:

```
$ mfdbtool --list
abrakat@mail.com           success Thu Aug 24 15:28:58 2006
baccl@EDnet.NS.CA          not_found Fri Aug 25 10:04:18 2006
bhzxhnyl@chello.pl       not_found Fri Aug 25 10:11:57 2006
brqp@aaanet.ru:24.1.173.165  not_found Fri Aug 25 14:16:06 2006
```

You can also list data for any particular key or keys. To do so, give the keys as arguments to `mfdbtool`:

```
$ mfdbtool --list abrakat@mail.com brqp@aaanet.ru:24.1.173.165
abrakat@mail.com           success Thu Aug 24 15:28:58 2006
brqp@aaanet.ru:24.1.173.165  not_found Fri Aug 25 14:16:06 2006
```

To list another database, give its format identifier with the `--format` (`-H`) option. For example, to list the '`rate`' database:

```
$ mfdbtool --list --format=rate
sam@mail.net-62.12.4.3 Wed Sep  6 19:41:42 2006  139   3 0.0216 6.82e-06
axw@rame.com-59.39.165.172 Wed Sep  6 20:26:24 2006  0  1  N/A  N/A
```

The `--format` option can be used with any database management option, described below.

Another useful operation you can do while listing '`rate`' database is the prediction of *estimated time of sending*, i.e. the time when the user will be able to send mail if currently his mail sending rate has exceeded the limit. This is done using `--predict` option. The option takes an argument, specifying the mail sending rate limit, e.g. (the second line is split for readability):

```
$ mfdbtool --predict="180 per 1 minute"
ed@fae.net-21.10.1.2 Wed Sep 13 03:53:40 2006  0 1 N/A N/A; free to send
service@19.netlay.com-69.44.129.19 Wed Sep 13 15:46:07 2006 7 2
   0.286   0.0224; in 46 sec. on Wed Sep 13 15:49:00 2006
```

Notice, that there is no need to use `--list --format=rate` along with this option, although doing so is not an error.

To delete an entry from the database, use `--delete` option, for example: *`mfdbtool --delete abrakat@mail.com`*. You can give any number of keys to delete in the command line.

3.15.3 Database Maintenance

There are two principal operations of database management: expiration and compaction. *Expiration* consists in removing expired entries from the database. In fact, it is rarely needed, since the expired entries are removed in the process of normal `mailfromd` work. Nevertheless, a special option is provided in case an explicit expiration is needed (for example, before dumping the database to another format, to avoid transferring useless information).

The command line option `--expire` instructs `mfdbtool` to delete expired entries from the specified database. As usual, the database is specified using `--format` option. If it is not given explicitly, '`cache`' is assumed.

While removing expired entries the space they occupied is marked as free, so it can be used by subsequent inserts. The database does not shrink after expiration is finished. To actually return the unused space to the file system you should *compact* your database.

This is done by running *`mfdbtool --compact`* (and, optionally, specifying the database to operate upon with `--format` option). Notice, that compacting a database needs roughly as much disk space on the partition where the database resides as is currently used by the database. Database compaction runs in three phases. First, the database is scanned and all non-expired records are stored in the memory. Secondly, a temporary database is created in the state directory and all the cached entries are flushed into it. This database is named after the PID of the running `mfdbtool` process. Finally, the temporary database is renamed to the source database.

Both `--compact` and `--expire` can be applied to all databases by combining them with `--all`. It is useful, for example, in `crontab` files. For example, I have the following monthly job in my `crontab`:

```
0 1 1 * * /usr/bin/mfdbtool --compact --all
```

3.16 Testing Filter Scripts

It is important to check your filter script before actually starting to use it. There are several ways to do so.

To test the syntax of your filter script, use the `--lint` option. It will cause `mailfromd` to exit immediately after attempting to compile the script file. If the compilation succeeds, the program will exit with code 0. Otherwise, it will exit with error code 78 ('`configuration error`'). In the latter case, `mailfromd` will also print a diagnostic message, describing the error along with the exact location where the error was diagnosed, for example:

```
mailfromd: /etc/mailfromd.mfl:39: syntax error, unexpected reject
```

The error location is indicated by the name of the file and the number of the line when the error occurred. By using the `--location-column` option you instruct `mailfromd` to also print the *column number*. E.g. with this option the above error message may look like:

```
mailfromd: /etc/mailfromd.mfl:39.12 syntax error, unexpected reject
```

Here, '`39`' is the line and '`12`' is the column number.

For complex scripts you may wish to obtain a listing of variables used in the script. This can be achieved using `--xref` command line option:

The output it produces consists of four columns:

Variable name

Data type Either `number` or `string`.

Offset in data segment
: Measured in words.

References A comma-separated list of locations where the variable was referenced. Each location is represented as *file*:*line*. If several locations pertain to the same *file*, the file name is listed only once.

Here is an example of the cross-reference output:

```
$ mailfromd --xref
Cross-references:
-----------------
cache_used               number 5   /etc/mailfromd.mfl:48
clamav_virus_name        string 9   /etc/mailfromd.mfl:240,240
db                       string 15  /etc/mailfromd.mfl:135,194,215
dns_record_ttl           number 16  /etc/mailfromd.mfl:136,172,173
ehlo_domain              string 11
gltime                   number 13  /etc/mailfromd.mfl:37,219,220,222,223
greylist_seconds_left    number 1   /etc/mailfromd.mfl:220,226,227
last_poll_host           string 2
```

If the script passes syntax check, the next step is often to test if it works as you expect it to. This is done with `--test` (`-t`) command line option. This option runs the `envfrom` handler (or another one, see below) and prints the result of its execution.

When running your script in test mode, you will need to supply the values of `Sendmail` macros it needs. You do this by placing the necessary assignments in the command line. For example, this is how to supply initial values for `f` and `client_addr` macros:

```
$ mailfromd --test f=gray@gnu.org client_addr=127.0.0.1
```

You may also need to alter initial values of some global variables your script uses. To do so, use `-v` (`--variable`) command line option. This option takes a single argument consisting of the variable name and its initial value, separated by an equals sign. For example, here is how to change the value of `ehlo_domain` global variable:

```
$ mailfromd -v ehlo_domain=mydomain.org
```

The `--test` option is often useful in conjunction with options `--debug`, `--trace` and `--transcript` (see Section 3.19 [Logging and Debugging], page 43. The following example

shows what the author got while debugging the filter script described in Section 4.27 [Filter Script Example], page 120:

```
$ mailfromd --test --debug=50 f=gray@gnu.org client_addr=127.0.0.1
MX 20 mx20.gnu.org
MX 10 mx10.gnu.org
MX 10 mx10.gnu.org
MX 20 mx20.gnu.org
getting cache info for gray@gnu.org
found status: success (0), time: Thu Sep 14 14:54:41 2006
getting rate info for gray@gnu.org-127.0.0.1
found time: 1158245710, interval: 29, count: 5, rate: 0.172414
rate for gray@gnu.org-127.0.0.1 is 0.162162
updating gray@gnu.org-127.0.0.1 rates
SET REPLY 450 4.7.0 Mail sending rate exceeded.  Try again later
State envfrom: tempfail
```

. If your script uses `echo` statements (see Section 4.19.4 [Echo], page 95), they will print their output on standard error. To direct them to the standard output, use the `--echo` option. You can also redirect the `echo` output to arbitrary file, by supplying its name as argument, as in: `--echo=`*`file`*. see [echo option], page 234.

To test any handler, other than '`envfrom`', give its name as the argument to `--test` option. Since this argument is optional, it is important that it be given immediately after the option, without any intervening white space, for example *`mailfromd --test=helo`*, or *`mailfromd -thelo`*.

This method allows to test one handler at a time. To test the script as a whole, use `mtasim` utility. When started it enters interactive mode, similar to that of `sendmail -bs`, where it expects SMTP commands on its standard input and sends answers to the standard output. The `--port=auto` command line option instructs it to start `mailfromd` and to create a unique socket for communication with it. For the detailed description of the program and the ways to use it, See Chapter 12 [mtasim], page 261.

3.17 Run Mode

Mailfromd provides a special option that allows to run arbitrary MFL scripts.

When given the `--run` command line option, `mailfromd` loads the script given in its command line, looks for the function called '`main`', and runs it.

This function must be declared as:

```
func main(...) returns number
```

Mailfromd passes all command line arguments that follow the script name as arguments to that function. When the function returns, its return value is used by `mailfromd` as exit code.

As an example, suppose the file `script.mfl` contains the following:

```
func main (...)
  returns number
do
  loop for number i 1,
```

```
        while i <= $#,
        set i i + 1
    do
      echo "arg %i=" . $(i)
    done
done
```

This function prints all its arguments (See [variadic functions], page 80, for a detailed description of functions with variable number of arguments). Now running:

```
$ mailfromd --run script.mfl 1 file dest
```

displays the following:

```
arg 1=1
arg 2=file
arg 3=dest
```

You can direct the script output to the standard output by using the `--echo`, as described above, e.g.:

```
$ mailfromd --echo --run script.mfl 1 file dest
```

Note, that MFL does not have a direct equivalent of shell's `$0` argument. If your function needs to know the name of the script that is being executed, use `__file__` built-in constant instead (see Section 4.11.1 [Built-in constants], page 63).

The name `main` is not hard-coded. You can use the `--run` option to run any function, provided that its definition is as discussed above. Just give the name of this function as the argument to the option. This argument is optional, therefore it must be separated from the option by an equals sign (with no whitespace from either side). For example, given the command line below, `mailfromd` will load the file `script.mfl` and execute the function '`start`':

```
$ mailfromd --run=start script.mfl
```

If you need to define sendmail macros (see Section 4.10 [Sendmail Macros], page 61) for use in the run mode, place the *macro=value* assignments *before* the script name, e.g.:

```
$ mailfromd --run=start i=feedbeef client_addr=::1 script.mfl
```

To summarize, the command line when using the run mode is:

```
mailfromd [options] --run [macro=value] file args...
```

Finally, notice that *file* together with *args...* can be omitted. In this case the default script file will be used (see [default script file], page 10).

The '*macro=value*' assignments define Sendmail macros, *args...* are passed as arguments to the `main` function defined in *file*, and *option* stands for any other `mailfromd` options that might be needed.

3.17.1 The Top of a Script File

The `--run` option makes it possible to use `mailfromd` scripts as standalone programs. The traditional way to do so was to set the executable bit on the script file and to begin the

script with the *interpreter selector*, i.e. the characters '`#!`' followed by the name of the `mailfromd` executable, e.g.:

```
#! /usr/sbin/mailfromd --run
```

This would cause the shell to invoke `mailfromd` with the command line constructed from the `--run` option, the name of the invoked script file itself, and any actual arguments from the invocation. Once invoked, `mailfromd` would treat the initial '`#!`' line as a usual single-line comment (see Section 4.1 [Comments], page 53).

However, the interpretation of the '`#!`' by shells has various deficiencies, which depend on the actual shell being used. For example, some shells pass any characters following the whitespace after the interpreter name as a single argument, some others silently truncate the command line after some number of characters, etc. This often make it impossible to pass additional arguments to `mailfromd`. For example, a script which begins with the following line would most probably fail to be executed properly:

```
#! /usr/sbin/mailfromd --echo --run
```

To compensate for these deficiencies and to allow for more complex invocation sequences, `mailfromd` handles initial '`#`' in a special way. If the first line of a source file begins with '`#!/`' or '`#! /`' (with a single space between '`!`' and '`/`'), it is treated as a start of a multi-line comment, which is closed by the two characters '`!#`' on a line by themselves.

Thus, the correct way to begin a `mailfromd` script is:

```
#! /usr/sbin/mailfromd --run
!#
```

Using this feature, you can start the `mailfromd` with arbitrary shell code, provided it ends with an `exec` statement invoking the interpreter itself. For example:

```
#!/bin/sh
exec /usr/sbin/mailfromd --echo --run $0 $@
!#

func main(...)
  returns number
do
  /* actual mfl code goes here */
done
```

Note the use of '`$0`' and '`$@`' to pass the actual script file name and command line arguments to `mailfromd`.

3.17.2 Parsing Command Line Arguments

A special function is provided to break (parse) the command line into options, and to check them for validity. It uses the GNU getopt routines (see Section "Getopt" in *The GNU C Library Reference Manual*).

`string getopt (number argc, pointer argv, ...)` [Built-in Function]

The `getopt` function parses the command line arguments, as supplied by *argc* and *argv*. The *argc* argument is the argument count, and *argv* is an opaque data structure,

representing the array of arguments[5]. The operator `vaptr` (see [vaptr], page 40) is provided to initialize this argument.

An argument that starts with '`-`' (and is not exactly '`-`' or '`--`'), is an option element. An argument that starts with a '`-`' is called *short* or *traditional* option. The characters of this element, except for the initial '`-`' are option characters. Each option character represents a separate option. An argument that starts with '`--`' is called *long* or *GNU* option. The characters of this element, except for the initial '`--`' form the *option name*.

Options may have arguments. The argument to a short option is supplied immediately after the option character, or as the next word in command line. E.g., if option `-f` takes a mandatory argument, then it may be given either as `-farg` or as `-f arg`. The argument to a long option is either given immediately after it and separated from the option name by an equals sign (as `--file=arg`), or is given as the next word in the command line (e.g. `--file arg`).

If the option argument is optional, i.e. it may not necessarily be given, then only the first form is allowed (i.e. either `-farg` or `--file=arg`.

The '`--`' command line argument ends the option list. Any arguments following it are not considered options, even if they begin with a dash.

If `getopt` is called repeatedly, it returns successively each of the option characters from each of the option elements (for short options) and each option name (for long options). In this case, the actual arguments are supplied only to the first invocation. Subsequent calls must be given two nulls as arguments. Such invocation instructs `getopt` to use the values saved on the previous invocation.

When the function finds another option, it returns its character or name updating the external variable `optind` (see below) so that the next call to `getopt` can resume the scan with the following option.

When there are no more options left, or a '`--`' argument is encountered, `getopt` returns an empty string. Then `optind` gives the index in *argv* of the first element that is not an option.

The legitimate options and their characteristics are supplied in additional arguments to `getopt`. Each such argument is a string consisting of two parts, separated by a vertical bar ('`|`'). Any one of these parts is optional, but at least one of them must be present. The first part specifies short option character. If it is followed by a colon, this character takes mandatory argument. If it is followed by two colons, this character takes an optional argument. If only the first part is present, the '`|`' separator may be omitted. Examples:

`"c"`
`"c|"` Short option `-c`.

`"f:"`
`"f:|"` Short option `-f`, taking a mandatory argument.

`"f::"`
`"f::|"` Short option `-f`, taking an optional argument.

[5] When MFL has array data type, the second argument will change to array of strings.

If the vertical bar is present and is followed by any characters, these characters specify the name of a long option, synonymous to the short one, specified by the first part. Any mandatory or optional arguments to the short option remain mandatory or optional for the corresponding long option. Examples:

`"f:|file"` Short option `-f`, or long option `--file`, requiring an argument.

`"f::|file"` Short option `-f`, or long option `--file`, taking an optional argument.

In any of the above cases, if this option appears in the command line, `getopt` returns its short option character.

To define a long option without a short equivalent, begin it with a bar, e.g.:

`"|help"`

If this option is to take an argument, this is specified using the mechanism described above, except that the short option character is replaced with a minus sign. For example:

`"-:|output"`
Long option `--output`, which takes a mandatory argument.

`"-::|output"`
Long option `--output`, which takes an optional argument.

If an option is returned that has an argument in the command line, `getopt` stores this argument in the variable `optarg`.

After each invocation, `getopt` sets the variable `optind` to the index of the next *argv* element to be parsed. Thus, when the list of options is exhausted and the function returned an empty string, `optind` contains the index of the the first element that is not an option.

When `getopt` encounters an option that is not described in its arguments or if it detects a missing option argument it prints an error message using `mailfromd` logging facilities, stores the offending option in the variable `optopt`, and returns '`?`'.

If printing error message is not desired (e.g. the application is going to take care of error messaging), it can be disabled by setting the variable `opterr` to '`0`'.

The third argument to `getopt`, called *controlling argument*, may be used to control the behavior of the function. If it is a colon, it disables printing the error message for unrecognized options and missing option arguments (as setting `opterr` to '`0`' does). In this case `getopt` returns '`:`', instead of '`?`' to indicate missing option argument.

If the controlling argument is a plus sign, or the environment variable `POSIXLY_CORRECT` is set, then option processing stops as soon as a non-option argument is encountered. By default, if options and non optional arguments are intermixed in *argv*, `getopt` permutes them so that the options go first, followed by non-optional arguments.

If the controlling argument is '`-`', then each non-option element in *argv* is handled as if it were the argument of an option with character code 1 ('`"\001"`', in MFL notation. This can used by programs that are written to expect options and other *argv*-elements in any order and that care about the ordering of the two.

Any other value of the controlling argument is handled as an option definition.

A special language construct is provided to supply the second argument (*argv*) to `getopt` and similar functions:

```
vaptr(param)
```

where *param* is a positional parameter, from which to start the array of *argv*. For example:

```
func main(...)
  returns number
do
  set rc getopt($#, vaptr($1), "|help")
  ...
```

Here, `vaptr($1)` constructs the *argv* array from all the arguments, supplied to the function `main`.

To illustrate the use of `getopt` function, let's suppose you write a script that takes the following options:

```
-f file
--file=file
--output[=dir]
--help
```

Then, the corresponding `getopt` invocation will be:

```
func main(...)
  returns number
do
  loop for string rc getopt($#, vaptr($1),
                            "f:|file", "-::|output", "h|help"),
       while rc != "",
       set rc getopt(0, 0)
  do
    switch rc
    do
      case "f":
        set file optarg
      case "output"
        set output 1
        set output_dir optarg
      case "h"
        help()
      default:
        return 1
    done
    ...
```

3.18 Examining Default Values

Sometimes you may need to check what are the default settings of the `mailfromd` binary and what values it uses actually. Both tasks are accomplished using the `--show-defaults` option. When used alone, it shows the settings actually in use (default values, eventually

modified by your configuration settings). When used together with `--no-config`, it displays the compiled defaults.

The output of `mailfromd --show-defaults` looks like this:

```
version:                   9.2
script file:               /etc/mailfromd.mfl
preprocessor:              /usr/bin/m4 -s -DWITH_DKIM -DWITH_MFMOD
                           /var/mailfromd/9.2/include/pp-setup
user:                      mail
statedir:                  /var/lib/mailfromd
socket:                    mailfrom
pidfile:                   mailfromd.pid
default syslog:            blocking
include path:              /etc/mailfromd:/usr/share/mailfromd/include:
                           /usr/share/mailfromd/8.14.94/include
module path:               /usr/share/mailfromd:
                           /usr/share/mailfromd/9.2
mfmod path:                /usr/lib/mailfromd
optional features:         DKIM, mfmod, STARTTLS
supported database types:  gdbm, bdb
default database type:     bdb
greylist database:         /var/lib/mailfromd/greylist.db
greylist expiration:       86400
tbf database:              /var/lib/mailfromd/tbf.db
tbf expiration:            86400
rate database:             /var/lib/mailfromd/rates.db
rate expiration:           86400
cache database:            /var/lib/mailfromd/mailfromd.db
cache positive expiration: 604800
cache negative expiration: 86400
```

The above format, called *human-readable*, with two-column output and long lines split across several physical lines, is used if `mailfromd` is linked with GNU `libmailutils` library version 3.16 or later and its standard output is connected to a terminal. Otherwise, *machine-readable* output format is used, in which additional whitespace is elided, and long lines are retained verbatim. This makes it possible to easily extract default values using familiar text processing tools, e.g.:

```
$ mailfromd --show-defaults --no-config | grep '^script file:'
script file:/etc/mailfromd.mfl
$ mailfromd --show-defaults --no-config | sed -ne '/^script file:/s///p'
/etc/mailfromd.mfl
```

The following table describes each line of the output in detail:

version Program version.

script file The script file used by the program. It is empty if the script file is not found.

preprocessor
: Preprocessor command line. See Section 4.26 [Preprocessor], page 116. This value can be changed in configuration: See Section 7.3 [conf-preprocessor], page 221.

user
: System user `mailfromd` runs as. See Section 7.10 [conf-priv], page 229.

statedir
: `mailfromd` local state directory. See [statedir], page 11.

socket
: The socket `mailfromd` listens on. If UNIX socket, the filename is shown. Unless it begins with '`/`', it is relative to the local state directory. TCP sockets are shown in [milter port specification], page 220.

  See Section 7.5 [conf-server], page 223.

pidfile
: PID file name (relative to local state directory, unless absolute).

  See Section 7.2 [conf-base], page 220.

default syslog
: Syslog implementation used: either '`blocking`', or '`non-blocking`'.

  See [syslog-async], page 11. See also Section 3.19 [Logging and Debugging], page 43.

include path
: Include search path. See [include search path], page 53.

  It can be changed from the command line, using the `-I` option (see Section 8.1.2 [General Settings], page 234), and in configuration file, using the `include-path` statement (see Section 7.2 [conf-base], page 220).

module path
: Search path for MFL modules. see [module search path], page 109.

  It can be changed from the command line, using the `-P` (`--module-path`) option (see Section 8.1.2 [General Settings], page 234), and in configuration file, using the `module-path` statement (see Section 7.2 [conf-base], page 220).

mfmod path
: Search path for dynamically loaded modules. see [mfmod-path], page 231.

optional features
: Comma-delimited list of optional features, included to `mailfromd` at compile time. It can contain the following feature names:

Feature	**Reference**
DKIM	See Section 5.38 [DKIM], page 199.
GeoIP2	See Section 5.25 [Geolocation functions], page 173.
mfmod	See Section 4.25 [mfmod], page 111.
STARTTLS	See Section 7.9 [conf-callout], page 227.

supported database types
: Comma-delimited list of supported database types. See Section 3.15 [Databases], page 30. These types can be used as scheme prefixes in database names (see [DBM scheme], page 229).

default database type
: Type of the DBM used by default. See Section 3.15 [Databases], page 30.

greylist database
greylist expiration
: File name and record expiration time of the greylisting database. See [greylist database], page 32.

tbf database
tbf expiration
: File name and record expiration time of the token-bucket filter rate-limiting database. See [tbf database], page 32.

rate database
rate expiration
: See [rate database], page 31, File name and record expiration time of the legacy rate-limiting database. See Section 5.32 [Rate limiting functions], page 189.

cache database
cache positive expiration
cache negative expiration
: File name and record expiration times of the call-out cache database. See [cache database], page 31.

The database settings can be changed using Section 7.11 [conf-database], page 229.

3.19 Logging and Debugging

Depending on its operation mode, `mailfromd` tries to guess whether it is appropriate to print its diagnostics and informational messages on standard error or to send them to syslog. Standard error is assumed if the program is run with one of the following command line options:

- `--test` (see Section 3.16 [Testing Filter Scripts], page 33)
- `--run` (see Section 3.17 [Run Mode], page 35)
- `--lint` (see Section 3.16 [Testing Filter Scripts], page 33)
- `--dump-code` (see Section 8.1.5 [Logging and Debugging Options], page 236)
- `--dump-grammar-trace` (see Section 8.1.5 [Logging and Debugging Options], page 236)
- `--dump-lex-trace` (see Section 8.1.5 [Logging and Debugging Options], page 236)
- `--dump-macros` (see Section 8.1.5 [Logging and Debugging Options], page 236)
- `--dump-tree` (see Section 8.1.5 [Logging and Debugging Options], page 236)
- `--xref` (or `--dump-xref`) (see Section 3.16 [Testing Filter Scripts], page 33)

If none of these are used, `mailfromd` switches to syslog as soon as it finishes its startup. There are two ways to communicate with the `syslogd` daemon: using the `syslog` function from the system `libc` library, which is a *blocking* implementation in most cases, or via internal, *asynchronous*, syslog implementation. Whether the latter is compiled in and which implementation is used by default is determined when compiling the package, as described in [syslog-async], page 11.

The `--logger` command line option allows you to manually select the diagnostic channel:

`--logger=stderr`
: Log everything to the standard error.

`--logger=syslog`
: Log to syslog.

`--logger=syslog:async`
: Log to syslog using the asynchronous syslog implementation.

Another way to select the diagnostic channel is by using the `logger` statement in the configuration file. The statement takes the same argument as its command line counterpart.

The rest of details regarding diagnostic output are controlled by the `logging` configuration statement.

The default syslog facility is '`mail`'; it can be changed using the `--log-facility` command line option or `facility` statement. Argument in both cases is a valid facility name, i.e. one of: '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', and '`local0`' through '`local7`'. The argument can be given in upper, lower or mixed cases, and it can be prefixed with '`log_`':

Another syslog-related parameter that can be configured is the *tag*, which identifies `mailfromd` messages. The default tag is the program name. It is changed by the `--log-tag` (`-L` command line option and the `tag` logging statement.

The following example configures both the syslog facility and tag:

```
logging {
  facility local7;
  tag "mfd";
}
```

As any other UNIX utility, `mailfromd` is very quiet unless it has something important to communicate, such as, e.g. an error condition. A set of command line options is provided for controlling the verbosity of its output.

The `--trace` option enables tracing Sendmail actions executed during message verifications. When this option is given, any `accept`, `discard`, `continue`, etc. triggered during execution of your filter program will leave their traces in the log file. Here is an example of how it looks like (syslog time stamp, tag and PID removed for readability):

```
k8DHxvO09030656: /etc/mailfromd.mfl:45: reject 550 5.1.1 Sender validity
not confirmed
```

This shows that while verifying the message with ID '`k8DHxvO09030656`' the `reject` action was executed by filter script `/etc/mailfromd.mfl` at line 45.

The use of message ID in the log deserves a special notice. The program will always identify its log messages with the '`Message-Id`', when it is available. Your responsibility as an administrator is to make sure it is available by configuring your MTA to export the macro '`i`' to `mailfromd`. The rule of thumb is: make '`i`' available to the very first handler `mailfromd` executes. It is not necessary to export it to the rest of the handlers, since `mailfromd` will cache it. For example, if your filter script contains '`envfrom`' and '`envrcpt`' handlers, export '`i`' for '`envfrom`'. The exact instructions on how to ensure it depend on the MTA you use. For '`Sendmail`', refer to Section 9.1 [Sendmail], page 243. For MeTA1, see

Section 9.2 [MeTA1], page 244, and Section 13.1.2 [pmult-macros], page 272. For '`Postfix`', see Section 9.3 [Postfix], page 246.

To push log verbosity further, use the `debug` configuration statement (see Section 7.7 [conf-debug], page 225) or its command line equivalent, `--debug` (`-d`, see [–debug], page 236). Its argument is a *debugging level*, whose syntax is described in `http://mailutils.org/wiki/Debug_level`.

The debugging output is controlled by a set of levels, each of which can be set independently of others. Each debug level consists of a category name, which identifies the part of package for which additional debugging is desired, and a level number, which indicates how verbose should its output be.

Valid debug levels are:

error
: Displays error conditions which are normally not reported, but passed to the caller layers for handling.

trace0 through trace9
: Ten levels of verbosity, `trace0` producing less output, `trace9` producing the maximum amount of output.

prot
: Displays network protocol interaction, where applicable.

The overall debugging level is specified as a list of individual levels, delimited with semicolons. Each individual level can be specified as one of:

!*category*
: Disables all levels for the specified category.

category
: Enables all levels for the specified category.

category.level
: For this category, enables all levels from '`error`' to *level*, inclusive.

category.=level
: Enables only the given *level* in this *category*.

category.!level
: Disables all levels from '`error`' to *level*, inclusive, in this *category*.

category.!=level
: Disables only the given *level* in this *category*.

category.levelA-levelB
: Enables all levels in the range from *levelA* to *levelB*, inclusive.

category.!levelA-levelB
: Disables all levels in the range from *levelA* to *levelB*, inclusive.

Additionally, a comma-separated list of level specifications is allowed after the dot. For example, the following specification:

```
acl.prot,!=trace9,!trace2
```

enables in category acl all levels, except trace9, trace0, trace1, and trace2.

Implementation and applicability of each level of debugging differs between various categories. Categories built-in to mailutils are described in `http://mailutils.org/wiki/Debug_level`. Mailfromd introduces the following additional categories:

db

trace0	Detailed debugging info about expiration and compaction.
trace5	List records being removed.

dns

trace8	Verbose information about attempted DNS queries and their results.
trace9	Enables '`libadns`' internal debugging.

srvman

trace0	Additional information about normal conditions, such as subprocess exiting successfully or a remote party being allowed access by ACL.
trace1	Detailed transcript of server manager actions: startup, shutdown, subprocess cleanups, etc.
trace3	Additional info about fd sets.
trace4	Individual subserver status information.
trace5	Subprocess registration.

pmult

trace1	Verbosely list incoming connections, functions being executed and erroneous conditions: missing headers in SMFIR_CHGHEADER, undefined macros, etc.
trace2	List milter requests being processed.
trace7	List SMTP body content in SMFIR_REPLBODY requests.
error	Verbosely list mild errors encountered: bad recipient addresses, etc.

callout

trace0	Verification session transcript.
trace1	MX servers checks.
trace5	List emails being checked.
trace9	Additional info.

main

trace5	Info about hostnames in relayed domain list

engine Debugging of the virtual engine.

trace5	Message modification lists.

	trace6	Debug message modification operations and Sendmail macros registered.
	trace7	List SMTP stages (‘`xxfi_*`’ calls).
	trace9	Cleanup calls.
pp	Preprocessor.	
	trace1	Show command line of the preprocessor being run.
prog		
	trace8	Stack operations
	trace9	Debug exception state save/restore operations.
spf		
	error	Mild errors.
	trace0	List calls to ‘`spf_eval_record`’, ‘`spf_test_record`’, ‘`spf_check_host_internal`’, etc.
	trace1	General debug info.
	trace6	Explicitly list A records obtained when processing the ‘`a`’ SPF mechanism.

Categories starting with ‘`bi_`’ debug built-in modules:

bi_db	Database functions.	
	trace5	List database look-ups.
	trace6	Trace operations on the greylisting database.
bi_sa	SpamAssassin and ClamAV API.	
	trace1	Report the findings of the ‘`clamav`’ function.
	trace9	Trace payload in interactions with ‘`spamd`’.
bi_io	I/O functions.	
	trace1	Debug the following functions: `open`, `spawn`, `write`.
	trace2	Report stderr redirection.
	trace3	Report external commands being run.
bi_mbox	Mailbox functions.	
	trace1	Report opened mailboxes.
bi_other	Other built-ins.	
	trace1	Report results of checks for existence of usernames.

For example, the following invocation enables levels up to ‘`trace2`’ in category ‘`engine`’, all levels in category ‘`savsrv`’ and levels up to ‘`trace0`’ in category ‘`srvman`’:

```
$ mailfromd --debug='engine.trace2;savsrv;srvman.trace0'
```

You need to have sufficient knowledge about `mailfromd` internal structure to use this form of the `--debug` option.

To control the execution of the sender verification functions (see Section 5.21 [SMTP Callout functions], page 161), you may use `--transcript` (`-X`) command line option which enables transcripts of SMTP sessions in the logs. Here is an example of the output produced running *`mailfromd --transcript`*:

```
k8DHxlCa001774: RECV: 220 spf-jail1.us4.outblaze.com ESMTP Postfix
k8DHxlCa001774: SEND: HELO mail.gnu.org.ua
k8DHxlCa001774: RECV: 250 spf-jail1.us4.outblaze.com
k8DHxlCa001774: SEND: MAIL FROM: <>
k8DHxlCa001774: RECV: 250 Ok
k8DHxlCa001774: SEND: RCPT TO: <t1Kmx17Q@malaysia.net>
k8DHxlCa001774: RECV: 550 <>: No thank you rejected: Account
 Unavailable: Possible Forgery
k8DHxlCa001774: poll exited with status: not_found; sent
 "RCPT TO: <t1Kmx17Q@malaysia.net>", got "550 <>: No thank you
 rejected: Account Unavailable: Possible Forgery"
k8DHxlCa001774: SEND: QUIT
```

3.20 Runtime Errors

A *runtime error* is a special condition encountered during execution of the filter program, that makes further execution of the program impossible. There are two kinds of runtime errors: fatal errors, and uncaught exceptions. Whenever a runtime error occurs, `mailfromd` writes into the log file the following message:

```
RUNTIME ERROR near file:line: text
```

where *file*:*line* indicates approximate source file location where the error occurred and *text* gives the textual description of the error.

Fatal runtime errors

Fatal runtime errors are caused by a condition that is impossible to fix at run time. For version 9.2 these are:

Not enough memory
: There is not enough memory for the execution of the program. Try to make more memory available for `mailfromd` or to reduce its memory requirements by rewriting your filter script.

Out of stack space; increase #pragma stacksize
Heap overrun; increase #pragma stacksize
memory chunk too big to fit into heap
: These errors are reported when there is not enough space left on stack to perform the requested operation, and the attempt to resize the stack has failed. Usually `mailfromd` expands the stack when the need arises (see [automatic stack resizing], page 55). This runtime error indicates that there were no more memory available for stack expansion. Try to make more memory available for `mailfromd` or to reduce its memory requirements by rewriting your filter script.

Stack underflow
: Program attempted to pop a value off the stack but the stack was already empty. This indicates an internal error in the MFL compiler or `mailfromd` runtime engine. If you ever encounter this error, please report it to `bug-mailfromd@gnu.org.ua`. Include the log fragment (about 10-15 lines before and after this log message) and your filter script. See Chapter 14 [Reporting Bugs], page 279, for more information about bug reporting.

pc out of range
: The *program counter* is out of allowed range. This is a severe error, indicating an internal inconsistency in `mailfromd` runtime engine. If you encounter it, please report it to `bug-mailfromd@gnu.org.ua`. Include the log fragment (about 10-15 lines before and after this log message) and your filter script. See Chapter 14 [Reporting Bugs], page 279, for more information about how to report a bug.

Programmatic runtime errors

These indicate a programmatic error in your filter script, which the MFL compiler was unable to discover at compilation stage:

Invalid exception number: *n*
: The `throw` statement used a not existent exception number *n*. Fix the statement and restart `mailfromd`. See [throw], page 104, for the information about `throw` statement and see Section 4.22 [Exceptions], page 99, for the list of available exception codes.

No previous regular expression
: You have used a back-reference (see Section 4.13 [Back references], page 69), where there is no previous regular expression to refer to. Fix this line in your code and restart the program.

Invalid back-reference number
: You have used a back-reference (see Section 4.13 [Back references], page 69), with a number greater than the number of available groups in the previous regular expression. For example:

```
if $f matches "(.*)@gnu.org"
  # Wrong: there is only one group in the regexp above!
  set x \2
...
```

: Fix your code and restart the daemon.

Uncaught exceptions

Another kind of runtime errors are *uncaught exceptions*, i.e. exceptional conditions for which no handler was installed (See Section 4.22 [Exceptions], page 99, for information on exceptions and on how to handle them). These errors mean that the programmer (i.e. you), made no provision for some specific condition. For example, consider the following code:

```
prog envfrom
do
  if $f mx matches "yahoo.com"
    foo()
  fi
done
```

It is syntactically correct, but it overlooks the fact that `mx matches` may generate `e_temp_failure` exception, if the underlying DNS query has timed out (see Section 4.17.7 [Special comparisons], page 86). If this happens, `mailfromd` has no instructions on what to do next and reports an error. This can easily be fixed using a `try/catch` (see Section 4.22.3 [Catch and Throw], page 101) statement, e.g.:

```
prog envfrom
do
  try
  do
    if $f mx matches "yahoo.com"
      foo()
    fi
  done
  # Catch DNS errors
  catch e_temp_failure or e_failure
  do
    tempfail 451 4.1.1 "MX verification failed"
  done
done
```

Another common case are undefined Sendmail macros. In this case the `e_macroundef` exception is generated:

```
RUNTIME ERROR near foo.c:34: Macro not defined: {client_adr}
```

These can be caused either by misspelling the macro name (as in the example message above) or by failing to export the required name in Sendmail milter configuration (see [exporting macros], page 243). This error should be fixed either in your source code or in `sendmail.cf` file, but if you wish to provide a special handling for it, you can use the following catch statement:

```
catch e_macroundef
do
  ...
done
```

Sometimes the location indicated with the runtime error message is not enough to trace the origin of the error. For example, an error can be generated explicitly with `throw` statement (see [throw], page 104):

```
RUNTIME ERROR near match_cidr.mfl:30: invalid CIDR (text)
```

If you look in module `match_cidr.mfl`, you will see the following code (line numbers added for reference):

```
23 func match_cidr(string ipstr, string cidr)
24   returns number
```

```
25 do
26   number netmask
27
28   if cidr matches '^(([0-9]{1,3}\.){3}[0-9]{1,3})/([0-9][0-9]?)'
29     return inet_aton(ipstr) & len_to_netmask(\3) = inet_aton(\1)
30   else
31     throw invcidr "invalid CIDR (%cidr)"
32   fi
33   return 0
34 done
```

Now, it is obvious that the value of `cidr` argument to `match_cidr` was wrong, but how to find the caller that passed the wrong value to it? The special command line option `--stack-trace` is provided for this. This option enables dumping *stack traces* when a fatal error occurs. Traces contain information about function calls. Continuing our example, using the `--stack-trace` option you will see the following diagnostics:

```
RUNTIME ERROR near match_cidr.mfl:30: invalid CIDR (127%)
mailfromd: Stack trace:
mailfromd: 0077: match_cidr.mfl:31: match_cidr
mailfromd: 0096: test.mfl:13: bar
mailfromd: 0110: mailfromd.mfl:18: foo
mailfromd: Stack trace finishes
mailfromd: Execution of the configuration program was not finished
```

Each trace line describes one stack frame. The lines appear in the order of most recently called to least recently called. Each frame consists of:

1. Value of the program counter at the time of its execution;
2. Source code location, if available;
3. Name of the function called.

Thus, the example above can be read as: "the function `match_cidr` was called by the function `bar` in file `test.mfl` at line 13. This function was called from the function `bar`, in file `test.mfl` at line 13. In its turn, `bar` was called by the function `foo`, in file `mailfromd.mfl` at line 18".

Examining caller functions will help you localize the source of the error and fix it.

You can also request a stack trace any place in your code, by calling the `stack_trace` function. This can be useful for debugging.

3.21 Notes and Cautions

This section discusses some potential culprits in the MFL.

It is important to execute special caution when writing format strings for `sprintf` (see Section 5.5 [String formatting], page 133) and `strftime` (see [strftime], page 181) functions. They use '`%`' as a character introducing conversion specifiers, while the same character is used to expand a MFL variable within a string. To prevent this misinterpretation, always enclose format specification in *single quotes* (see [singe-vs-double], page 60). To illustrate this, let's consider the following example:

```
echo sprintf ("Mail from %s", $f)
```

If a variable `s` is not declared, this line will produce the '`Variable s is not defined`' error message, which will allow you to identify and fix the bug. The situation is considerably worse if `s` is declared. In that case you will see no warning message, as the statement is perfectly valid, but at the run-time the variable `s` will be interpreted within the format string, and its value will replace `%s`. To prevent this from happening, single quotes must be used:

```
echo sprintf ('Mail from %s', $f)
```

This does not limit the functionality, since there is no need to fall back to variable interpretation in format strings.

Yet another dangerous feature of the language is the way to refer to variable and constant names within literal strings. To expand a variable or a constant the same notation is used (See Section 4.12 [Variables], page 65, and see Section 4.11 [Constants], page 62). Now, lets consider the following code:

```
const x 2
string x "X"

prog envfrom
do
  echo "X is %x"
done
```

Does `%x` in `echo` refers to the variable or to the constant? The correct answer is '`to the variable`'. When executed, this code will print '`X is X`'.

As of version 9.2, `mailfromd` will always print a diagnostic message whenever it stumbles upon a variable having the same name as a previously defined constant or vice versa. The resolution of such name clashes is described in detail in See [variable–constant shadowing], page 91.

Future versions of the program may provide a non-ambiguous way of referring to variables and constants from literal strings.

4 Mail Filtering Language

The *mail filtering language*, or MFL, is a special language designed for writing filter scripts. It has a simple syntax, similar to that of Bourne shell. In contrast to the most existing programming languages, MFL does not have any special terminating or separating characters (like, e.g. newlines and semicolons in shell)[1]. All syntactical entities are separated by any amount of white-space characters (i.e. spaces, tabulations or newlines).

The following sections describe MFL syntax in detail.

4.1 Comments

Two types of comments are allowed: C-style, enclosed between '`/*`' and '`*/`', and shell-style, starting with '`#`' character and extending up to the end of line:

```
/* This is
   a comment. */
# And this too.
```

There are, however, several special cases, where the characters following '`#`' are not ignored:

If the first line begins with '`#!/`' or '`#! /`', this is treated as a start of a multi-line comment, which is closed by the characters '`!#`' on a line by themselves. This feature allows for writing sophisticated scripts. See Section 3.17.1 [top-block], page 36, for a detailed description.

A '`#`' is followed by '`include`', '`include_once`', '`line`', '`error`', or '`warning`' is treated specially. These cases are covered by the subsequent sections.

4.2 #include and #include_once

If '`#`' is followed by word '`include`' (with optional whitespace between them), this statement requires inclusion of the specified file, as in C. There are two forms of the '`#include`' statement:

1. `#include <`*file*`>`
2. `#include "`*file*`"`

The quotes around *file* in the second form quotes are optional.

Both forms are equivalent if *file* is an absolute file name. Otherwise, the first form will look for *file* in the *include search path*. The second one will look for it in the current working directory first, and, if not found there, in the include search path.

The default include search path is:

1. *prefix*`/share/mailfromd/include`
2. *prefix*`/share/mailfromd/9.2/include`

[1] There are two noteworthy exceptions: `module` and `from ... import` statements, which must be terminated with a period. For details, refer to Section 4.24.1 [module structure], page 108, and Section 4.24.3 [import], page 109.

where *prefix* is the installation prefix.

New directories can be appended in front of it using `-I` (`--include-path`) command line option, or `include-path` configuration statement (see Section 7.2 [conf-base], page 220).

For example, invoking

```
$ mailfromd -I/var/mailfromd -I/com/mailfromd
```

creates the following include search path

1. `/var/mailfromd`
2. `/com/mailfromd`
3. *prefix*`/share/mailfromd/include`
4. *prefix*`/share/mailfromd/9.2/include`

Along with `#include`, there is also a special form `#include_once`, that has the same syntax:

```
#include_once <file>
#include_once "file"
```

This form works exactly as `#include`, except that, if the *file* has already been included, it will not be included again. As the name suggests, it will be included only once.

This form should be used to prevent re-inclusions of a code, which can cause problems due to function redefinitions, variable reassignments etc.

4.3 #line

A line in the form

```
#line number "identifier"
```

causes the MFL compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by *number* and the current input file is named by *identifier*. If the identifier is absent, the remembered file name does not change.

Line directives in `cpp` style are also understood:

```
# number "identifier"
```

4.4 #warning and #error

If '`#`' is followed by the word '`warning`', any amount of whitespace and a string in double quotes, the compiler will use that string to generate a warning message at that point. As usual, whitespace characters are allowed between '`#`' and '`warning`':

```
# warning "The code below is suspicious"
```

Similarly, '`#`' followed by the word '`error`', whitespace and a doubly-quoted string causes the compiler to generate a compilation error at that point.

To use backslash or double quote in the message text, precede them with a single slash, e.g.:

```
#error "the \"quoted\" text"
```

A backslash in front of any other character is retained.

4.5 Pragmatic comments

If '`#`' is immediately followed by word '`pragma`' (with optional whitespace between them), such a construct introduces a *pragmatic comment*, i.e. an instruction that controls some configuration setting.

The available pragma types are described in the following subsections.

4.5.1 Pragma prereq

The `#pragma prereq` statement ensures that the correct `mailfromd` version is used to compile the source file it appears in. It takes version number as its arguments and produces a compilation error if the actual `mailfromd` version number is earlier than that. For example, the following statement:

```
#pragma prereq 7.0.94
```

results in error if compiled with `mailfromd` version 7.0.93 or prior.

4.5.2 Pragma stacksize

The `stacksize` pragma sets the initial size of the run-time stack and may also define the policy of its growing, in case it becomes full. The default stack size is 4096 words. You may need to increase this number if your configuration program uses recursive functions or does an excessive amount of string manipulations.

`stacksize` *size* [*incr* [*max*]] [pragma]

Sets stack size to *size* units. Optional *incr* and *max* define stack growth policy (see below). The default *units* are words. The following example sets the stack size to 7168 words:

```
#pragma stacksize 7168
```

The *size* may end with a *unit size* suffix:

Suffix	**Meaning**
k	Kiloword, i.e. 1024 words
m	Megawords, i.e. 1048576 words
g	Gigawords,
t	Terawords (ouch!)

Table 4.1: Unit Size Suffix

File suffixes are case-insensitive, so the following two pragmas are equivalent and set the stack size to `7*1048576 = 7340032` words:

```
#pragma stacksize 7m
#pragma stacksize 7M
```

When the MFL engine notices that there is no more stack space available, it attempts to expand the stack. If this attempt succeeds, the operation continues. Otherwise, a runtime error is reported and the execution of the filter stops.

The optional *incr* argument to `#pragma stacksize` defines growth policy for the stack. Two growth policies are implemented: *fixed increment policy*, which expands stack in a fixed number of *expansion chunks*, and *exponential growth policy*, which duplicates

the stack size until it is able to accommodate the needed number of words. The fixed increment policy is the default. The default chunk size is 4096 words.

If *incr* is the word '`twice`', the duplicate policy is selected. Otherwise *incr* must be a positive number optionally suffixed with a size suffix (see above). This indicates the expansion chunk size for the fixed increment policy.

The following example sets initial stack size to 10240, and expansion chunk size to 2048 words:

```
#pragma stacksize 10M 2K
```

The pragma below enables exponential stack growth policy:

```
#pragma stacksize 10240 twice
```

In this case, when the run-time evaluator hits the stack size limit, it expands the stack to twice the size it had before. So, in the example above, the stack will be sequentially expanded to the following sizes: 20480, 40960, 81920, 163840, etc.

The optional *max* argument defines the maximum size of the stack. If stack grows beyond this limit, the execution of the script will be aborted.

If you are concerned about the execution time of your script, you may wish to avoid stack reallocations. To help you find out the optimal stack size, each time the stack is expanded, `mailfromd` issues a warning in its log file, which looks like this:

```
warning: stack segment expanded, new size=8192
```

You can use these messages to adjust your stack size configuration settings.

4.5.3 Pragma regex

The '`#pragma regex`', controls compilation of regular expressions. You can use any number of such pragma directives in your `mailfromd.mfl`. The scope of '`#pragma regex`' extends to the next occurrence of this directive or to the end of the script file, whichever occurs first.

`regex` [*push*|*pop*] *flags* [pragma]

The optional *push*|*pop* parameter is one of the words '`push`' or '`pop`' and is discussed in detail below. The *flags* parameter is a whitespace-separated list of *regex flags*. Each regex-flag is a word specifying some regex feature. It can be preceded by '`+`' to enable this feature (this is the default), by '`-`' to disable it or by '`=`' to reset regex flags to its value. Valid regex-flags are:

'`extended`'
: Use POSIX Extended Regular Expression syntax when interpreting regex. If not set, POSIX Basic Regular Expression syntax is used.

'`icase`'
: Do not differentiate case. Subsequent regex searches will be case insensitive.

'`newline`'
: *Match-any-character* operators don't match a newline.

  A non-matching list ('`[^...]`') not containing a newline does not match a newline.

  Match-beginning-of-line operator ('`^`') matches the empty string immediately after a newline.

Match-end-of-line operator (‘`$`’) matches the empty string immediately before a newline.

For example, the following pragma enables POSIX extended, case insensitive matching (a good thing to start your `mailfromd.mfl` with):

```
#pragma regex +extended +icase
```

Optional modifiers ‘`push`’ and ‘`pop`’ can be used to maintain a stack of regex flags. The statement

```
#pragma regex push [flags]
```

saves current regex flags on stack and then optionally modifies them as requested by *flags*.

The statement

```
#pragma regex pop [flags]
```

does the opposite: restores the current regex flags from the top of stack and applies *flags* to it.

This statement is useful in module and include files to avoid disturbing user regex settings. E.g.:

```
#pragma regex push +extended +icase
 .
 .
 .
#pragma regex pop
```

4.5.4 Pragma dbprop

`dbprop` *pattern prop . . .* [pragma]

This pragma configures properties for a DBM database. See Section 5.26 [Database functions], page 175, for its detailed description.

4.5.5 Pragma greylist

`greylist` *type* [pragma]

Selects the greylisting implementation to use. Allowed values for *type* are:

traditional
gray
: Use the traditional greylisting implementation. This is the default.

con-tassios
ct
: Use Con Tassios greylisting implementation.

See [greylisting types], page 28, for a detailed description of these greylisting implementations.

Notice, that this pragma can be used only once. A second use of this pragma would constitute an error, because you cannot use both greylisting implementations in the same program.

4.5.6 Pragma miltermacros

`miltermacros` *handler macro ...* [pragma]

Declare that the Milter stage *handler* uses MTA macro listed as the rest of arguments. The *handler* argument must be a valid handler name (see Section 4.14 [Handlers], page 70).

Normally, the `mailfromd` parser collects names of the macros used in each handler and registers them for use by that handler. This includes all '`$name`' references (see Section 4.10 [Sendmail Macros], page 61) appearing within the handler, in all functions called by that handler, as well as in the `prog action` hook (see Section 4.15.3 [action hook], page 77), if the handler invokes any Sendmail action explicitly.

In particular, macro names given as arguments to `getmacro` and `macro_defined` functions are recognized, as long as these arguments are constant strings, or can be optimized to such. However, if thes arguments are constructed dynamically, `mailfromd` cannot predict which macros will be needed. In this case, the necessary macros should be declared using `#pragma miltermacros`.

During initial negotiation with the MTA, `mailfromd` will ask it to export macro names declared by the procedure described above. The MTA is free to honor or to ignore this request. In particular, Sendmail versions prior to 8.14.0 and Postfix versions prior to 2.5 do not support this feature. If you use one of these, you will need to export the needed macros explicitly in the MTA configuration. For more details, refer to the section in Chapter 9 [MTA Configuration], page 243, corresponding to your MTA type.

4.5.7 Pragma provide-callout

The `#pragma provide-callout` statement is used in the `callout` module to inform `mailfromd` that the module has been loaded.

Do not use this pragma.

4.6 Data Types

The `mailfromd` filter script language operates on entities of two types: numeric and string.

The *numeric* type is represented internally as a signed long integer. Depending on the machine architecture, its size can vary. For example, on machines with Intel-based CPUs it is 32 bits long.

A *string* is a string of characters of arbitrary length. Strings can contain any characters except ASCII NUL.

There is also a *generic pointer*, which is designed to facilitate certain operations. It appears only in the `body` handler. See [body handler], page 73, for more information about it.

4.7 Numbers

A *decimal number* is any sequence of decimal digits, not beginning with '`0`'.

An *octal number* is '`0`' followed by any number of octal digits ('`0`' through '`7`'), for example: `0340`.

A *hex number* is '`0x`' or '`0X`' followed by any number of hex digits ('`0`' through '`9`' and '`a`' through '`f`' or '`A`' through '`F`'), for example: `0x3ef1`.

4.8 Literals

A literal is any sequence of characters enclosed in single or double quotes.

After `tempfail` and `reject` actions two special kinds of literals are recognized: three-digit numeric values represent RFC 2821 reply codes, and literals consisting of tree digit groups separated by dots represent an extended reply code as per RFC 1893/2034. For example:

```
510   # A reply code
5.7.1 # An extended reply code
```

Double-quoted strings

String literals enclosed in double quotation marks (*double-quoted strings*) are subject to *backslash interpretation*, *macro expansion*, *variable interpretation* and *back reference interpretation*.

Backslash interpretation is performed at compilation time. It consists in replacing the following *escape sequences* with the corresponding single characters:

Sequence	Replaced with
\a	Audible bell character (ASCII 7)
\b	Backspace character (ASCII 8)
\f	Form-feed character (ASCII 12)
\n	Newline character (ASCII 10)
\r	Carriage return character (ASCII 13)
\t	Horizontal tabulation character (ASCII 9)
\v	Vertical tabulation character (ASCII 11)

Table 4.2: Backslash escapes

In addition, the sequence '`\`*`newline`*' has the same effect as '`\n`', for example:

```
"a string with\
 embedded newline"
"a string with\n embedded newline"
```

Any escape sequence of the form '`\x`*`hh`*', where *h* denotes any hex digit is replaced with the character whose ASCII value is *hh*. For example:

```
"\x61nother" ⇒ "another"
```

Similarly, an escape sequence of the form '`\0`*`ooo`*', where *o* is an octal digit, is replaced with the character whose ASCII value is *ooo*.

Macro expansion and variable interpretation occur at run-time. During these phases all Sendmail macros (see Section 4.10 [Sendmail Macros], page 61), `mailfromd` variables (see Section 4.12 [Variables], page 65), and constants (see Section 4.11 [Constants], page 62) referenced in the string are replaced by their actual values. For example, if the Sendmail

macro `f` has the value '`postmaster@gnu.org.ua`' and the variable `last_ip` has the value '`127.0.0.1`', then the string[2]

```
"$f last connected from %last_ip;"
```

will be expanded to

```
"postmaster@gnu.org.ua last connected from 127.0.0.1;"
```

A *back reference* is a sequence '`\d`', where *d* is a decimal number. It refers to the *d*th parenthesized subexpression in the last `matches` statement[3]. Any back reference occurring within a double-quoted string is replaced by the value of the corresponding subexpression. See Section 4.17.7 [Special comparisons], page 86, for a detailed description of this process. Back reference interpretation is performed at run time.

Single-quoted strings

Any characters enclosed in single quotation marks are read unmodified.

The following examples contain pairs of equivalent strings:

```
"a string"
'a string'

"\\(.*\\):"
'\(.*\):'
```

Notice the last example. Single quotes are particularly useful in writing regular expressions (see Section 4.17.7 [Special comparisons], page 86).

4.9 Here Documents

Here-document is a special form of a string literal is, allowing to specify multiline strings without having to use backslash escapes. The format of here-documents is:

```
<<[flags]word
...
word
```

[2] Implementation note: actually, the references are not interpreted within the string, instead, each such string is split at compilation time into a series of concatenated atoms. Thus, our sample string will actually be compiled as:

```
$f . " last connected from " . last_ip . ";"
```

See Section 4.17.3 [Concatenation], page 85, for a description of this construct. You can easily see how various strings are interpreted by using `--dump-tree` option (see [–dump-tree], page 237). In this case, it will produce:

```
CONCAT:
  CONCAT:
    CONCAT:
      SYMBOL: f
      CONSTANT: " last connected from "
    VARIABLE last_ip (13)
  CONSTANT: ";"
```

[3] The subexpressions are numbered by the positions of their opening parentheses, left to right.

The `<<word` construct instructs the parser to read all the following lines up to the line containing only *word*, with possible trailing blanks. The lines thus read are concatenated together into a single string. For example:

```
set str <<EOT
A multiline
string
EOT
```

The body of a here-document is interpreted the same way as double-quoted strings (see [Double-quoted strings], page 59). For example, if Sendmail macro `f` has the value `jsmith@some.com` and the variable `count` is set to `10`, then the following string:

```
set s <<EOT
<$f> has tried to send %count mails.
Please see docs for more info.
EOT
```

will be expanded to:

```
<jsmith@some.com> has tried to send 10 mails.
Please see docs for more info.
```

If the *word* is quoted, either by enclosing it in single quote characters or by prepending it with a backslash, all interpretations and expansions within the document body are suppressed. For example:

```
set s <<'EOT'
The following line is read verbatim:
<$f> has tried to send %count mails.
Please see docs for more info.
EOT
```

Optional *flags* in the here-document construct control the way leading white space is handled. If *flags* is `-` (a dash), then all leading tab characters are stripped from input lines and the line containing *word*. Furthermore, if `-` is followed by a single space, all leading whitespace is stripped from them. This allows here-documents within configuration scripts to be indented in a natural fashion. Examples:

```
<<- TEXT
    <$f> has tried to send %count mails.
    Please see docs for more info.
TEXT
```

Here-documents are particularly useful with `reject` actions (see [reject and tempfail syntax], page 92).

4.10 Sendmail Macros

Sendmail macros are referenced exactly the same way they are in `sendmail.cf` configuration file, i.e. '`$name`', where *name* represents the macro name. Notice, that the notation is the same for both single-character and multi-character macro names. For consistency with the `Sendmail` configuration the '`${name}`' notation is also accepted.

Another way to reference Sendmail macros is by using function `getmacro` (see Section 5.1 [Macro access], page 125).

Sendmail macros evaluate to string values.

Notice, that to reference a macro, you must properly export it in your MTA configuration. Attempt to reference a not exported macro will result in raising a `e_macroundef` exception at the run time (see [uncaught exceptions], page 49).

4.11 Constants

A *constant* is a symbolic name for an MFL value. Constants are defined using `const` statement:

```
[qualifier] const name expr
```

where *name* is an identifier, and *expr* is any valid MFL expression evaluating immediately to a constant literal or numeric value. Optional *qualifier* defines the scope of visibility for that constant (see Section 4.24.2 [scope of visibility], page 109): either `public` or `static`.

Once defined, any appearance of *name* in the program text is replaced by its value. For example:

```
const x 10/5
const text "X is "
```

defines the numeric constant '`x`' with the value '`5`', and the literal constant '`text`' with the value '`X is `'.

A special construct is provided to define a series of numeric constants (an *enumeration*):

```
[qualifier] const
do
  name0 [expr0]
  name1 [expr1]
  ...
  nameN [exprN]
done
```

Each *exprN*, if present, must evaluate to a constant numeric expression. The resulting value will be assigned to constant *nameN*. If *exprN* is not supplied, the constant will be defined to the value of the previous constant plus one. If *expr0* is not supplied, 0 is assumed.

For example, consider the following statement

```
const
do
  A
  B
  C 10
  D
done
```

This defines '`A`' to 0, '`B`' to 1, '`C`' to 10 and '`D`' to 11.

As a matter of fact, *exprN* may also evaluate to a constant string expression, provided that all expressions in the enumeration '`const`' statement are provided. That is, the following is correct:

```
const
do
  A "one"
  B "two"
  C "three"
  D "four"
done
```

whereas the following is not:

```
const
do
  A "one"
  B
  C "three"
  D "four"
done
```

Trying to compile the latter example will produce:

```
mailfromd: filename:5.3: initializer element is not numeric
```

which means that `mailfromd` was trying to create constant '`B`' with the value of '`A`' incremented by one, but was unable to do so, because the value in question was not numeric.

Constants can be used in normal MFL expressions as well as in literals. To expand a constant within a literal string, prepend a percent sign to its name, e.g.:

```
echo "New %text %x" ⇒ "New X is 2"
```

This way of expanding constants creates an ambiguity if there happen to be a variable of the same name as the constant. See [variable–constant clashes], page 52, for more information of this case and ways to handle it.

4.11.1 Built-in constants

Several constants are built into the MFL compiler. To discern them from user-defined ones, their names start and end with two underscores ('`__`').

The following constants are defined in `mailfromd` version 9.2:

`string __file__` [Built-in constant]
: Expands to the name of the current source file.

`string __function__` [Built-in constant]
: Expands to the name of the current lexical context, i.e. the function or handler name.

`string __git__` [Built-in constant]
: This built-in constant is defined for alpha versions only. Its value is the Git tag of the recent commit corresponding to that version of the package. If the release contains some uncommitted changes, the value of the '`__git__`' constant ends with the suffix '`-dirty`'.

`number __line__` [Built-in constant]
: Expands to the current line number in the input source file.

`number __major__` [Built-in constant]
Expands to the major version number.

The following example uses `__major__` constant to determine if some version-dependent feature can be used:

```
if __major__ > 2
  # Use some version-specific feature
fi
```

`number __minor__` [Built-in constant]
Expands to the minor version number.

`string __module__` [Built-in constant]
Expands to the name of the current module (see Section 4.24 [Modules], page 108).

`string __package__` [Built-in constant]
Expands to the package name (‘`mailfromd`’)

`number __patch__` [Built-in constant]
For alpha versions and maintenance releases expands to the version patch level. For stable versions, expands to ‘`0`’.

`string __defpreproc__` [Built-in constant]
Expands to the default external preprocessor command line, if the preprocessor is used, or to an empty string if it is not, e.g.:

```
__defpreproc__ ⇒ "/usr/bin/m4 -s"
```

See Section 4.26 [Preprocessor], page 116, for information on preprocessor and its features.

`string __preproc__` [Built-in constant]
Expands to the current external preprocessor command line, if the preprocessor is used, or to an empty string if it is not. Notice, that it equals `__defpreproc__`, unless the preprocessor was redefined using `--preprocessor` command line option (see Section 4.26 [Preprocessor], page 116).

`string __version__` [Built-in constant]
Expands to the textual representation of the program version (e.g. ‘`3.0.90`’)

`string __defstatedir__` [Built-in constant]
Expands to the default state directory (see [statedir], page 11).

`string __statedir__` [Built-in constant]
Expands to the current value of the program state directory (see [statedir], page 11). Notice, that it is the same as `__defstatedir__` unless the state directory was redefined at run time.

Built-in constants can be used as variables, this allows to expand them within strings or here-documents. The following example illustrates the common practice used for debugging configuration scripts:

```
func foo(number x)
do
  echo "%__file__:%__line__: foo called with arg %x"
  ...
done
```

If the function `foo` were called in line 28 of the script file `/etc/mailfromd.mfl`, like this: `foo(10)`, you will see the following string in your logs:

```
/etc/mailfromd.mfl:28: foo called with arg 10
```

4.12 Variables

Variables represent regions of memory used to hold variable data. These memory regions are identified by *variable names*. A variable name must begin with a letter or underscore and must consist of letters, digits and underscores.

Each variable is associated with its *scope of visibility*, which defines the part of source code where it can be used (see Section 4.24.2 [scope of visibility], page 109). Depending on the scope, we discern three main classes of variables: public, static and automatic (or local).

Public variables have indefinite lexical scope, so they may be referred to anywhere in the program. *Static* are variables visible only within their module (see Section 4.24 [Modules], page 108). *Automatic* or *local variables* are visible only within the given function or handler.

Public and static variables are sometimes collectively called *global*.

These variable classes occupy separate *namespaces*, so that an automatic variable can have the same name as an existing public or static one. In this case this variable is said to *shadow* its global counterpart. All references to such a name will refer to the automatic variable until the end of its scope is reached, where the global one becomes visible again.

Likewise, a static variable may have the same name as a static variable defined in another module. However, it may not have the same name as a public variable.

A variable is *declared* using the following syntax:

```
[qualifiers] type name
```

where *name* is the variable name, *type* is the type of the data it is supposed to hold. It is '`string`' for string variables and '`number`' for numeric ones.

For example, this is a declaration of a string variable '`var`':

```
string var
```

If a variable declaration occurs within a function (see Section 4.16 [Functions], page 78) or handler (see Section 4.14 [Handlers], page 70), it declares an automatic variable, local to this function or handler. Otherwise, it declares a global variable.

Optional *qualifiers* are allowed only in global declarations, i.e. in the variable declarations that appear outside of functions. They specify the scope of the variable. The `public` qualifier declares the variable as public and the `static` qualifier declares it as static. The default scope is '`public`', unless specified otherwise in the module declaration (see Section 4.24.1 [module structure], page 108).

Additionally, *qualifiers* may contain the word `precious`, which instructs the compiler to mark this variable as *precious*. (see Section 3.10 [precious variables], page 22). The value of

the precious variable is not affected by the SMTP 'RSET' command. If both scope qualifier and `precious` are used, they may appear in any order, e.g.:

```
static precious string rcpt_list
```

or

```
precious static string rcpt_list
```

Declaration can be followed by any valid MFL expression, which supplies the initial value or *initializer* for the variable, for example:

```
string var "test"
```

A variable declared without initializer is implicitly initialized to a null value, no matter what its scope: a numeric variable assumes initial value 0, a string variables is initialized to an empty string.

A variable is assigned a value using the `set` statement:

```
set name expr
```

where *name* is the variable name and *expr* is a `mailfromd` expression (see Section 4.17 [Expressions], page 85). The effect of this statement is that the *expr* is evaluated and the value it yields is assigned to the variable *name*.

If the `set` statement is located outside a function or handler definition, the *expr* must be a constant expression, i.e. the compiler should be able to evaluate it immediately.

It is not an error to assign a value to a variable that is not declared. In this case the assignment first declares a global or automatic variable having the type of *expr* and then assigns a value to it. Automatic variable is created if the assignment occurs within a function or handler, global variable is declared if it occurs at topmost lexical level. This is called *implicit variable declaration*.

In the MFL program, variables are referenced by their name. When appearing inside a double-quoted string, variables are referenced using the notation '`%name`'. Any variable being referenced must have been declared earlier (either explicitly or implicitly).

4.12.1 Predefined Variables

Several variables are predefined. In `mailfromd` version 9.2 these are:

`number milter_state` [Predefined Variable]

Identifies the current *milter state* (see [milter state], page 13). The module `milter.mfl` defines the following symbolic names:

```
milter_state_none
milter_state_startup
milter_state_shutdown
milter_state_begin
milter_state_end
milter_state_connect
milter_state_helo
milter_state_envfrom
milter_state_envrcpt
milter_state_data
milter_state_header
milter_state_eoh
milter_state_body
milter_state_eom
milter_action
```

Use the `milter_state_name` function to obtain the corresponding textual string (see Section 5.43 [Informative Functions], page 212).

`string milter_server_id` [Predefined Variable]
Identifier of the milter server which executes the code. This is the string passed to the `id` statement in the `server` section of the configuration file (see Section 7.5 [conf-server], page 223),

`string milter_server_address` [Predefined Variable]
Address of the socket the milter server is listening to. This is defined by the `listen` statement in the `server` section of the configuration file (see Section 7.5 [conf-server], page 223),

`number milter_server_family` [Predefined Variable]
Address family of the milter server address, as defined by the `listen` statement in the `server` section of the configuration file (see Section 7.5 [conf-server], page 223). See the `FAMILY_` constants in Table 4.3.

`string milter_client_address` [Predefined Variable]
Address of the milter client which initiated the connection.

`number milter_client_family` [Predefined Variable]
Address family of `milter_client_address`. See the `FAMILY_` constants in Table 4.3.

`number cache_used` [Predefined Variable]
This variable is set by `stdpoll` and `strictpoll` built-ins (and, consequently, by the `on poll` statement). Its value is '`1`' if the function used the cached data instead of directly polling the host, and '`0`' if the polling took place. See Section 5.21 [SMTP Callout functions], page 161.

You can use this variable to make your reject message more informative for the remote party. The common paradigm is to define a function, returning empty string if the result was obtained from polling, or some notice if cached data were used, and to use the function in the `reject` text, for example:

```
func cachestr() returns string
do
  if cache_used
    return "[CACHED] "
  else
    return ""
  fi
done
```

Then, in `prog envfrom` one can use:

```
on poll $f
do
when not_found or failure:
  reject 550 5.1.0 cachestr() . "Sender validity not confirmed"
done
```

`number clamav_stream_max_length` [Predefined Variable]

Maximum size of a message chunk, used when sending message to `ClamAV` daemon. Must be less than or equal to the value of `StreamMaxLength` parameter in `clamd.conf`.

Default value is 4096.

See Section 5.31.2 [ClamAV], page 188.

`string clamav_virus_name` [Predefined Variable]

Name of virus identified by `ClamAV`. Set by `clamav` function (see Section 5.31.2 [ClamAV], page 188).

`number greylist_seconds_left` [Predefined Variable]

Number of seconds left to the end of greylisting period. Set by `greylist` and `is_greylisted` functions (see Section 5.34 [Special test functions], page 190).

`string ehlo_domain` [Predefined Variable]

Name of the domain used by polling functions in SMTP `EHLO` or `HELO` command. Default value is the fully qualified domain name of the host where `mailfromd` is run. See Section 4.23 [Polling], page 104.

`string last_poll_greeting` [Predefined Variable]

Callout functions (see Section 5.21 [SMTP Callout functions], page 161) set this variable before returning. It contains the initial SMTP reply from the last polled host.

`string last_poll_helo` [Predefined Variable]

Callout functions (see Section 5.21 [SMTP Callout functions], page 161) set this variable before returning. It contains the reply to the `HELO` (`EHLO`) command, received from the last polled host.

`string last_poll_host` [Predefined Variable]

Callout functions (see Section 5.21 [SMTP Callout functions], page 161) set this variable before returning. It contains the host name or IP address of the last polled host.

`string last_poll_recv` [Predefined Variable]
Callout functions (see Section 5.21 [SMTP Callout functions], page 161) set this variable before returning. It contains the last SMTP reply received from the remote host. In case of multi-line replies, only the first line is stored. If nothing was received the variable contains the string '`nothing`'.

`string last_poll_sent` [Predefined Variable]
Callout functions (see Section 5.21 [SMTP Callout functions], page 161) set this variable before returning. It contains the last SMTP command sent to the polled host. If nothing was sent, `last_poll_sent` contains the string '`nothing`'.

`string mailfrom_address` [Predefined Variable]
Email address used by polling functions in SMTP `MAIL FROM` command (see Section 4.23 [Polling], page 104.). Default is '`<>`'. Here is an example of how to change it:

```
set mailfrom_address "postmaster@my.domain.com"
```

You can set this value to a comma-separated list of email addresses, in which case the probing will try each address until either the remote party accepts it or the list of addresses is exhausted, whichever happens first.

It is not necessary to enclose emails in angle brackets, as they will be added automatically where appropriate. The only exception is null return address, when used in a list of addresses. In this case, it should always be written as '`<>`'. For example:

```
set mailfrom_address "postmaster@my.domain.com, <>"
```

`number sa_code` [Predefined Variable]
Spam score for the message, set by `sa` function (see [sa], page 186).

`number rcpt_count` [Predefined Variable]
The variable `rcpt_count` keeps the number of recipients given so far by `RCPT TO` commands. It is defined only in '`envrcpt`' handlers.

`number sa_threshold` [Predefined Variable]
Spam threshold, set by `sa` function (see [sa], page 186).

`string sa_keywords` [Predefined Variable]
Spam keywords for the message, set by `sa` function (see [sa], page 186).

`number safedb_verbose` [Predefined Variable]
This variable controls the verbosity of the exception-safe database functions. See [safedb_verbose], page 177.

4.13 Back references

A *back reference* is a sequence '`\d`', where *d* is a decimal number. It refers to the *d*th parenthesized subexpression in the last `matches` statement[4]. Any back reference occurring

[4] The subexpressions are numbered by the positions of their opening parentheses, left to right.

within a double-quoted string is replaced with the value of the corresponding subexpression. For example:

```
if $f matches '.*@\(.*\)\.gnu\.org\.ua'
  set host \1
fi
```

If the value of `f` macro is '`smith@unza.gnu.org.ua`', the above code will assign the string '`unza`' to the variable `host`.

Notice, that each occurrence of `matches` will reset the table of back references, so try to use them as early as possible. The following example illustrates a common error, when the back reference is used after the reference table has been reused by another matching:

```
# Wrong!
if $f matches '.*@\(.*\)\.gnu\.org\.ua'
  if $f matches 'some.*'
    set host \1
  fi
fi
```

This will produce the following run time error:

```
mailfromd: RUNTIME ERROR near file.mfl:3: Invalid back-reference number
```

because the inner match ('`some.*`') does not have any parenthesized subexpressions.

See Section 4.17.7 [Special comparisons], page 86, for more information about `matches` operator.

4.14 Handlers

Milter stage handler (or *handler*, for short) is a subroutine responsible for processing a particular milter state. There are eight handlers available. Their order of invocation and arguments are described in Figure 3.1.

A handler is defined using the following construct:

```
prog handler-name
do
  handler-body
done
```

where *handler-name* is the name of the handler (see [handler names], page 13), *handler-body* is the list of filter statements composing the handler body. Some handlers take arguments, which can be accessed within the *handler-body* using the notation *$n*, where *n* is the ordinal number of the argument. Here we describe the available handlers and their arguments:

`connect` (*string $1, number $2, number $3, string $4*) [Handler]

Invocation:

This handler is called once at the beginning of each SMTP connection.

Arguments:

1. `string`; The host name of the message sender, as reported by MTA. Usually it is determined by a reverse lookup on the host address. If the reverse lookup fails, '`$1`' will contain the message sender's IP address enclosed in square brackets (e.g. '`[127.0.0.1]`').

2. `number`; Socket address family. You need to require the '`status`' module to get symbolic definitions for the address families. Supported families are:

Constant	Value	Meaning
FAMILY_STDIO	0	Standard input/output (the MTA is run with `-bs` option)
FAMILY_UNIX	1	UNIX socket
FAMILY_INET	2	IPv4 protocol
FAMILY_INET6	3	IPv6 protocol

Table 4.3: Supported socket families

3. `number`; Port number if '`$2`' is '`FAMILY_INET`'.
4. `string`; Remote IP address if '`$2`' is '`FAMILY_INET`' or full file name of the socket if '`$2`' is '`FAMILY_UNIX`'. If '`$2`' is '`FAMILY_STDIO`', '`$4`' is an empty string.

The actions (see Section 4.19.1 [Actions], page 92) appearing in this handler are handled by Sendmail in a special way. First of all, any textual message is ignored. Secondly, the only action that immediately closes the connection is `tempfail 421`. Any other reply codes result in Sendmail switching to *nullserver* mode, where it accepts any commands, but answers with a failure to any of them, except for the following: `QUIT`, `HELO`, `NOOP`, which are processed as usual.

The following table summarizes the Sendmail behavior depending on the action used:

`tempfail 421` *`excode message`*
: The caller is returned the following error message:

```
421 4.7.0 hostname closing connection
```

Both *excode* and *message* are ignored.

`tempfail 4`*`xx excode message`*
: (where *xx* represents any digits, except '`21`') Both *excode* and *message* are ignored. Sendmail switches to nullserver mode. Any subsequent command, excepting the ones listed above, is answered with

```
454 4.3.0 Please try again later
```

`reject 5`*`xx excode message`*
: (where *xx* represents any digits). All arguments are ignored. Sendmail switches to nullserver mode. Any subsequent command, excepting ones listed above, is answered with

```
550 5.0.0 Command rejected
```

Regarding reply codes, this behavior complies with RFC 2821 (section 3.9), which states:

> An SMTP server *must not* intentionally close the connection except:
> [...]
> - After detecting the need to shut down the SMTP service and returning a 421 response code. This response code can be issued after the server

receives any command or, if necessary, asynchronously from command receipt (on the assumption that the client will receive it after the next command is issued).

However, the RFC says nothing about textual messages and extended error codes, therefore Sendmail's ignoring of these is, in my opinion, absurd. My practice shows that it is often reasonable, and even necessary, to return a meaningful textual message if the initial connection is declined. The opinion of `mailfromd` users seems to support this view. Bearing this in mind, `mailfromd` is shipped with a patch for Sendmail 8.18.2, which makes it honor both extended return code and textual message given with the action. The patch file is `etc/sendmail-8.18.2.connect.diff`.

`helo` (*string $1*) [Handler]

Invocation:

This handler is called whenever the SMTP client sends `HELO` or `EHLO` command. Depending on the actual MTA configuration, it can be called several times or even not at all.

Arguments:

1. `string`; Argument to `HELO` (`EHLO`) commands.

Notes: According to RFC 28221, `$1` must be domain name of the sending host, or, in case this is not available, its IP address enclosed in square brackets. Be careful when taking decisions based on this value, because in practice many hosts send arbitrary strings. We recommend to use `heloarg_test` function (see [heloarg_test], page 191) if you wish to analyze this value.

`envfrom` (*string $1, string $2*) [Handler]

Invocation:

Called when the SMTP client sends `MAIL FROM` command, i.e. once at the beginning of each message.

Arguments:

1. `string`; First argument to the `MAIL FROM` command, i.e. the email address of the sender.
2. `string`; Rest of arguments to `MAIL FROM` separated by space character. This argument can be '`""`'.

Notes

1. `$1` is not the same as `$f` Sendmail variable, because the latter contains the sender email after address rewriting and normalization, while `$1` contains exactly the value given by sending party.
2. When the array type is implemented, `$2` will contain an array of arguments.

`envrcpt` (*string $1, string $2*) [Handler]

Invocation:

Called once for each `RCPT TO` command, i.e. once for each recipient, immediately after `envfrom`.

Arguments:

1. `string`; First argument to the `RCPT TO` command, i.e. the email address of the recipient.
2. `string`; Rest of arguments to `RCPT TO` separated by space character. This argument can be '`""`'.

Notes: When the array type is implemented, `$2` will contain an array of arguments.

`data` () [Handler]

Invocation:

Called after the MTA receives SMTP '`DATA`' command. Notice that this handler is not supported by Sendmail versions prior to 8.14.0 and Postfix versions prior to 2.5.

Arguments:

None

`header` (*string $1, string $2*) [Handler]

Invocation:

Called once for each header line received after SMTP `DATA` command.

Arguments:

1. `string`; Header field name.
2. `string`; Header field value. The content of the header may include folded white space, i.e., multiple lines with following white space where lines are separated by LF (ASCII 10). The trailing line terminator (CR/LF) is removed.

`eoh` [Handler]

Invocation:

This handler is called once per message, after all headers have been sent and processed.

Arguments:

None.

`body` (*pointer $1, number $2*) [Handler]

Invocation:

This header is called zero or more times, for each piece of the message body obtained from the remote host.

Arguments:

1. `pointer`; Piece of body text. See '`Notes`' below.
2. `number`; Length of data pointed to by `$1`, in bytes.

Notes: The first argument points to the body chunk. Its size may be quite considerable and passing it as a string may be costly both in terms of memory and execution time. For this reason it is not passed as a string, but rather as a *generic pointer*, i.e. an object having the same size as

`number`, which can be used to retrieve the actual contents of the body chunk if the need arises.

A special function `body_string` is provided to convert this object to a regular MFL string (see Section 5.15 [Mail body functions], page 153). Using it you can collect the entire body text into a single global variable, as illustrated by the following example:

```
string text

prog body
do
  set text text . body_string($1,$2)
done
```

The text collected this way can then be used in the `eom` handler (see below) to parse and analyze it.

If you wish to analyze both the headers and mail body, the following code fragment will do that for you:

```
string text

# Collect all headers.
prog header
do
  set text text . $1 . ": " . $2 . "\n"
done

# Append terminating newline to the headers.
prog eoh
do
  set text "%text\n"
done

# Collect message body.
prog body
do
  set text text . body_string($1, $2)
done
```

`eom` [Handler]

Invocation:

This handler is called once per message, when the terminating dot after `DATA` command has been received.

Arguments:

None

Notes: This handler is useful for calling *message capturing* functions, such as `sa` or `clamav`. For more information about these, refer to Section 5.31 [Interfaces to Third-Party Programs], page 185.

For your reference, the following table shows each handler with its arguments:

Handler	**$1**	**$2**	**$3**	**$4**
connect	Hostname	Socket Family	Port	Remote address
helo	`HELO` domain	N/A	N/A	N/A
envfrom	Sender email address	Rest of arguments	N/A	N/A
envrcpt	Recipient email address	Rest of arguments	N/A	N/A
header	Header name	Header value	N/A	N/A
eoh	N/A	N/A	N/A	N/A
body	Body segment (pointer)	Length of the segment (numeric)	N/A	N/A
eom	N/A	N/A	N/A	N/A

Table 4.4: State Handler Arguments

4.14.1 Multiple Handler Definitions

Any handler may be declared multiple times. When compiling the filter program, `mailfromd` combines the code from all `prog` declarations having the same handler name into one code block and compiles it. The resulting code is guaranteed to be executed in the order in which it appears in the source files.

4.15 Initialization and Cleanup Handlers

Apart from the milter handlers described in the previous section, MFL provides several *special handlers*, that serve as *hooks*, allowing the programmer to insert code in certain important points of the control flow.

Syntactically, special handlers are similar to milter state handlers, i.e. they are defined as:

```
prog handler
do
  ...
done
```

(*handler* being the handler name).

Special handlers can be subdivided into three groups.

The first group are `begin` and `end` handlers. These are run at the beginning and before the end of each SMTP session and are used to provide a session-specific initialization and cleanup routines.

The second group are `startup` and `shutdown` handlers, which provide global initialization and cleanup routines. These handlers are invoked exactly once: `startup` when `mailfromd` has started up, but hasn't yet begun to serve milter requests, and `shutdown` when `mailfromd` is about to terminate.

Finally, the `action` handler is run before executing each reply action (see [reply actions], page 92).

4.15.1 The 'begin' and 'end' special handlers

These two special handlers are executed once for each session, marking its beginning and end. Neither of them takes any arguments:

`begin` [Handler]

```
# Begin handler
prog begin
do
  ...
done
```

The `begin` handler is run once for each SMTP session, after the connection has been established but before the first milter handler has been called.

`end` [Handler]

```
# End handler
prog end
do
  ...
done
```

The `end` handler is run once for each SMTP session, after all other handlers have finished their work and `mailfromd` has already returned the resulting status to the MTA and closed connection.

Multiple '`begin`' and '`end`' handlers are a useful feature when writing modules (see Section 4.24 [Modules], page 108), because each module can thus have its own initialization and cleanup blocks. Notice, however, that in this case the order in which subsequent '`begin`' and '`end`' blocks are executed is not defined. It is only warranted that all '`begin`' blocks are executed at startup and all '`end`' blocks are executed at shutdown. It is also warranted that all '`begin`' and '`end`' blocks defined within a compilation unit (i.e. a single abstract source file, with all `#include` and `#include_once` statements expanded in place) are executed in order of their appearance in the unit.

Due to their special nature, the `begin` and `end` blocks impose certain restrictions on the statements that can be used within them:

1. Sendmail actions cannot be used `prog end`
2. Sendmail macros cannot be used in `prog end`
3. Header manipulation actions (see [header manipulation], page 93) cannot be used in '`end`' handler.

The '`begin`' handlers are the usual place to put global initialization code to. For example, if you do not want to use DNS caching, you can do it this way:

```
prog begin
do
  db_set_active("dns", 0)
done
```

Additionally, you can set up global exception handling routines there. For example, the following '`begin`' statement installs a handler for all exceptions not handled otherwise that logs the exception along with the stack trace and continues processing the message:

```
prog begin
do
  catch *
  do
    echo "Caught exception $1: $2"
    stack_trace()
    continue
  done
done
```

4.15.2 Global startup and shutdown handlers.

Yet another pair of special handlers, `startup` and `shutdown`, can be used for global initialization and cleanup.

`startup` [Handler]

The `startup` handler is called exactly once, as a part of `mailfromd` startup session.

This handler is normally used in *mfmod* interface modules to load the shared library part (see Section 4.25 [mfmod], page 111).

`shutdown` [Handler]

This handler is called during the normal program shutdown sequence, before exiting.

Both handlers bear certain similarity to `begin` and `end`: they take no arguments, and their use is subject to the same restrictions (see [begin/end restrictions], page 76). Additionally, the following features cannot be used in global handlers:

1. Sendmail macros.
2. Message modification functions.

4.15.3 Action Hook

Action hook handler is run implicitly before executing each reply action, such as `accept`, `reject`, etc. See [reply actions], page 92, for a discussion of reply action statements.

Upon invocation, the handler is passed six arguments:

1. Identifier of the action which is about to be performed. Each action is assigned a numeric identifier. The `status` module defines the following symbolic names for action identifiers:

   ```
   ACCEPT_ACTION
   CONTINUE_ACTION
   DISCARD_ACTION
   REJECT_ACTION
   TEMPFAIL_ACTION
   ```

   To convert these to textual action names, use the `milter_action_name` function (see Section 5.43 [Informative Functions], page 212).
2. SMTP code of the reply
3. Extended reply code
4. Textual message passed to the action.

5. File name of the source location where the action was called.
6. Line number of source location where the action was called.

Last three arguments are meaningful only for `reject` and `tempfail` actions. For the remaining three actions (`accept`, `discard`, and `continue`), empty strings are passed.

Action hook handlers are useful mainly for logging and accounting purposes. For example, the code fragment below assumes that the `openmetrics` module is used (see *mfmod_openmetrics reference*). It increases the corresponding metrics before taking the action. Additionally, for `reject` and `tempfail` actions, the metrics '`reject_`*code*' and '`tempfail_`*code*' are increased, where *code* is the three-digit SMTP status code being sent to the server.

```
prog action
do
  openmetrics_incr(milter_action_name($1))
  switch $1
  do
  case REJECT_ACTION:
     openmetrics_incr("reject_" . $2)
  case TEMPFAIL_ACTION:
     openmetrics_incr("tempfail_" . $2)
  done
done
```

4.16 Functions

A *function* is a named `mailfromd` subroutine, which takes zero or more *parameters* and optionally returns a certain value. Depending on the return value, functions can be subdivided into *string functions* and *number functions*. A function may have *mandatory* and *optional parameters*. When invoked, the function must be supplied exactly as many *actual arguments* as the number of its mandatory parameters.

Functions are invoked using the following syntax:

```
name (args)
```

where *name* is the function name and *args* is a comma-separated list of expressions. For example, the following are valid function calls:

```
foo(10)
interval("1 hour")
greylist("/var/my.db", 180)
```

The number of parameters a function takes and their data types compose the *function signature*. When actual arguments are passed to the function, they are converted to types of the corresponding formal parameters.

There are two major groups of functions: *built-in* functions, that are implemented in the `mailfromd` binary, and *user-defined* functions, that are written in MFL. The invocation syntax is the same for both groups.

`Mailfromd` is shipped with a rich set of *library functions*. These are described in Chapter 5 [Library], page 125. In addition to these you can define your own functions.

Function definitions can appear anywhere between the handler declarations in a filter program, the only requirement being that the function definition occur before the place where the function is invoked.

The syntax of a function definition is:

```
[qualifier] func name (param-decl)
  [returns data-type]
do
  function-body
done
```

where *name* is the name of the function to define, *param-decl* is a comma-separated list of parameter declarations. The syntax of the latter is the same as that of variable declarations (see Section 4.12 [Variables], page 65), i.e.:

```
type name
```

declares the parameter *name* having the type *type*. The *type* is `string` or `number`.

Optional *qualifier* declares the scope of visibility for that function (see Section 4.24.2 [scope of visibility], page 109). It is similar to that of variables, except that functions cannot be local (i.e. you cannot declare function within another function).

The `public` qualifier declares a function that may be referred to from any module, whereas the `static` qualifier declares a function that may be called only from the current module (see Section 4.24 [Modules], page 108). The default scope is '`public`', unless specified otherwise in the module declaration (see Section 4.24.1 [module structure], page 108).

For example, the following declares a function '`sum`', that takes two numeric arguments and returns a numeric value:

```
func sum(number x, number y) returns number
```

Similarly, the following is a declaration of a static function:

```
static func sum(number x, number y) returns number
```

Parameters are referenced in the *function-body* by their name, the same way as other variables. Similarly, the value of a parameter can be altered using `set` statement.

A function can be declared to take a certain number of *optional arguments*. In a function declaration, optional abstract arguments must be placed after the mandatory ones, and must be separated from them with a semicolon. The following example is a definition of function `foo`, which takes two mandatory and two optional arguments:

```
func foo(string msg, string email; number x, string pfx)
```

Mandatory parameters are: `msg` and `email`. Optional parameters are: `x` and `pfx`. The actual number of arguments supplied to the function is returned by a special construct `$#`. In addition, the special construct `@`*arg* evaluates to the ordinal number of variable *arg* in the list of formal parameters (the first argument has number '`0`'). These two constructs can be used to verify whether an argument is supplied to the function.

When an actual argument for parameter `n` is supplied, the number of actual arguments (`$#`) is greater than the ordinal number of that parameter in the declaration list (`@n`). Thus, the following construct can be used to check if an optional argument *arg* is actually supplied:

```
func foo(string msg, string email; number x, string arg)
do
```

```
if $# > @arg
  ...
fi
```

The default `mailfromd` installation provides a special macro for this purpose: see [defined], page 118. Using it, the example above could be rewritten as:

```
func foo(string msg, string email; number x, string arg)
do
  if defined(arg)
    ...
  fi
```

Within a function body, optional arguments are referenced exactly the same way as the mandatory ones. Attempt to dereference an optional argument for which no actual parameter was supplied, results in an undefined value, so be sure to check whether a parameter is passed before dereferencing it.

A function can also take variable number of arguments (such functions are called *variadic*). This is indicated by ellipsis in place of the last abstract parameter name. The statement below defines a function `foo` taking one mandatory, one optional and any number of additional arguments:

```
func foo (string a ; string b, string ...)
```

The data type before the ellipsis indicates the type to promote all actual arguments to. If it is omitted, `string` is assumed, so the above declaration can also be written as:

```
func foo (string a ; string b, ...)
```

To refer to the actual arguments in the function body, the following construct is used:

```
$(expr)
```

where *expr* is any valid MFL expression, evaluating to a number *n*. This construct refers to the value of *n*th actual parameter from the variable argument list. Parameters are numbered from '1', so the first variable parameter is `$(1)`, and the last one is `$($# - Nm - No)`, where *Nm* and *No* are numbers of mandatory and optional parameters to the function.

The construct '`$(n)`' where 1 <= *n* <= 9 can also be written as '`$n`'.

For example, the function below prints all its arguments:

```
func pargs (string text, ...)
do
  echo "text=%text"
  loop for number i 1,
       while i < $# - @text,
       set i i + 1
  do
    echo "arg %i=" . $(i)
  done
done
```

Note how the ordinal number operator is used to compute the upper limit.

As another example, the function below computes the sum of its arguments.

```
func sum(number ...)
do
  number s 0
  loop for number i 1,
       while i <= $#,
       set i i + 1
  do
    set s s + $(i)
  done
  return s
done
```

Sometimes it is necessary to pass all variable arguments passed to a variadic function on to another variadic function. To do so, use the `$@` operator. For example:

```
func y(string x, number ...)
do
  echo "x is " . sum($@)
done
```

Suppose `y` is called as `y("test", 1, 3, 5)`. Then, it will call `sum` as: `sum(1, 3, 5)`.

The `$@` can be used with a numeric argument, which indicates number of arguments to remove from the resulted argument list. This is similar to `shift` statement in other languages. Thus, if `y` in the above example were written as:

```
func y(string x, ...)
do
  x($@(2))
done
```

then `y("test", "a", "b", "c")`, it will call `x` as follows: `x("c")`.

Notice the following important points. First, `$@` can be used only as the last argument in the argument list. Secondly, it cannot be used to pass mandatory and optional arguments to a function. In other words, arguments passed via `$@` must correspond to ellipsis in the function declaration. Finally, passing shift count greater than the actual number of variable arguments results in a runtime error.

The *function-body* is any list of valid `mailfromd` statements. In addition to the statements discussed below (see Section 4.19 [Statements], page 92) it can also contain the `return` statement, which is used to return a value from the function. The syntax of the return statement is

```
return value
```

As an example of this, consider the following code snippet that defines the function '`sum`' to return a sum of its two arguments:

```
func sum(number x, number y) returns number
do
        return x + y
done
```

The `returns` part in the function declaration is optional. A declaration lacking it defines a *procedure*, or *void function*, i.e. a function that is not supposed to return any value. Such

functions cannot be used in expressions, instead they are used as statements (see Section 4.19 [Statements], page 92). The following example shows a function that emits a customized temporary failure notice:

```
func stdtf()
do
  tempfail 451 4.3.5 "Try again later"
done
```

A function may have several names. An alternative name (or *alias*) can be assigned to a function by using `alias` keyword, placed after *param-decl* part, for example:

```
func foo()
alias bar
returns string
do
  ...
done
```

After this declaration, both `foo()` and `bar()` will refer to the same function.

The number of function aliases is unlimited. The following fragment declares a function having three names:

```
func foo()
alias bar
alias baz
returns string
do
  ...
done
```

Although this feature is rarely needed, there are sometimes cases when it may be necessary.

A variable declared within a function becomes a local variable to this function. Its lexical scope ends with the terminating `done` statement.

Parameters, local variables and global variables are using separate namespaces, so a parameter name can coincide with the name of a global, in which case a parameter is said to *shadow* the global. All references to its name will refer to the parameter, until the end of its scope is reached, where the global one becomes visible again. Consider the following example:

```
number x

func foo(string x)
do
  echo "foo: %x"
done

prog envfrom
do
  set x "Global"
  foo("Local")
  echo x
done
```

Running `mailfromd --test` with this configuration will display:

```
foo: Local
Global
```

4.16.1 Some Useful Functions

To illustrate the concept of user-defined functions, this subsection shows the definitions of some of the library functions shipped with `mailfromd`[5]. These functions are contained in modules installed along with the `mailfromd` binary. To use any of them in your code, require the appropriate module as described in Section 4.24.3 [import], page 109, e.g. to use the `revip` function, do `require 'revip'`.

Functions and their definitions:

1. `revip`

   The function `revip`, that was used in releases of `mailfromd` up to 9.0 (see [revip], page 288) was implemented as follows:

   ```
   func revip(string ip) returns string
   do
     return inet_ntoa(ntohl(inet_aton(ip)))
   done
   ```

   Previously it was implemented using regular expressions. Below we include this variant as well, as an illustration for the use of regular expressions:

[5] Notice that these are intended for educational purposes and do not necessarily coincide with the actual definitions of these functions in Mailfromd version 9.2.

```
#pragma regex push +extended
func revip(string ip) returns string
do
  if ip matches '([0-9]+)\.([0-9]+)\.([0-9]+)\.([0-9]+)'
    return "\4.\3.\2.\1"
  fi
  return ip
done
#pragma regex pop
```

2. `strip_domain_part`

   This function returns at most *n* last components of the domain name *domain* (see [strip_domain_part], page 133).

```
#pragma regex push +extended

func strip_domain_part(string domain, number n) returns string
do
  if n > 0 and
    domain matches '.*((\.[^.]+){' . $2 . '})'
    return substring(\1, 1, -1)
  else
    return domain
  fi
done
#pragma regex pop
```

3. `valid_domain`

   See [valid_domain], page 190, for a description of this function. Its definition follows:

```
require dns

func valid_domain(string domain) returns number
do
  return not (resolve(domain) = "0" and not hasmx(domain))
done
```

4. `match_dnsbl`

   The function `match_dnsbl` (see [match_dnsbl], page 194) is defined as follows:

```
require dns
require match_cidr
#pragma regex push +extended

func match_dnsbl(string address, string zone, string range)
    returns number
do
  string rbl_ip
```

```
  if range = 'ANY'
    set rbl_ip '127.0.0.0/8'
  else
    set rbl_ip range
    if not range matches '^([0-9]{1,3}\.){3}[0-9]{1,3}$'
      return 0
    fi
  fi

  if not (address matches '^([0-9]{1,3}\.){3}[0-9]{1,3}$'
          and address != range)
    return 0
  fi

  if address matches
        '^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$'
    if match_cidr (resolve ("\4.\3.\2.\1", zone), rbl_ip)
      return 1
    else
      return 0
    fi
  fi
  # never reached
done
```

4.17 Expressions

Expressions are language constructs, that evaluate to a value, that can subsequently be echoed, tested in a conditional statement, assigned to a variable or passed to a function.

4.17.1 Constant Expressions

Literals and numbers are *constant expressions*. They evaluate to string and numeric types.

4.17.2 Function Calls

A function call is an expression. Its type is the return type of the function.

4.17.3 Concatenation

Concatenation operator is '`.`' (a dot). For example, if `$f` is '`smith`', and `$client_addr` is '`10.10.1.1`', then:

```
$f . "-" . $client_addr ⇒ "smith-10.10.1.1"
```

Any two adjacent literal strings are concatenated, producing a new string, e.g.

```
"GNU's" " not " "UNIX" ⇒ "GNU's not UNIX"
```

4.17.4 Arithmetic Operations

The filter script language offers the common arithmetic operators: '`+`', '`-`', '`*`' and '`/`'. In addition, the '`%`' is a *modulo* operator, i.e. it computes the remainder of division of its operands.

All of them follow usual precedence rules and work as you would expect them to.

4.17.5 Bitwise shifts

The '`<<`' represents a *bitwise shift left* operation, which shifts the binary representation of the operand on its left by the number of bits given by the operand on its right.

Similarly, the '`>>`' represents a *bitwise shift right*.

4.17.6 Relational Expressions

Relational expressions are:

Expression	**Result**
x `<` *y*	True if *x* is less than *y*.
x `<=` *y*	True if *x* is less than or equal to *y*.
x `>` *y*	True if *x* is greater than *y*.
x `>=` *y*	True if *x* is greater than or equal to *y*.
x `=` *y*	True if *x* is equal to *y*.
x `!=` *y*	True if *x* is not equal to *y*.

Table 4.5: Relational Expressions

The relational expressions apply to string as well as to numbers. When a relational operation applies to strings, case-sensitive comparison is used, e.g.:

```
"String" = "string" ⇒ False
"String" < "string" ⇒ True
```

4.17.7 Special Comparisons

In addition to the traditional relational operators, described above, `mailfromd` provides two operators for regular expression matching:

Expression	**Result**
x `matches` *y*	True if the string *x* matches the regexp denoted by *y*.
x `fnmatches` *y*	True if the string *x* matches the globbing pattern denoted by *y*.

Table 4.6: Regular Expression Matching

The type of the regular expression used by `matches` operator is controlled by `#pragma regex` (see [pragma regex], page 56). For example:

```
$f ⇒ "gray@gnu.org.ua"
$f matches '.*@gnu\.org\.ua' ⇒ true
$f matches '.*@GNU\.ORG\.UA' ⇒ false
#pragma regex +icase
$f matches '.*@GNU\.ORG\.UA' ⇒ true
```

The `fnmatches` operator compares its left-hand operand with a globbing pattern (see *glob(7)*) given as its right-hand side operand. For example:

```
$f ⇒ "gray@gnu.org.ua"
$f fnmatches "*ua" ⇒ true
$f fnmatches "*org" ⇒ false
$f fnmatches "*org*" ⇒ true
```

Both operators have a special form, for *'MX' pattern matching*. The expression:

```
x mx matches y
```

is evaluated as follows: first, the expression *x* is analyzed and, if it is an email address, its domain part is selected. If it is not, its value is used verbatim. Then the list of 'MX's for this domain is looked up. Each of 'MX' names is then compared with the regular expression *y*. If any of the names matches, the expression returns true. Otherwise, its result is false.

Similarly, the expression:

```
x mx fnmatches y
```

returns true only if any of the 'MX's for (domain or email) *x* match the globbing pattern *y*.

Both `mx matches` and `mx fnmatches` can signal the following exceptions: `e_temp_failure`, `e_failure`.

The value of any parenthesized subexpression occurring within the right-hand side argument to `matches` or `mx matches` can be referenced using the notation '`\d`', where *d* is the ordinal number of the subexpression (subexpressions are numbered from left to right, starting at 1). This notation is allowed in the program text as well as within double-quoted strings and here-documents, for example:

```
if $f matches '.*@\(.*\)\.gnu\.org\.ua'
  set message "Your host name is \1;"
fi
```

Remember that the grouping symbols are '`\(`' and '`\)`' for basic regular expressions, and '`(`' and '`)`' for extended regular expressions. Also make sure you properly escape all special characters (backslashes in particular) in double-quoted strings, or use single-quoted strings to avoid having to do so (see [singe-vs-double], page 60, for a comparison of the two forms).

4.17.8 Boolean Expressions

A *boolean expression* is a combination of relational or matching expressions using the boolean operators `and`, `or` and `not`, and, eventually, parentheses to control nesting:

Expression	**Result**
x `and` *y*	True only if both *x* and *y* are true.
x `or` *y*	True if any of *x* or *y* is true.
`not` *x*	True if *x* is false.

table 4.1: Boolean Operators

Binary boolean expressions are computed using *shortcut evaluation*:

x `and` *y* If *x* ⇒ `false`, the result is `false` and *y* is not evaluated.

x `or` *y* If *x* ⇒ `true`, the result is `true` and *y* is not evaluated.

4.17.9 Operator Precedence

Operator *precedence* is an abstract value associated with each language operator, that determines the order in which operators are executed when they appear together within a single expression. Operators with higher precedence are executed first. For example, '`*`' has a higher precedence than '`+`', therefore the expression `a + b * c` is evaluated in the following order: first `b` is multiplied by `c`, then `a` is added to the product.

When operators of equal precedence are used together they are evaluated from left to right (i.e., they are *left-associative*), except for comparison operators, which are non-associative (these are explicitly marked as such in the table below). This means that you cannot write:

```
if 5 <= x <= 10
```

Instead, you should write:

```
if 5 <= x and x <= 10
```

The precedence of the `mailfromd` operators where selected so as to match that used in most programming languages.[6]

The following table lists all operators in order of decreasing precedence:

`(...)` Grouping

`$ %` `Sendmail` macros and `mailfromd` variables

`* /` Multiplication, division

`+ -` Addition, subtraction

`<< >>` Bitwise shift left and right

`< <= >= >` Relational operators (non-associative)

`= != matches fnmatches`
Equality and special comparison (non-associative)

`&` Logical (bitwise) AND

`^` Logical (bitwise) XOR

`|` Logical (bitwise) OR

`not` Boolean negation

`and` Logical '`and`'.

`or` Logical '`or`'

`.` String concatenation

4.17.10 Type Casting

When two operands on each side of a binary expression have different type, `mailfromd` evaluator coerces them to a common type. This is known as *implicit type casting*. The rules for implicit type casting are:

1. Both arguments to an arithmetical operation are cast to numeric type.
2. Both arguments to the concatenation operation are cast to string.

---

[6] The only exception is '`not`', whose precedence in MFL is much lower than usual (in most programming languages it has the same precedence as unary '`-`'). This allows to write conditional expressions in more understandable manner. Consider the following condition:

```
if not x < 2 and y = 3
```

It is understood as "if `x` is not less than 2 and `y` equals 3", whereas with the usual precedence for '`not`' it would have meant "if negated `x` is less than 2 and `y` equals 3".

3. Both arguments to 'match' or 'fnmatch' function are cast to string.
4. The argument of the unary negation (arithmetical or boolean) is cast to numeric.
5. Otherwise the right-hand side argument is cast to the type of the left-hand side argument.

The construct for explicit type cast is:

```
type(expr)
```

where *type* is the name of the type to coerce *expr* to. For example:

```
string(2 + 4*8) ⇒ "34"
```

A special case of type casting is cast to `void`. It is used to ignore return value of a function call between the braces, e.g.:

```
void(dlcall(libh, "extlog", "s", text))
```

4.18 Variable and Constant Shadowing

When any two named entities happen to have the same name we say that a *name clash* occurs. The handling of name clashes depends on types of the entities involved in it.

function – any

A name of a constant or variable can coincide with that of a function, it does not produce any warnings or errors because functions, variables and constants use different namespaces. For example, the following code is correct:

```
const a 4

func a()
do
  echo a
done
```

When executed, it prints '4'.

function – function, handler – function, and function – handler

Redefinition of a function or using a predefined handler name (see Section 4.14 [Handlers], page 70) as a function name results in a fatal error. For example, compiling this code:

```
func a()
do
  echo "1"
done

func a()
do
  echo "2"
done
```

causes the following error message:

```
mailfromd: sample.mfl:9: syntax error, unexpected
FUNCTION_PROC, expecting IDENTIFIER
```

handler – variable

A variable name can coincide with a handler name. For example, the following code is perfectly OK:

```
string envfrom "M"
prog envfrom
do
        echo envfrom
done
```

handler – handler

If two handlers with the same name are defined, the definition that appears further in the source text replaces the previous one. A warning message is issued, indicating locations of both definitions, e.g.:

```
mailfromd: sample.mfl:116: Warning: Redefinition of handler
`envfrom'
mailfromd: sample.mfl:34: Warning: This is the location of the
previous definition
```

variable – variable

Defining a variable having the same name as an already defined one results in a warning message being displayed. The compilation succeeds. The second variable *shadows* the first, that is any subsequent references to the variable name will refer to the second variable. For example:

```
string x "Text"
number x 1

prog envfrom
do
  echo x
done
```

Compiling this code results in the following diagnostics:

```
mailfromd: sample.mfl:4: Redeclaring `x' as different data type
mailfromd: sample.mfl:2: This is the location of the previous
definition
```

Executing it prints '`1`', i.e. the value of the last definition of `x`.

The scope of the shadowing depends on storage classes of the two variables. If both of them have external storage class (i.e. are global ones), the shadowing remains in effect until the end of input. In other words, the previous definition of the variable is effectively forgotten.

If the previous definition is a global, and the shadowing definition is an automatic variable or a function parameter, the scope of this shadowing ends with the scope of the second variable, after which the previous definition (global) becomes visible again. Consider the following code:

```
set x "initial"
```

```
func foo(string x) returns string
do
  return x
done

prog envfrom
do
  echo foo("param")
  echo x
done
```

Its compilation produces the following warning:

```
mailfromd: sample.mfl:3: Warning: Parameter `x' is shadowing a global
```

When executed, it produces the following output:

```
param
initial
State envfrom: continue
```

variable – constant

If a constant is defined which has the same name as a previously defined variable (the constant *shadows* the variable), the compiler prints the following diagnostic message:

```
file:line: Warning: Constant name `name' clashes with a variable name
file:line: Warning: This is the location of the previous definition
```

A similar diagnostics is issued if a variable is defined whose name coincides with a previously defined constant (the variable shadows the constant).

In any case, any subsequent notation %*name* refers to the last defined symbol, be it variable or constant.

Notice, that shadowing occurs only when using %*name* notation. Referring to the constant using its name without '`%`' allows to avoid shadowing effects.

If a variable shadows a constant, the scope of the shadowing depends on the storage class of the variable. For automatic variables and function parameters, it ends with the final `done` closing the function. For global variables, it lasts up to the end of input.

For example, consider the following code:

```
const a 4

func foo(string a)
do
  echo a
done

prog envfrom
do
  foo(10)
  echo a
```

```
done
```

When run, it produces the following output:

```
$ mailfromd --test sample.mfl
mailfromd: sample.mfl:3: Warning: Variable name `a' clashes with a
constant name
mailfromd: sample.mfl:1: Warning: This is the location of the previous
definition
10
4
State envfrom: continue
```

constant – constant

Redefining a constant produces a warning message. The latter definition shadows the former. Shadowing remains in effect until the end of input.

4.19 Statements

Statements are language constructs, that, unlike expressions, do not return any value. Statements execute some actions, such as assigning a value to a variable, or serve to control the execution flow in the program.

4.19.1 Action Statements

An *action* statement instructs `mailfromd` to perform a certain action over the message being processed. There are two kinds of actions: return actions and header manipulation actions.

Reply Actions

Reply actions tell `Sendmail` to return given response code to the remote party. There are five such actions:

`accept` Return an `accept` reply. The remote party will continue transmitting its message.

`reject` *code excode message-expr*
`reject (`*code-expr*`,` *excode-expr*`,` *message-expr*`)`
Return a `reject` reply. The remote party will have to cancel transmitting its message. The three arguments are optional, their usage is described below.

`tempfail` *code excode message*
`tempfail (`*code-expr*`,` *excode-expr*`,` *message-expr*`)`
Return a '`temporary failure`' reply. The remote party can retry to send its message later. The three arguments are optional, their usage is described below.

`discard` Instructs `Sendmail` to accept the message and silently discard it without delivering it to any recipient.

`continue` Stops the current handler and instructs `Sendmail` to continue processing of the message.

Two actions, `reject` and `tempfail` can take up to three optional parameters. There are two forms of supplying these parameters.

In the first form, called *literal* or *traditional* notation, the arguments are supplied as additional words after the action name, and are separated by whitespace. The first argument is a three-digit RFC 2821 reply code. It must begin with '`5`' for `reject` and with '`4`' for `tempfail`. If two arguments are supplied, the second argument must be either an *extended reply code* (RFC 1893/2034) or a textual string to be returned along with the SMTP reply. Finally, if all three arguments are supplied, then the second one must be an extended reply code and the third one must give the textual string. The following examples illustrate the possible ways of using the `reject` statement:

```
reject
reject 503
reject 503 5.0.0
reject 503 "Need HELO command"
reject 503 5.0.0 "Need HELO command"
```

Used without arguments, `reject` is equivalent to

```
reject 550
```

and `tempfail` to

```
tempfail 451
```

In literal notation, the values of code and extendended code (if supplied) must be literal strings. The third argument (textual message) can be either a literal string or MFL expression that evaluates to string.

The second form of supplying arguments is called *functional* notation, because it resembles the function syntax. When used in this form, the action word is followed by a parenthesized group of exactly three arguments, separated by commas. Each argument is a MFL expression. The meaning and ordering of the arguments is the same as in literal form. Any or all of these three arguments may be absent, in which case the corresponding default value will be used[7]. To illustrate this, here are the statements from the previous example, written in functional notation:

```
reject(,,)
reject(503,,)
reject(503, 5.0.0)
reject(503, , "Need HELO command")
reject(503, 5.0.0, "Need HELO command")
```

Notice that there is an important difference between the two notations. The functional notation allows to compute both reply codes at run time, e.g.:

```
reject(500 + dig2*10 + dig3, "5.%edig2.%edig2")
```

Header Actions

Header manipulation actions provide basic means to add, delete or modify the message RFC 2822 headers.

`add` *`name string`*

Add the header *name* with the value *string*. E.g.:

```
add "X-Seen-By" "Mailfromd 9.2"
```

[7] The default value for code is 550 for `reject` and 451 for `tempfail`. The remaining two arguments default to empty strings.

(notice argument quoting)

`replace` *`name string`*
: The same as `add`, but if the header *name* already exists, it will be removed first, for example:

```
replace "X-Last-Processor" "Mailfromd 9.2"
```

`delete` *`name`*
: Delete the header named *name*:

```
delete "X-Envelope-Date"
```

These actions impose some restrictions. First of all, their first argument must be a literal string (not a variable or expression). Secondly, there is no way to select a particular header instance to delete or replace, which may be necessary to properly handle multiple headers (e.g. '`Received`'). For more elaborate ways of header modifications, see Section 5.11 [Header modification functions], page 148.

4.19.2 Variable Assignments

An *assignment* is a special statement that assigns a value to the variable. It has the following syntax:

```
set name value
```

where *name* is the variable name and *value* is the value to be assigned to it.

Assignment statements can appear in any part of a filter program. If an assignment occurs outside of function or handler definition, the *value* must be a literal value (see Section 4.8 [Literals], page 59). If it occurs within a function or handler definition, *value* can be any valid `mailfromd` expression (see Section 4.17 [Expressions], page 85). In this case, the expression will be evaluated and its value will be assigned to the variable. For example:

```
set delay 150

prog envfrom
do
  set delay delay * 2
  ...
done
```

4.19.3 The `pass` statement

The `pass` statement has no effect. It is used in places where no statement is needed, but the language syntax requires one:

```
on poll $f do
when success:
  pass
when not_found or failure:
  reject 550
done
```

4.19.4 The `echo` statement

The `echo` statement concatenates all its arguments into a single string and sends it to the `syslog` using the priority '`info`'. It is useful for debugging your script, in conjunction with built-in constants (see Section 4.11.1 [Built-in constants], page 63), for example:

```
func foo(number x)
do
  echo "%__file__:%__line__: foo called with arg %x"
  ...
done
```

4.20 Conditional Statements

Conditional expressions, or conditionals for short, test some conditions and alter the control flow depending on the result. There are two kinds of conditional statements: *if-else* branches and *switch* statements.

The syntax of an *if-else* branching construct is:

```
if condition then-body [else else-body] fi
```

Here, *condition* is an expression that governs control flow within the statement. Both *then-body* and *else-body* are lists of `mailfromd` statements. If *condition* is true, *then-body* is executed, if it is false, *else-body* is executed. The '`else`' part of the statement is optional. The condition is considered false if it evaluates to zero, otherwise it is considered true. For example:

```
if $f = ""
  accept
else
  reject
fi
```

This will accept the message if the value of the `Sendmail` macro `$f` is an empty string, and reject it otherwise. Both *then-body* and *else-body* can be compound statements including other `if` statements. Nesting level of conditional statements is not limited.

To facilitate writing complex conditional statements, the `elif` keyword can be used to introduce alternative conditions, for example:

```
if $f = ""
  accept
elif $f = "root"
  echo "Mail from root!"
else
  reject
fi
```

Another type of branching instruction is `switch` statement:

```
switch condition
do
case x1 [or x2 ...]:
  stmt1
case y1 [or y2 ...]:
  stmt2
  .
  .
  .
[default:
  stmt]
done
```

Here, *x1*, *x2*, *y1*, *y2* are literal expressions; *stmt1*, *stmt2* and *stmt* are arbitrary `mailfromd` statements (possibly compound); *condition* is the controlling expression. The vertical dotted row represent another eventual '`case`' branches.

This statement is executed as follows: the *condition* expression is evaluated and if its value equals *x1* or *x2* (or any other *x* from the first `case`), then *stmt1* is executed. Otherwise, if *condition* evaluates to *y1* or *y2* (or any other *y* from the second `case`), then *stmt2* is executed. Other `case` branches are tried in turn. If none of them matches, *stmt* (called the *default branch*) is executed.

There can be as many `case` branches as you wish. The `default` branch is optional. There can be at most one `default` branch.

An example of `switch` statement follows:

```
switch x
do
case 1 or 3:
  add "X-Branch" "1"
  accept
case 2 or 4 or 6:
  add "X-Branch" "2"
default:
  reject
done
```

If the value of `mailfromd` variable `x` is 2 or 3, it will accept the message immediately, and add a '`X-Branch: 1`' header to it. If `x` equals 2 or 4 or 6, this code will add '`X-Branch: 2`' header to the message and will continue processing it. Otherwise, it will reject the message.

The controlling condition of a `switch` statement may evaluate to numeric or string type. The type of the condition governs the type of comparisons used in `case` branches: for numeric types, numeric equality will be used, whereas for string types, string equality is used.

4.21 Loop Statements

The loop statement allows for repeated execution of a block of code, controlled by some conditional expression. It has the following form:

```
loop [label]
     [for stmt1] [,while expr1] [,stmt2]
do
  stmt3
done [while expr2]
```

where *stmt1*, *stmt2*, and *stmt3* are statement lists, *expr1* and *expr2* are expressions.

The control flow is as follows:

1. If *stmt1* is specified, execute it.
2. Evaluate *expr1*. If it is zero, go to 6. Otherwise, continue.
3. Execute *stmt3*.
4. If *stmt2* is supplied, execute it.
5. If *expr2* is given, evaluate it. If it is zero, go to 6. Otherwise, go to 2.
6. End.

Thus, *stmt3* is executed until either *expr1* or *expr2* yield a zero value.

The *loop body* – *stmt3* – can contain special statements:

`break [label]`
: Terminates the loop immediately. Control passes to '`6`' (End) in the formal definition above. If *label* is supplied, the statement terminates the loop statement marked with that label. This allows to break from nested loops.

  It is similar to `break` statement in C or shell.

`next [label]`
: Initiates next iteration of the loop. Control passes to '`4`' in the formal definition above. If *label* is supplied, the statement starts next iteration of the loop statement marked with that label. This allows to request next iteration of an upper-level loop from a nested loop statement.

The `loop` statement can be used to create iterative statements of arbitrary complexity. Let's illustrate it in comparison with C.

The statement:

```
loop
do
  stmt-list
done
```

creates an infinite loop. The only way to exit from such a loop is to call `break` (or `return`, if used within a function), somewhere in *stmt-list*.

The following statement is equivalent to `while (expr1) stmt-list` in C:

```
loop while expr
do
  stmt-list
done
```

The C construct `for (expr1; expr2; expr3)` is written in MFL as follows:

```
loop for stmt1, while expr2, stmt2
do
  stmt3
done
```

For example, to repeat *stmt3* 10 times:

```
loop for set i 0, while i < 10, set i i + 1
do
  stmt3
done
```

Finally, the C 'do' loop is implemented as follows:

```
loop
do
  stmt-list
done while expr
```

As a real-life example of a loop statement, let's consider the implementation of function `ptr_validate`, which takes a single argument *ipstr*, and checks its validity using the following algorithm:

Perform a DNS reverse-mapping for *ipstr*, looking up the corresponding `PTR` record in '`in-addr.arpa`'. For each record returned, look up its IP addresses (A records). If *ipstr* is among the returned IP addresses, return 1 (`true`), otherwise return 0 (`false`).

The implementation of this function in MFL is:

```
#pragma regex push +extended

func ptr_validate(string ipstr) returns number
do
  loop for string names dns_getname(ipstr) . " "
           number i index(names, " "),
       while i != -1,
       set names substr(names, i + 1)
       set i index(names, " ")
  do
    loop for string addrs dns_getaddr(substr(names, 0, i)) . " "
             number j index(addrs, " "),
         while j != -1,
         set addrs substr(addrs, j + 1)
         set j index(addrs, " ")
    do
      if ipstr == substr(addrs, 0, j)
        return 1
      fi
    done
  done
  return 0
done
```

4.22 Exceptional Conditions

When the running program encounters a condition it is not able to handle, it signals an *exception*. To illustrate the concept, let's consider the execution of the following code fragment:

```
if primitive_hasmx(domainpart($f))
  accept
fi
```

The function `primitive_hasmx` (see [primitive_hasmx], page 169) tests whether the domain name given as its argument has any '`MX`' records. It should return a boolean value. However, when querying the Domain Name System, it may fail to get a definite result. For example, the DNS server can be down or temporary unavailable. In other words, `primitive_hasmx` can be in a situation when, instead of returning '`yes`' or '`no`', it has to return '`don't know`'. It has no way of doing so, therefore it signals an *exception*.

Each exception is identified by *exception type*, an integer number associated with it.

4.22.1 Built-in Exceptions

The first 22 exception numbers are reserved for *built-in exceptions*. These are declared in module `status.mfl`. The following table summarizes all built-in exception types implemented by `mailfromd` version 9.2. Exceptions are listed in lexicographic order.

`e_badmmq` [Exception]
: The called function cannot finish its task because an incompatible message modification function was called at some point before it. For details, [MMQ and dkim_sign], page 205.

`e_dbfailure` [Exception]
: General database failure. For example, the database cannot be opened. This exception can be signaled by any function that queries any DBM database.

`e_divzero` [Exception]
: Division by zero.

`e_exists` [Exception]
: This exception is emitted by `dbinsert` built-in if the requested key is already present in the database (see Section 5.26 [Database functions], page 175).

`e_eof` [Exception]
: Function reached end of file while reading. See Section 5.7 [I/O functions], page 137, for a description of functions that can signal this exception.

`e_failure` [Exception]
: A general failure has occurred. In particular, this exception is signaled by DNS lookup functions when any permanent failure occurs. This exception can be signaled by any DNS-related function (`hasmx`, `poll`, etc.) or operation (`mx matches`).

`e_format` [Exception]
: Invalid input format. This exception is signaled if input data to a function are improperly formatted. In version 9.2 it is signaled by `message_burst` function if its

input message is not formatted according to RFC 934. See Section 5.19.4 [Message digest functions], page 160.

`e_ilseq` [Exception]
Illegal byte sequence. Signaled when a string cannot be converted between character sets because a sequence of bytes was encountered that is not defined for the source character set or cannot be represented in the destination character set.

See [MIME decoding], page 159, for details.

`e_inval` [Exception]
Arguments supplied to a function are invalid.

`e_invcidr` [Exception]
Invalid CIDR notation. This is signaled by `match_cidr` function when its second argument is not a valid CIDR.

`e_invip` [Exception]
Invalid IP address. This is signaled by `match_cidr` function when its first argument is not a valid IP address.

`e_invtime` [Exception]
Invalid time interval specification. It is signaled by `interval` function if its argument is not a valid time interval (see [time interval specification], page 220).

`e_io` [Exception]
An error occurred during the input-output operation. See Section 5.7 [I/O functions], page 137, for a description of functions that can signal this exception.

`e_macroundef` [Exception]
A Sendmail macro is undefined.

`e_not_found` [Exception]
Required entity is not found. It is raised, for example, by `message_find_header`, when the requested header is not present in the message and by DNS resolver functions when unable to resolve host name or IP address.

`e_range` [Exception]
The supplied argument is outside the allowed range. This is signalled, for example, by `substring` function (see [substring], page 132).

`e_regcomp` [Exception]
Regular expression cannot be compiled. This can happen when a regular expression (a right-hand argument of a `matches` operator) is built at the runtime and the produced string is an invalid regex.

`e_ston_conv` [Exception]
String-to-number conversion failed. This can be signaled when a string is used in numeric context which cannot be converted to the numeric data type. For example:

```
set x "10a"
set y x / 2
```

In this code fragment, line 2 will raise the `e_ston_conv` exception, since '`10a`' cannot be converted to a number.

`e_success` [Exception]
: This is not an exception in the strict sense of the word, but a constant indicating success.

`e_temp_failure` [Exception]
: A temporary failure has occurred. This can be signaled by DNS-related functions or operations.

`e_too_many` [Exception]
: Raised by various DNS functions when they encounter a long chain of CNAME records when trying to resolve a hostname. See [CNAME chains], page 223.

`e_url` [Exception]
: The supplied URL is invalid. See Section 5.31 [Interfaces to Third-Party Programs], page 185.

4.22.2 User-defined Exceptions

You can define your own exception types using the `dclex` statement:

```
dclex type
```

In this statement, *type* must be a valid MFL identifier, not used for another constant (see Section 4.11 [Constants], page 62). The `dclex` statement defines a new exception identified by the constant *type* and allocates a new exception number for it.

The *type* can subsequently be used in `throw` and `catch` statements, for example:

```
dclex myrange

number fact(number val)
  returns number
do
  if val < 0
    throw myrange "fact argument is out of range"
  fi
  ...
done
```

4.22.3 Exception Handling

Normally when an exception is signalled, the program execution is terminated and the MTA is returned a `tempfail` status. Additional information regarding the exception is then output to the logging channel (see Section 3.19 [Logging and Debugging], page 43). However, the user can intercept any exception by installing his own exception-handling routines.

An exception-handling routine is introduced by a *try–catch* statement, which has the following syntax:

```
try
do
  stmtlist
done
catch exception-list
do
  handler-body
done
```

where *stmtlist* and *handler-body* are sequences of MFL statements and *exception-list* is the list of exception types, separated by the word `or`. A special *exception-list* '`*`' is allowed and means all exceptions.

This construct works as follows. First, the statements from *stmtlist* are executed. If the execution finishes successfully, control is passed to the first statement after the '`catch`' block. Otherwise, if an exception is signalled and this exception is listed in *exception-list*, the execution is passed to the *handler-body*. If the exception is not listed in *exception-list*, it is handled as usual.

The following example shows a '`try--catch`' construct used for handling eventual exceptions, signalled by `primitive_hasmx`.

```
try
do
  if primitive_hasmx(domainpart($f))
    accept
  else
    reject
  fi
done
catch e_failure or e_temp_failure
do
  echo "primitive_hasmx failed"
  continue
done
```

The '`try--catch`' statement can appear anywhere inside a function or a handler, but it cannot appear outside of them. It can also be nested within another '`try--catch`', in either of its parts. Upon exit from a function or milter handler, all exceptions are restored to the state they had when it has been entered.

A `catch` block can also be used alone, without preceding `try` part. Such a construct is called a *standalone catch*. It is mostly useful for setting global exception handlers in a `begin` statement (see Section 4.15.1 [begin/end], page 76). When used within a usual function or handler, the exception handlers set by a standalone catch remain in force until either another standalone catch appears further in the same function or handler, or an end of the function is encountered, whichever occurs first.

A standalone catch defined within a function must return from it by executing `return` statement. If it does not do that explicitly, the default value of 1 is returned. A standalone catch defined within a milter handler must end execution with any of the following actions: `accept`, `continue`, `discard`, `reject`, `tempfail`. By default, `continue` is used.

It is not recommended to mix '`try--catch`' constructs and standalone catches. If a standalone catch appears within a '`try--catch`' statement, its scope of visibility is undefined.

Upon entry to a *handler-body*, two implicit positional arguments are defined, which can be referenced in *handler-body* as `$1` and `$2`[8]. The first argument gives the numeric code of the exception that has occurred. The second argument is a textual string containing a human-readable description of the exception.

The following is an improved version of the previous example, which uses these parameters to supply more information about the failure:

```
try
do
  if primitive_hasmx(domainpart($f))
    accept
  else
    reject
  fi
done
catch e_failure or e_temp_failure
do
  echo "Caught exception $1: $2"
  continue
done
```

The following example defines the function `hasmx` that returns true if the domain part of its argument has any '`MX`' records, and false if it does not or if an exception occurs[9].

```
func hasmx (string s)
  returns number
do
  try
  do
    return primitive_hasmx(domainpart(s))
  done
  catch *
  do
    return 0
  done
done
```

The same function can written using standalone `catch`:

```
func hasmx (string s)
  returns number
do
  catch *
```

[8] As of `mailfromd` version 9.2, there is also a third implicit argument, which holds the value of program counter where the exception occurred. Currently it is considered to be an implementation artifact. Filter writers are discouraged from relying on it.

[9] This function is part of the `mailfromd` library, See [hasmx], page 169.

```
    do
      return 0
    done
    return primitive_hasmx(domainpart(s))
  done
```

All variables remain visible within `catch` body, with the exception of positional arguments of the enclosing handler. To access positional arguments of a handler from the `catch` body, assign them to local variables prior to the '`try--catch`' construct, e.g.:

```
prog header
do
  string hname $1
  string hvalue $2
  try
  do
    ...
  done
  catch *
  do
    echo "Exception $1 while processing header %hname: %hvalue"
    echo $2
    tempfail
  done
```

You can also generate (or *raise*) exceptions explicitly in the code, using `throw` statement:

```
throw excode descr
```

The arguments correspond exactly to the positional parameters of the `catch` statement: *excode* gives the numeric code of the exception, *descr* gives its textual description. This statement can be used in complex scripts to create non-local exits from deeply nested statements.

Notice, that the the *excode* argument must be an immediate value: an exception identifier (either a built-in one or one declared previously using a `dclex` statement).

4.23 Sender Verification Tests

The filter script language provides a wide variety of functions for sender address verification or *polling*, for short. These functions, which were described in Section 5.21 [SMTP Callout functions], page 161, can be used to implement any sender verification method. The additional data that can be needed is normally supplied by two global variables: `ehlo_domain`, keeping the default domain for the `EHLO` command, and `mailfrom_address`, which stores the sender address for probe messages (see Section 4.12.1 [Predefined variables], page 66).

For example, a simplest way to implement standard polling would be:

```
prog envfrom
do
  if stdpoll($1, ehlo_domain, mailfrom_address) == 0
    accept
  else
```

```
    reject 550 5.1.0 "Sender validity not confirmed"
  fi
done
```

However, this does not take into account exceptions that `stdpoll` can signal. To handle them, one will have to use `catch`, for example thus:

```
require status

prog envfrom
do
  try
  do
    if stdpoll($1, ehlo_domain, mailfrom_address) == 0
      accept
    else
      reject 550 5.1.0 "Sender validity not confirmed"
    fi
  done
  catch e_failure or e_temp_failure
  do
    switch $1
    do
    case failure:
      reject 550 5.1.0 "Sender validity not confirmed"
    case temp_failure:
      tempfail 450 4.1.0 "Try again later"
    done
  done
done
```

If polls are used often, one can define a wrapper function, and use it instead. The following example illustrates this approach:

```
func poll_wrapper(string email) returns number
do
  catch e_failure or e_temp_failure
  do
    return email
  done
  return stdpoll(email, ehlo_domain, mailfrom_address)
done

prog envfrom
do
  switch poll_wrapper($f)
  do
  case success:
    accept
  case not_found or failure:
    reject 550 5.1.0 "Sender validity not confirmed"
  case temp_failure:
    tempfail 450 4.1.0 "Try again later"
  done
done
```

Figure 4.1: Building Poll Wrappers

Notice the way `envfrom` handles `success` and `not_found`, which are not exceptions in the strict sense of the word.

The above paradigm is so common that `mailfromd` provides a special language construct to simplify it: the `on` statement. Instead of manually writing the wrapper function and using it as a `switch` condition, you can rewrite the above example as:

```
prog envfrom
do
  on stdpoll($1, ehlo_domain, mailfrom_address)
  do
  when success:
    accept
  when not_found or failure:
    reject 550 5.1.0 "Sender validity not confirmed"
  when temp_failure:
    tempfail 450 4.1.0 "Try again later"
  done
done
```

Figure 4.2: Standard poll example

As you see the statement is pretty similar to `switch`. The major syntactic difference is the use of the keyword `when` to introduce conditional branches.

General syntax of the `on` statement is:

```
on condition
do
  when x1 [or x2 ...]:
    stmt1
  when y1 [or y2 ...]:
    stmt2
    .
    .
    .
done
```

The *condition* is either a function call or a special `poll` statement (see below). The values used in `when` branches are normally symbolic exception names (see [exception names], page 99).

When the compiler processes the `on` statement it does the following:

1. Builds a unique wrapper function, similar to that described in Figure 4.1; The name of the function is constructed from the *condition* function name and an unsigned number, called *exception mask*, that is unique for each combination of exceptions used in `when` branches; To avoid name clashes with the user-defined functions, the wrapper name begins and ends with '`$`' which normally is not allowed in the identifiers;
2. Translates the `on` body to the corresponding `switch` statement;

A special form of the *condition* is `poll` keyword, whose syntax is:

```
poll [for] email
     [host host]
     [from domain]
     [as email]
```

The order of particular keywords in the `poll` statement is arbitrary, for example `as` *`email`* can appear before *email* as well as after it.

The simplest form, `poll` *`email`*, performs the standard sender verification of email address *email*. It is translated to the following function call:

```
stdpoll(email, ehlo_domain, mailfrom_address)
```

The construct `poll` *`email`* `host` *`host`*, runs the strict sender verification of address *email* on the given host. It is translated to the following call:

```
strictpoll(host, email, ehlo_domain, mailfrom_address)
```

Other keywords of the `poll` statement modify these two basic forms. The `as` keyword introduces the email address to be used in the SMTP `MAIL FROM` command, instead of `mailfrom_address`. The `from` keyword sets the domain name to be used in `EHLO` command. So, for example the following construct:

```
poll email host host from domain as addr
```

is translated to

```
strictpoll(host, email, domain, addr)
```

To summarize the above, the code described in Figure 4.2 can be written as:

```
prog envfrom
do
  on poll $f do
  when success:
    accept
  when not_found or failure:
    reject 550 5.1.0 "Sender validity not confirmed"
  when temp_failure:
    tempfail 450 4.1.0 "Try again later"
  done
done
```

4.24 Modules

A *module* is a logically isolated part of code that implements a separate concern or feature and contains a collection of conceptually united functions and/or data. Each module occupies a separate compilation unit (i.e. file). The functionality provided by a module is incorporated into another module or the main program by *requiring* this module or by *importing* the desired components from it.

4.24.1 Declaring Modules

A module file must begin with a *module declaration*:

```
module modname [interface-type].
```

Note the final dot.

The *modname* parameter declares the name of the module. It is recommended that it be the same as the file name without the '`.mfl`' extension. The module name must be a valid MFL literal. It also must not coincide with any defined MFL symbol, therefore we recommend to always quote it (see example below).

The optional parameter *interface-type* defines the *default scope of visibility* for the symbols declared in this module. If it is '`public`', then all symbols declared in this module are made public (importable) by default, unless explicitly declared otherwise (see Section 4.24.2 [scope of visibility], page 109). If it is '`static`', then all symbols, not explicitly marked as public, become static. If the *interface-type* is not given, '`public`' is assumed.

The actual MFL code follows the '`module`' line.

The module definition is terminated by the *logical end* of its compilation unit, i.e. either by the end of file, or by the keyword `bye`, whichever occurs first.

Special keyword `bye` may be used to prematurely end the current compilation unit before the physical end of the containing file. Any material between `bye` and the end of file is ignored by the compiler.

Let's illustrate these concepts by writing a module '`revip`':

```
module 'revip' public.

func revip(string ip)
  returns string
do
  return inet_ntoa(ntohl(inet_aton(ip)))
done

bye

This text is ignored.  You may put any additional
documentation here.
```

4.24.2 Scope of Visibility

Scope of Visibility of a symbol defines from where this symbol may be referred to. Symbols in MFL may have either of the following two scopes:

Public Public symbols are visible from the current module, as well as from any external modules, including the main script file, provided that they are properly imported (see Section 4.24.3 [import], page 109).

Static Static symbols are visible only from the current module. There is no way to refer to them from outside.

The default scope of visibility for all symbols declared within a module is defined in the module declaration (see Section 4.24.1 [module structure], page 108). It may be overridden for any individual symbol by prefixing its declaration with an appropriate *qualifier*: either `public` or `static`.

4.24.3 Require and Import

Functions or variables declared in another module must be *imported* prior to their actual use. MFL provides two ways of doing so: by *requiring* the entire module or by importing selected symbols from it.

Modules are looked up in the *module search path*. The default module search path consists of two directories:

1. *prefix*`/share/mailfromd`
2. *prefix*`/share/mailfromd/9.2`

where *prefix* stands for the installation prefix (normally `/usr` or `/usr/local`).

Module search path can be changed in the configuration file, using the `module-path` statement (see Section 7.2 [conf-base], page 220), or from the command line, using the `-P` (`--module-path`) option (see [–module-path], page 234).

`require` *modname* [Module Import]

The `require` statement instructs the compiler to locate the module *modname* and to load all public interfaces from it.

The compiler looks for the file *`modname`*`.mfl` in the module search path. If no such file is found, a compilation error is reported.

For example, the following statement:

```
require revip
```

imports all interfaces from the module `revip.mfl`.

Another, more sophisticated way to import from a module is to use the '`from ... import`' construct:

```
from module import symbols.
```

Note the final dot. The '`from`' and '`module`' statements are the only two constructs in MFL that require the delimiter.

The *module* has the same semantics as in the `require` construct. The *symbols* is a comma-separated list of symbol names to import from *module*. A symbol name may be given in several forms:

1. Literal

   Literals specify exact symbol names to import. For example, the following statement imports from module `A.mfl` symbols '`foo`' and '`bar`':

   ```
   from A import foo,bar.
   ```

2. Regular expression

   Regular expressions must be surrounded by slashes. A regular expression instructs the compiler to import all symbols whose names match that expression. For example, the following statement imports from `A.mfl` all symbols whose names begin with '`foo`' and contain at least one digit after it:

   ```
   from A import '/^foo.*[0-9]/'.
   ```

   The type of regular expressions used in the '`from`' statement is controlled by `#pragma regex` (see Section 4.5.3 [regex], page 56).

3. Regular expression with transformation

   Regular expression may be followed by a *s-expression*, i.e. a `sed`-like expression of the form:

   ```
   s/regexp/replace/[flags]
   ```

   where *regexp* is a *regular expression*, *replace* is a replacement for each part of the input that matches *regexp*. S-expressions and their parts are discussed in detail in [s-expression], page 128.

   The effect of such construct is to import all symbols that match the regular expression and apply the s-expression to their names.

   For example:

   ```
   from A import '/^foo.*[0-9]/s/.*/my_&/'.
   ```

   This statement imports all symbols whose names begin with '`foo`' and contain at least one digit after it, and renames them, by prefixing their names with the string '`my_`'. Thus, if `A.mfl` declared a function '`foo_1`', it becomes visible under the name of '`my_foo_1`'.

4.25 Dynamically Loaded Modules

Native mailfromd modules described above rely on the functions provided by the `mailfromd` binary. For more sophisticated tasks you might need to use C functions, either for efficiency reasons or to make use of some third-party library. This is possible using special kind of modules called *mfmod*.

An mfmod consists of two major parts: a dynamically loaded library that provides its main functionality and a small *interface* mailfromd module. The convention is that for the module *x* the library is named `mfmod_x.so`[10], and the interface module file is `x.mfl`.

At the time of this writing, three mfmods exist:

mfmod_pcre
: Provides support for Perl-compatible regular expressions. It also contains a special function for scanning an email message for a match to a regular expression. See *Mfmod_pcre*.

mfmod_ldap
: Functions for searching in LDAP directory. See *Mfmod_ldap*.

mfmod_openmetrics
: Openmetrics support for `mailfromd`. See *mfmod_openmetrics reference*.

The subsections below describe the internal structure of an mfmod in detail.

4.25.1 Loadable Library

External functions in the loadable library must be declared as

```
int funcname(long count, MFMOD_PARAM *param, MFMOD_PARAM *retval);
```

The `MFMOD_PARAM` type is declared in the header file `mailfromd/mfmod.h`, which must be included at the start of the source code.

`MFMOD_PARAM` *type string number* [mfmod type]

This type is defined as follows:

```
typedef struct mfmod_param {
        mfmod_data_type type;
        union {
                char *string;
                long number;
                mu_message_t message;
        };
} MFMOD_PARAM;
```

The `type` fields defines the type of the data represented by the object. Its possible values are:

`mfmod_string` [mfmod constant]

String data.

`mfmod_number` [mfmod constant]

Numeric data.

[10] The actual suffix depends on operating system. It is '`.so`' on all POSIX systems.

`mfmod_message` [mfmod constant]

A `mailutils` message object (`mu_message_t`).

The actual data are accessed as `string`, `number`, or `message`, depending on the value of `type`.

The first parameter in the external function declaration, *count*, is the number of arguments passed to that function. Actual arguments are passed in the `MFMOD_PARAM` array *param*. The function should never modify its elements. If the function returns a value to MFL, it must pass it in the *retval* parameter. For example, the following code returns the numeric value '`1`':

```
retval->type = mfmod_number;
retval->number = 1;
```

To return a string value, allocate it using `malloc`, `calloc` or a similar function, like this:

```
retval->type = mfmod_string;
retval->string = strdup("text");
```

If a message is returned, it should be created using mailutils message creation primitives. `Mailutils` will call `mu_message_destroy` on it, when it is no longer used.

The return value (in the C sense) of the function is used to determine whether it succeeded or not. Zero means success. Returning -1 causes a runtime exception `e_failure` with a generic error text indicating the names of the module and function that caused the exception. Any other non-zero value is treated as a `mailfromd` exception code (see Section 4.22 [Exceptions], page 99). In this case an additional textual explanation of the error can be supplied in the `retval` variable, whose type must then be set to `mfmod_string`. This explanation string must be allocated using `malloc`.

To facilitate error handling, the following functions are provided (declared in the `mailfromd/mfmod.h` header file):

`int mfmod_error (MFMOD_PARAM *retval, int ecode, char const *fmt, ...)` [mfmod]

Raises exception *ecode* with the error message formatted from the variadic arguments using `printf`-style format string *fmt*.

Example use:

```
if (error_condition)
   return mfmod_error(retval, "error %s occurred", error_text);
```

`int mfmod_error_argtype (MFMOD_PARAM *param, MFMOD_PARAM *retval, int n, int exptype)` [mfmod]

Reports argument type mismatch error (`e_inval` with appropriately formatted error text). Arguments are:

param
retval The two arguments passed to the interface function.

n 0-based index of the erroneous argument in *param*.

exptype Expected data type of *param*`[n]`.

You will seldom need to use this function directly. Instead, use the `ASSERT_ARGTYPE` macro described below.

`char const * mfmod_data_type_str(mfmod_data_type type)` [mfmod]
Returns the MFL name of the mfmod data type *type*.

The following convenience macros are provided for checking the number of argument and their types and returning error if necessary:

`ASSERT_ARGCOUNT` (*MFMOD_PARAM* *`retval`, *long* `count`, *long* `expcount`) [mfmod macro]
Assert that the number of arguments (*count*) equals the expected number (*expcount*). If it does not, return the `e_inval` exception with a descriptive error text.

retval and *count* are corresponding arguments from the calling function.

`ASSERT_ARGTYPE` (*MFMOD_PARAM* *`param`, *MFMOD_PARAM* *`retval`, *int* `n`, *int* `exptype`) [mfmod macro]
Check if the data type of the *n*th parameter (i.e. `param[n]`) is *exptype* and return the `e_inval` exception if it does not.

As an example, suppose you want to write an interface to the system `crypt` function. The loadable library source, `mfmod_crypt.c`, will look as follows:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <mailfromd/mfmod.h>
#include <mailfromd/exceptions.h>

/*
 * Arguments:
 *   param[0] - key string to hash.
 *   param[1] - salt value.
 */
int
cryptval(long count, MFMOD_PARAM *param, MFMOD_PARAM *retval)
{
     char *hash;

     /* Check if input arguments are correct: */
     ASSERT_ARGCOUNT(retval, count, 2);
     ASSERT_ARGTYPE(param, retval, 0, mfmod_string);
     ASSERT_ARGTYPE(param, retval, 1, mfmod_string);

     /* Hash the key string. */
     hash = crypt(param[0].string, param[1].string);

     /* Return string to MFL */
     retval->type = mfmod_string;
     retval->string = strdup(hash);

     /* Throw exception if out of memory */
```

```
        if (retval->string == NULL)
             return -1;

        return 0;
    }
```

The exact way of building a loadable library from this source file depends on the operating system. For example, on GNU/Linux you would do:

```
cc -shared -fPIC -DPIC -omfmod_crypt.so -lcrypt mfmod_crypt.c
```

The preferred and portable way of doing so is via `libtool` (see *Libtool*). `Mailfromd` provides a special command `mfmodnew` that creates infrastructure necessary for building loadable modules. See Section 4.25.3 [mfmodnew], page 115.

4.25.2 Interface Module

The interface module is responsible for loading the library, and providing MFL wrappers over external functions defined in it.

For the first task, the `dlopen` function is provided. It takes a single argument, the file name of the library to load. This can be an absolute pathname, in which case it is used as is, or a relative file name, which will be searched in the *library search path* (see [mfmod-path], page 231). On success, the function returns the *library handle*, which will be used in subsequent calls to identify that library. On error, a runtime exception is signalled.

It is common to call the `dlopen` function in the `startup` section of the interface module (see Section 4.15.2 [startup/shutdown], page 77), so that the library gets loaded at the program startup. For example:

```
static number libh

prog startup
do
  set libh dlopen("mfmod_crypt.so")
done
```

The function `dlcall` is provided to call a function from the already loaded library. It is a variadic function with three mandatory parameters:

1. The handle of the loadable library as returned by `dlopen`.
2. Name of the external function to call.
3. Type string

The *type string* argument declares data types of the variable arguments. It contains a single letter for each additional argument passed to `dlcall`. The valid letters are:

s
: The argument is of string type.

n
d
: The argument is of numeric type.

m
: The argument is of message type.

For example, the following will call the `cryptval` function defined in the previous section (supposing `key` and `salt` are two string MFL variables):

```
set x dlcall(libh, "cryptval", "ss", key, salt)
```

The last letter in type string can be '`+`' or '`*`'. Both mean that any number of arguments are allowed (all of the type given by the penultimate type letter). The difference between the two is that '`+`' allows for one or more arguments, while '`*`' allows for zero or more arguments. For example, '`n+`' means one or more numeric arguments, and '`n*`' means zero or more such arguments. Both are intended to be used in variadic functions, e.g.:

```
func pringstddev(number ...)
  returns number
do
  return dlcall(libh, "stddev", "n*", $@)
done
```

The `dlcall` function returns the value returned by the library function it invoked. If the library function returns no meaningful value, it is recommended to use the `void` type cast around the `dlcall` invocation (see [void type cast], page 89). E.g.:

```
func out(string text)
do
  void(dlcall(libh, "output", "s", text))
done
```

Without `void` type cast, the definition above will produce the following warning when compiled:

```
return from dlcall is ignored
```

4.25.3 Creating a Mfmod Structure

The `mfmodnew` provides a convenient start for writing a new *mfmod*. Given a name of the planned module, this command creates directory `mfmod_`*`name`* and populates it with the files necessary for building the new module using GNU autotools, as well as boilerplate files for the loadable library and interface module.

Let's see how to use it to create the `crypt` module outlined in previous subsections.

First, invoke the command:

```
$ mfmodnew crypt
mfmodnew: setting up new module in mfmod_crypt
```

Let's change to the new directory and see the files in it:

```
$ cd mfmod_crypt
$ ls
Makefile.am  configure.ac  crypt.mfl  mfmod_crypt.c
```

Now, open the `mfmod_crypt.c` file and add to it the definition of the `cryptval` function (see Section 4.25.1 [Loadable Library], page 111). Then, add the interface function definition from Section 4.25.2 [Interface Module], page 114, to file `crypt.mfl`.

The last thing to do is to edit `configure.ac`. The `crypt` function requires the `libcrypt` library, so the following line should be added to the '`Checks for libraries.`' section.

```
AC_CHECK_LIB([crypt], [crypt])
```

Now, run `autoreconf`, as follows:

```
$ autoreconf -f -i -s
```

It will bootstrap the autotools infrastructure, importing additional files as necessary. Once done, you can build the project:

```
$ ./configure
$ make
```

Notice, that if the `autoreconf` stage ends abnormally with a diagnostics like:

```
configure.ac:21: error: possibly undefined macro: AC_MFMOD
```

that means that `autoconf` was unable to find the file `mfmod.m4`, which provides that macro. That's because the directory where this file is installed is not searched by `autoreconf`. To fix this, supply the name of that directory using the `-I` option. E.g. assuming `mfmod.m4` is installed in `/usr/local/share`:

```
$ autoreconf -fis -I /usr/local/share/aclocal
```

4.25.3.1 mfmodnew invocation

The `mfmodnew` is invoked as:

```
mfmodnew [options] modname [dir]
```

where *modname* is the name of the new module and *dir* is the directory where to store the module infrastructure files. Normally you would omit *dir* altogether: in this case the utility will use `mfmod_`*modname* as the directory name.

Options are:

`-C` *dir* Search for template files in *dir*, instead of the default location.

`-e` *email* Supply the author's email. The email is passed as argument to the `AC_INIT` macro in `configure.ac`. By default it is constructed as '*username*`@`*hostname*'. If it is incorrect, you can either edit `configure.ac` afterwards, or just supply the correct one using this option.

`-q` Suppress informative messages.

`-h` Display a short command line usage help.

4.26 MFL Preprocessor

Before compiling the script file, `mailfromd` preprocesses it. The built-in preprocessor handles only file inclusion (see Section 4.2 [include], page 53), while the rest of traditional facilities, such as macro expansion, are supported via `m4`, which is used as an external preprocessor.

The detailed description of `m4` facilities lies far beyond the scope of this document. You will find a complete user manual in Section "GNU M4" in *GNU M4 macro processor*. For the rest of this section we assume the reader is sufficiently acquainted with `m4` macro processor.

The external preprocessor is invoked with `-s` flag, instructing it to include line synchronization information in its output, which is subsequently used by MFL compiler for purposes of error reporting. The initial set of macro definitions is supplied in *preprocessor setup* file `pp-setup`, located in the library search path[11], which is fed to the preprocessor input before

[11] It is usually located in `/usr/local/share/mailfromd/9.2/include/pp-setup`.

the script file itself. The default `pp-setup` file renames all `m4` built-in macro names so they all start with the prefix '`m4_`'[12]. It changes comment characters to '`/*`', '`*/`' pair, and leaves the default quoting characters, grave ('`` ` ``') and acute ('`'`') accents without change. Finally, `pp-setup` defines several useful macros (see Section 4.26.3 [m4 macros], page 118).

4.26.1 Preprocessor Configuration

The preprocessor is configured in the mailfromd configuration file, using the preprocessor statement (see Section 7.3 [conf-preprocessor], page 221). The default settings correspond to the following configuration:

```
preprocessor {
  # Enable preprocessor
  enable yes;
  # Preprocessor command line stub.
  command "m4 -s";
  # Pass current include path to the preprocessor via -I options.
  pass-includes false;
  # Pass to the preprocessor the feature definitions via -D options
  # as well as any -D/-U options from the command line
  pass-defines true;
  # Name of the preprocessor setup file.  Unless absolute, it is
  # looked up in the include path.
  setup-file "pp-setup";
}
```

If `pass-includes` is true, the `command` value is augmented by zero or more `-I` options supplying it the mailfromd include search path (see [include search path], page 53).

Furthermore, if `pass-defines` is set, zero or more `-D` options defining optional features are passed to it (e.g. `-DWITH_DKIM`) as well as any `-D` and `-U` options from the mailfromd command line.

Unless the value of `setup-file` begins with a slash, the file with this name is looked up in the current include search path. If found, its absolute name is passed to the preprocessor as first argument.

If it begins with a slash, it is passed to the preprocessor as is.

4.26.2 Preprocessor Usage

You can obtain the preprocessed output, without starting actual compilation, using `-E` command line option:

```
$ mailfromd -E file.mfl
```

The output is in the form of preprocessed source code, which is sent to the standard output. This can be useful, among others, to debug your own macro definitions.

Macro definitions and deletions can be made on the command line, by using the `-D` and `-U` options, provided that their use is allowed by the `pass-defines` preprocessor configuration

[12] This is similar to GNU m4 `--prefix-builtin` options. This approach was chosen to allow for using non-GNU `m4` implementations as well.

setting (see Section 4.26.1 [Configuring Preprocessor], page 117. They have the following format:

`-D` *name*`[=`*value*`]`
`--define=`*name*`[=`*value*`]`
: Define a symbol *name* to have a value *value*. If *value* is not supplied, the value is taken to be the empty string. The *value* can be any string, and the macro can be defined to take arguments, just as if it was defined from within the input using the `m4_define` statement.

  For example, the following invocation defines symbol `COMPAT` to have a value `43`:

  ```
  $ mailfromd -DCOMPAT=43
  ```

`-U` *name*

`--undefine=`*name*
: A counterpart of the `-D` option is the option `-U` (`--undefine`). It undefines a preprocessor symbol whose name is given as its argument. The following example undefines the symbol `COMPAT`:

  ```
  $ mailfromd -UCOMPAT
  ```

The following two options are supplied mainly for debugging purposes:

`--no-preprocessor`
: Disables the external preprocessor.

`--preprocessor[=`*command*`]`
: Use *command* as external preprocessor. If *command* is not supplied, use the default preprocessor, overriding the `enable` preprocessor configuration setting.

  Be especially careful with this option, because `mailfromd` cannot verify whether *command* is actually some kind of a preprocessor or not.

4.26.3 Preprocessor Macros

`boolean defined (`*identifier*`)` [M4 Macro]

The *identifier* must be the name of an optional abstract argument to the function. This macro must be used only within a function definition. It expands to the MFL expression that yields `true` if the actual parameter is supplied for *identifier*. For example:

```
func rcut(string text; number num)
  returns string
do
  if (defined(num))
    return substr(text, length(text) - num)
  else
    return text
  fi
done
```

This function will return last *num* characters of *text* if *num* is supplied, and entire *text* otherwise, e.g.:

```
rcut("text string") ⇒ "text string"
rcut("text string", 3) ⇒ "ing"
```

Invoking the `defined` macro with the name of a mandatory argument yields `true`

`printf` (*format*, ...) [M4 Macro]

Provides a `printf` statement, that formats its optional parameters in accordance with *format* and sends the resulting string to the current log output (see Section 3.19 [Logging and Debugging], page 43). See Section 5.5 [String formatting], page 133, for a description of *format*.

Example usage:

```
printf('Function %s returned %d', funcname, retcode)
```

`string _` (*msgid*) [M4 Macro]

A convenience macro. Expands to a call to `gettext` (see Section 5.40 [NLS Functions], page 207).

`string_list_iterate` (*list*, *delim*, *var*, *code*) [M4 Macro]

This macro intends to compensate for the lack of array data type in MFL. It splits the string *list* into segments delimited by string *delim*. For each segment, the MFL code *code* is executed. The code can use the variable *var* to refer to the segment string.

For example, the following fragment prints names of all existing directories listed in the `PATH` environment variable:

```
string path getenv("PATH")
string seg

string_list_iterate(path, ":", seg, `
     if access(seg, F_OK)
       echo "%seg exists"
     fi')
```

Care should be taken to properly quote its arguments. In the code below the string `str` is treated as a comma-separated list of values. To avoid interpreting the comma as argument delimiter the second argument must be quoted:

```
string_list_iterate(str, `","', seg, `
     echo "next segment: " . seg')
```

`N_` (*msgid*) [M4 Macro]

A convenience macro, that expands to *msgid* verbatim. It is intended to mark the literal strings that should appear in the `.po` file, where actual call to `gettext` (see Section 5.40 [NLS Functions], page 207) cannot be used. For example:

```
/* Mark the variable for translation: cannot use gettext here */
string message N_("Mail accepted")

prog envfrom
```

```
do
  ...
  /* Translate and log the message */
  echo gettext(message)
```

4.27 Example of a Filter Script File

In this section we will discuss a working example of the filter script file. For the ease of illustration, it is divided in several sections. Each section is prefaced with a comment explaining its function.

This filter assumes that the `mailfromd.conf` file contains the following:

```
relayed-domain-file (/etc/mail/sendmail.cw,
                     /etc/mail/relay-domains);
io-timeout 33;
database cache {
  negative-expire-interval 1 day;
  positive-expire-interval 2 weeks;
};
```

Of course, the exact parameter settings may vary, what is important is that they be declared. See Chapter 7 [Mailfromd Configuration], page 219, for a description of `mailfromd` configuration file syntax.

Now, let's return to the script. Its first part defines the configuration settings for this host:

```
#pragma regex +extended +icase

set mailfrom_address "<>"
set ehlo_domain "gnu.org.ua"
```

The second part loads the necessary source modules:

```
require 'status'
require 'dns'
require 'rateok'
```

Next we define `envfrom` handler. In the first two rules, it accepts all mails coming from the null address and from the machines which we relay:

```
prog envfrom
do
  if $f = ""
    accept
  elif relayed hostname($client_addr)
    accept
  elif hostname($client_addr) = $client_addr
    reject 550 5.7.7 "IP address does not resolve"
```

Next rule rejects all messages coming from hosts with dynamic IP addresses. A regular expression used to catch such hosts is not 100% fail-proof, but it tries to cover most existing host naming patterns:

```
    elif hostname($client_addr) matches
          ".*(adsl|sdsl|hdsl|ldsl|xdsl|dialin|dialup|\
ppp|dhcp|dynamic|[-.]cpe[-.]).*"
      reject 550 5.7.1 "Use your SMTP relay"
```

Messages coming from the machines whose host names contain something similar to an IP are subject to strict checking:

```
    elif hostname($client_addr) matches
    ".*[0-9]{1,3}[-.][0-9]{1,3}[-.][0-9]{1,3}[-.][0-9]{1,3}.*"
      on poll host $client_addr for $f do
      when success:
        pass
      when not_found or failure:
        reject 550 5.1.0 "Sender validity not confirmed"
      when temp_failure:
        tempfail
      done
```

If the sender domain is relayed by any of the '`yahoo.com`' or '`nameserver.com`' '`MX`'s, no checks are performed. We will greylist this message in `envrcpt` handler:

```
    elif $f mx fnmatches "*.yahoo.com"
         or $f mx fnmatches "*.namaeserver.com"
      pass
```

Finally, if the message does not meet any of the above conditions, it is verified by the standard procedure:

```
    else
      on poll $f do
      when success:
        pass
      when not_found or failure:
        reject 550 5.1.0 "Sender validity not confirmed"
      when temp_failure:
        tempfail
      done
    fi
```

At the end of the handler we check if the sender-client pair does not exceed allowed mail sending rate:

```
    if not rateok("$f-$client_addr", interval("1 hour 30 minutes"), 100)
      tempfail 450 4.7.0 "Mail sending rate exceeded.  Try again later"
    fi
done
```

Next part defines the `envrcpt` handler. Its primary purpose is to greylist messages from some domains that could not be checked otherwise:

```
prog envrcpt
do
  set gltime 300
  if $f mx fnmatches "*.yahoo.com"
```

```
        or $f mx fnmatches "*.namaeserver.com"
        and not dbmap("/var/run/whitelist.db", $client_addr)
       if greylist("$client_addr-$f-$rcpt_addr", gltime)
         if greylist_seconds_left = gltime
           tempfail 450 4.7.0
                  "You are greylisted for %gltime seconds"
         else
           tempfail 450 4.7.0
                  "Still greylisted for " .
                  %greylist_seconds_left . " seconds"
         fi
       fi
     fi
   done
```

4.28 Reserved Words

For your reference, here is an alphabetical list of all reserved words:

- __defpreproc__
- __defstatedir__
- __file__
- __function__
- __line__
- __major__
- __minor__
- __module__
- __package__
- __patch__
- __preproc__
- __statedir__
- __version__
- accept
- add
- and
- alias
- begin
- break
- bye
- case
- catch
- const
- continue

- default
- delete
- discard
- do
- done
- echo
- end
- elif
- else
- fi
- fnmatches
- for
- from
- func
- if
- import
- loop
- matches
- module
- next
- not
- number
- on
- or
- pass
- precious
- prog
- public
- reject
- replace
- return
- returns
- require
- set
- static
- string
- switch
- tempfail
- throw

- try
- vaptr
- when
- while

Several keywords are context-dependent: `mx` is a keyword if it appears before `matches` or `fnmatches`. Following strings are keywords in `on` context:

- as
- host
- poll

The following keywords are preprocessor macros:

- defined
- _ (an underscore)
- N_

Any keyword beginning with a '`m4_`' prefix is a reserved preprocessor symbol.

5 The MFL Library Functions

This chapter describes library functions available in Mailfromd version 9.2. For the simplicity of explanation, we use the word '`boolean`' to indicate variables of numeric type that are used as boolean values. For such variables, the term '`False`' stands for the numeric 0, and '`True`' for any non-zero value.

5.1 Sendmail Macro Access Functions

`string getmacro (string` *`macro`*`)` [Built-in Function]

Returns the value of Sendmail macro *macro*. If *macro* is not defined, raises the `e_macroundef` exception.

Calling `getmacro(`*`name`*`)` is completely equivalent to referencing `${`*`name`*`}`, except that it allows to construct macro names programmatically, e.g.:

```
if getmacro("auth_%var") = "foo"
  ...
fi
```

`boolean macro_defined (string` *`name`*`)` [Built-in Function]

Return true if Sendmail macro *name* is defined.

Notice, that if your MTA supports macro name negotiation[1], you will have to export macro names used by these two functions using '`#pragma miltermacros`' construct. Consider this example:

```
func authcheck(string name)
do
  string macname "auth_%name"
  if macro_defined(macname)
    if getmacro(macname)
      ...
    fi
  fi
done

#pragma miltermacros envfrom auth_authen

prog envfrom
do
  authcheck("authen")
done
```

In this case, the parser cannot deduce that the `envfrom` handler will attempt to reference the '`auth_authen`' macro, therefore the '`#pragma miltermacros`' is used to help it.

[1] That is, if it supports Milter protocol 6 and upper. Sendmail 8.14.0 and Postfix 2.6 and newer do. MeTA1 (via `pmult`) does as well. See Chapter 9 [MTA Configuration], page 243, for more details.

5.2 `tr`, `dc`, and `sq` functions

`string tr (string` *`subj`*`, string` *`set1`*`, string` *`set2`*`)` [Built-in Function]

Translates characters in string *subj* and returns the resulting string.

Translation rules are defined by two character sets: *set1* is a set of characters which, when encountered in *subj*, must be replaced with the corresponding characters from *set2*. E.g.:

```
tr('text', 'tx', 'ni') ⇒ 'nein'
```

The source set *set1* can contain *character classes*, sets of characters enclosed in square brackets. A character class matches the input character if that character is listed in the class. When a match occurs, the character is replaced with the corresponding character from *set2*:

```
tr('abacus', '[abc]', '_') ⇒ '____us'
```

An exclamation sign at the beginning of a character class reverses its meaning, i.e. the class matches any character *not* listed in it:

```
tr('abacus', '[!abc]', '_') ⇒ 'abac__'
```

A character set can contain *ranges*, specified as the first and last characters from the range separated by a dash. A range '`x-y`' comprises all characters between *x* and *y* inclusive. For example, '`[a-d]`' is equivalent to '`[abcd]`'. Character sets must be ascending, i.e. '`[a-d]`' is correct, but '`[d-a]`' is not. You may include '`-`' in its literal meaning by making it the first or last character between the brackets: '`[0-9-]`' matches any digit or dash.

Similarly, to include a closing bracket, make it the first character in the class (after negation character, for excluding ranges), e.g. '`[][!]`' matches the three characters '`[`', '`]`' and '`!`', whereas '`[!][]`' matches any character except '`[`' and '`]`'.

Named character classes are special reserved names between '`[:`' and '`:]`' delimiters:

`[:alnum:]`
: Matches any alphanumeric character. Equivalent to '`[:alpha:][:digit:]`'.

`[:alpha:]`
: Matches any alphabetic character.

`[:blank:]`
: Matches horizontal space or tab.

`[:cntrl:]`
: Matches a *control character*, i.e. a character with ASCII code less than 32.

`[:digit:]`
: Matches a decimal digit (0 through 9).

`[:graph:]`
: Matches any printable character except space (horizontal space and tab).

`[:lower:]`
: Matches any lowercase letter.

`[:print:]`
: Matches any printable character including space.

`[:punct:]`
: Matches any printable character which is not a space or an alphanumeric character.

`[:space:]`
: Matches '`white-space`' characters: horizontal space (ASCII 32), formfeed (ASCII 12, or '`\f`'), newline (ASCII 10, or '`\n`'), carriage return (ASCII 13, or '`\r`'), horizontal tab (ASCII 9, or '`\t`'), and vertical tab (ASCII 11, or '`\v`').

`[:upper:]`
: Matches any upper case letter.

`[:xdigit:]`
: Matches any hexagesimal digit: '`0`' through '`9`', '`a`' through '`f`' and '`A`' through '`F`'.

Named classes can appear in character classes in *set1* anywhere a regular character is allowed. Examples:

'`[][:alpha:]-]`'
: Mathes alphabet letters (both cases), digits, closing bracket and dash.

'`[!][:alpha:]-]`'
: A complement of the above: matches any character except the ones listed above.

'`[[:xdigit:][:blank:]]`'
: Matches any hexagesimal digit or horizontal whitespace characters.

The replacement set must not be empty. Its length must be equal to or less than that of *set1* (character classes being counted as one character). If *set1* contains more characters than *set2*, the surplus ones will be translated to the last character from *set2*:

```
tr('lasted', 'alde', 'iL?') ⇒ 'List??'
```

Both sets can contain character ranges, represented as '*c1-c2*'. Whenever a range appears in *set1*, a range must appear in the corresponding position of *set2*:

```
tr('gnu', 'a-z', 'A-Z') ⇒ 'GNU'
```

Character ranges are not to be confused with ranges in character classes: they are similiar, but quite distinct. Both match a single character, but while ranges translate to a corresponding character from the replacement range, ranges within character class translate to a single character:

```
tr('gnu', '[a-z]', 'A') ⇒ 'AAA'
```

Character ranges in *set1* must always be in ascending order (i.e. '`a-z`' is allowed, whereas '`z-a`' is not). Ranges in *set2* can be both ascending and descending, e.g.:

```
tr('8029', '0-9', '9-0') ⇒ '1970'
```

To translate a dash, place it as the first or last character in *set1*:

```
tr('in-place', '-ilp', ' Irg') ⇒ 'In grace'
```

The `tr` function will raise the `e_inval` exception if *set2* is empty or *set1* contains a range without matching range in *set2*. It will raise the `e_range` exception, if a descending range appears in *set1* or number of characters in a range from *set1* does not match that from the corresponding range in *set2*.

`string dc (string subj, string set1)` [Built-in Function]

Deletes from *subj* characters that appear in *set1*. The syntax of *set1* is as described in `tr`, except that character ranges are treated as if appearing within character class (e.g. '`a-z`' is the same as '`[a-z]`').

For example, `dc(subj, '0-9')` removes decimal digits from first argument.

`string sq (string subj, string set1)` [Built-in Function]

Squeezes repeats, i.e. replaces each sequence of a repeated character that is listed in *set1*, with a single occurrence of that character. The syntax of *set1* is as described in `tr`, except that character ranges are treated as if appearing within character class (e.g. '`a-z`' is the same as '`[a-z]`').

For example, `sq(subj, '[[:space:]]')` replaces multiple occurrences of whitespace characters with a single character.

5.3 The `sed` function

The `sed` function allows you to transform a string by replacing parts of it that match a regular expression with another string. This function is somewhat similar to the `sed` command line utility (hence its name) and bears similarities to analogous functions in other programming languages (e.g. `sub` in `awk` or the `s//` operator in `perl`).

`string sed (string subject, expr, ...)` [Built-in Function]

The *expr* argument is an *s-expressions* of the the form:

```
s/regexp/replacement/[flags]
```

where *regexp* is a *regular expression*, and *replacement* is a replacement string for each part of the *subject* that matches *regexp*. When `sed` is invoked, it attempts to match *subject* against the *regexp*. If the match succeeds, the portion of *subject* which was matched is replaced with *replacement*. Depending on the value of *flags* (see [global replace], page 128), this process may continue until the entire *subject* has been scanned.

The resulting output serves as input for next argument, if such is supplied. The process continues until all arguments have been applied.

The function returns the output of the last s-expression.

Both *regexp* and *replacement* are described in detail in Section "The 's' Command" in *GNU sed*.

Supported *flags* are:

'`g`' Apply the replacement to *all* matches to the *regexp*, not just the first.

'`i`' Use case-insensitive matching. In the absence of this flag, the value set by the recent `#pragma regex icase` is used (see [pragma regex], page 56).

'`x`' *regexp* is an *extended regular expression* (see Section "Extended regular expressions" in *GNU sed*). In the absence of this flag, the value set by the recent `#pragma regex extended` (if any) is used (see [pragma regex], page 56).

'*number*' Only replace the *number*th match of the *regexp*.

Note: the POSIX standard does not specify what should happen when you mix the '`g`' and *number* modifiers. `Mailfromd` follows the GNU `sed` implementation in this regard, so the interaction is defined to be: ignore matches before the *number*th, and then match and replace all matches from the *number*th on.

Any delimiter can be used in lieue of '`/`', the only requirement being that it be used consistently throughout the expression. For example, the following two expressions are equivalent:

```
s/one/two/
s,one,two,
```

Changing delimiters is often useful when the *regex* contains slashes. For instance, it is more convenient to write `s,/,-,` than `s/\//-/`.

Here is an example of `sed` usage:

```
set email sed(input, 's/^<(.*)>$/\1/x')
```

It removes angle quotes from the value of the '`input`' variable and assigns the result to '`email`'.

To apply several s-expressions to the same input, you can either give them as multiple arguments to the `sed` function:

```
set email sed(input, 's/^<(.*)>$/\1/x', 's/(.+@)(.+)/\1\L\2\E/x')
```

or give them in a single argument separated with semicolons:

```
set email sed(input, 's/^<(.*)>$/\1/x;s/(.+@)(.+)/\1\L\2\E/x')
```

Both examples above remove optional angle quotes and convert the domain name part to lower case.

Regular expressions used in `sed` arguments are controlled by the `#pragma regex`, as another expressions used throughout the MFL source file. To avoid using the '`x`' modifier in the above example, one can write:

```
#pragma regex +extended
set email sed(input, 's/^<(.*)>$/\1/', 's/(.+@)(.+)/\1\L\2\E/')
```

See Section 4.5.3 [regex], page 56, for details about that `#pragma`.

So far all examples used constant s-expressions. However, this is not a requirement. If necessary, the expression can be stored in a variable or even constructed on the fly before passing it as argument to `sed`. For example, assume that you wish to remove the domain part from the value, but only if that part matches one of predefined domains. Let a regular expression that matches these domains be stored in the variable `domain_rx`. Then this can be done as follows:

```
set email sed(input, "s/(.+)(@%domain_rx)/\1/")
```

If the constructed regular expression uses variables whose value should be matched exactly, such variables must be quoted before being used as part of the regexp. Mailfromd provides a convenience function for this:

`string qr (string` *`str`*`[; string` *`delim`*`])` [Built-in Function]
Quote the string *str* as a regular expression. This function selects the characters to be escaped using the currently selected regular expression flavor (see Section 4.5.3 [regex], page 56). At most two additional characters that must be escaped can be supplied in the *delim* optional parameter. For example, to quote the variable '`x`' for use in double-quoted s-expression:

```
qr(x, '/"')
```

5.4 String Manipulation Functions

`string escape (string` *`str`*`, [string` *`chars`*`])` [Built-in Function]
Returns a copy of *str* with the characters from *chars* escaped, i.e. prefixed with a backslash. If *chars* is not specified, '`\"`' is assumed.

```
escape('"a\tstr"ing') ⇒ '\"a\\tstr\"ing'
escape('new "value"', '\" ') ⇒ 'new\ \"value\"'
```

`string unescape (string` *`str`*`)` [Built-in Function]
Performs the reverse to '`escape`', i.e. removes any prefix backslash characters.

```
unescape('a \"quoted\" string') ⇒ 'a "quoted" string'
```

`string unescape (string` *`str`*`, [string` *`chars`*`])` [Built-in Function]

`string domainpart (string` *`str`*`)` [Built-in Function]
Returns the domain part of *str*, if it is a valid email address, otherwise returns *str* itself.

```
domainpart("gray") ⇒ "gray"
domainpart("gray@gnu.org.ua") ⇒ "gnu.org.ua"
```

`number index (string` *`s`*`, string` *`t`*`)` [Built-in Function]
`number index (string` *`s`*`, string` *`t`*`, number` *`start`*`)` [Built-in Function]
Returns the index of the first occurrence of the string *t* in the string *s*, or -1 if *t* is not present.

```
index("string of rings", "ring") ⇒ 2
```

Optional argument *start*, if supplied, indicates the position in string where to start searching.

```
index("string of rings", "ring", 3) ⇒ 10
```

To find the last occurrence of a substring, use the function *rindex* (see [rindex], page 131).

`number interval (string` *`str`*`)` [Built-in Function]
Converts *str*, which should be a valid time interval specification (see [time interval specification], page 220), to seconds.

`number length (string str)` [Built-in Function]

Returns the length of the string *str* in bytes.

```
length("string") ⇒ 6
```

`string dequote (string str)` [Built-in Function]

Removes '<' and '>' surrounding *str*. If *str* is not enclosed by angle brackets or these are unbalanced, the argument is returned unchanged:

```
dequote("<root@gnu.org.ua>") ⇒ "root@gnu.org.ua"
dequote("root@gnu.org.ua") ⇒ "root@gnu.org.ua"
dequote("there>") ⇒ "there>"
```

`string localpart (string str)` [Built-in Function]

Returns the local part of *str* if it is a valid email address, otherwise returns *str* unchanged.

```
localpart("gray") ⇒ "gray"
localpart("gray@gnu.org.ua") ⇒ "gray"
```

`string replstr (string s, number n)` [Built-in Function]

Replicate a string, i.e. return a string, consisting of *s* repeated *n* times:

```
replstr("12", 3) ⇒ "121212"
```

`string revstr (string s)` [Built-in Function]

Returns the string composed of the characters from *s* in reversed order:

```
revstr("foobar") ⇒ "raboof"
```

`number rindex (string s, string t)` [Built-in Function]
`number rindex (string s, string t, number start)` [Built-in Function]

Returns the index of the last occurrence of the string *t* in the string *s*, or -1 if *t* is not present.

```
rindex("string of rings", "ring") ⇒ 10
```

Optional argument *start*, if supplied, indicates the position in string where to start searching. E.g.:

```
rindex("string of rings", "ring", 10) ⇒ 2
```

See also [String manipulation], page 130.

`string substr (string str, number start)` [Built-in Function]
`string substr (string str, number start, number length)` [Built-in Function]

Returns the at most *length*-character substring of *str* starting at *start*. If *length* is omitted, the rest of *str* is used.

If *length* is greater than the actual length of the string, the `e_range` exception is signalled.

```
substr("mailfrom", 4) ⇒ "from"
substr("mailfrom", 4, 2) ⇒ "fr"
```

`string substring (string str, number start, number end)` [Built-in Function]

Returns a substring of *str* between offsets *start* and *end*, inclusive. Negative *end* means offset from the end of the string. In other words, yo obtain a substring from *start* to the end of the string, use `substring(str, start, -1)`:

```
substring("mailfrom", 0, 3) ⇒ "mail"
substring("mailfrom", 2, 5) ⇒ "ilfr"
substring("mailfrom", 4, -1) ⇒ "from"
substring("mailfrom", 4, length("mailfrom") - 1) ⇒ "from"
substring("mailfrom", 4, -2) ⇒ "fro"
```

This function signals `e_range` exception if either *start* or *end* are outside the string length.

`string tolower (string str)` [Built-in Function]

Returns a copy of the string *str*, with all the upper-case characters translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

```
tolower("MAIL") ⇒ "mail"
```

`string toupper (string str)` [Built-in Function]

Returns a copy of the string *str*, with all the lower-case characters translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

```
toupper("mail") ⇒ "MAIL"
```

`string ltrim (string str[, string cset)` [Built-in Function]

Returns a copy of the input string *str* with any leading characters present in *cset* removed. If the latter is not given, white space is removed (spaces, tabs, newlines, carriage returns, and line feeds).

```
ltrim("  a string") ⇒ "a string"
ltrim("089", "0") ⇒ "89"
```

Note the last example. It shows how `ltrim` can be used to convert decimal numbers in string representation that begins with '`0`'. Normally such strings will be treated as representing octal numbers. If they are indeed decimal, use `ltrim` to strip off the leading zeros, e.g.:

```
set dayofyear ltrim(strftime('%j', time()), "0")
```

`string rtrim (string str[, string cset)` [Built-in Function]

Returns a copy of the input string *str* with any trailing characters present in *cset* removed. If the latter is not given, white space is removed (spaces, tabs, newlines, carriage returns, and line feeds).

`number vercmp (string a, string b)` [Built-in Function]

Compares two strings as `mailfromd` version numbers. The result is negative if *b* precedes *a*, zero if they refer to the same version, and positive if *b* follows *a*:

```
vercmp("5.0", "5.1") ⇒ 1
vercmp("4.4", "4.3") ⇒ -1
vercmp("4.3.1", "4.3") ⇒ -1
vercmp("8.0", "8.0") ⇒ 0
```

`string sa_format_score (number code, number prec)` [Library Function]
Format *code* as a floating-point number with *prec* decimal digits:

```
sa_format_score(5000, 3) ⇒ "5.000"
```

This function is convenient for formatting SpamAssassin scores for use in message headers and textual reports. It is defined in module `sa.mfl`.

See [sa], page 186, for examples of its use.

`string sa_format_report_header (string text)` [Library Function]
Format a SpamAssassin report text in order to include it in a RFC 822 header. This function selects the score listing from *text*, and prefixes each line with '`* `'. Its result looks like:

```
*  0.2 NO_REAL_NAME           From: does not include a real name
*  0.1 HTML_MESSAGE           BODY: HTML included in message
```

See [sa], page 186, for examples of its use.

`string strip_domain_part (string domain, number n)` [Library Function]
Returns at most *n* last components of the domain name *domain*. If *n* is 0 the function returns *domain*.

This function is defined in the module `strip_domain_part.mfl` (see Section 4.24 [Modules], page 108).

Examples:

```
require strip_domain_part
strip_domain_part("puszcza.gnu.org.ua", 2) ⇒ "org.ua"
strip_domain_part("puszcza.gnu.org.ua", 0) ⇒ "puszcza.gnu.org.ua"
```

`string verp_extract_user (string email, string domain)` [Library Function]
If *email* is a valid VERP-style email address for *domain*, that corresponds to a valid local user name (see [validuser], page 190), this function returns the local user name, corresponding to that email. Otherwise, it returns empty string.

For example, assuming the local user '`gray`' exists:

```
verp_extract_user("gray=gnu.org.ua@tuhs.org", 'gnu\..*')
  ⇒ "gray"
```

5.5 String formatting

`string sprintf (string format, ...)` [Built-in Function]
The function `sprintf` formats its argument according to *format* (see below) and returns the resulting string. It takes varying number of parameters, the only mandatory one being *format*.

Format string

The format string is a simplified version of the format argument to C `printf`-family functions.

The format string is composed of zero or more *directives*: ordinary characters (not '`%`'), which are copied unchanged to the output stream; and *conversion specifications*, each of

which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character '`%`', and ends with a conversion specifier. In between there may be (in this order) zero or more *flags*, an optional *minimum field width*, and an optional *precision*.

Notice, that in practice that means that you should use single quotes with the *format* arguments, to protect conversion specifications from being recognized as variable references (see [singe-vs-double], page 60).

No type conversion is done on arguments, so it is important that the supplied arguments match their corresponding conversion specifiers. By default, the arguments are used in the order given, where each '`*`' and each conversion specifier asks for the next argument. If insufficiently many arguments are given, `sprintf` raises '`e_range`' exception. One can also specify explicitly which argument is taken, at each place where an argument is required, by writing '`%`*m*`$`', instead of '`%`' and '`*`*m*`$`' instead of '`*`', where the decimal integer *m* denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
sprintf('%*d', width, num);
```

and

```
sprintf('%2$*1$d', width, num);
```

are equivalent. The second style allows repeated references to the same argument.

Flag characters

The character '`%`' is followed by zero or more of the following *flags*:

'`#`' The value should be converted to an *alternate form*. For '`o`' conversions, the first character of the output string is made zero (by prefixing a '`0`' if it was not zero already). For '`x`' and '`X`' conversions, a non-zero result has the string '`0x`' (or '`0X`' for '`X`' conversions) prepended to it. Other conversions are not affected by this flag.

'`0`' The value should be zero padded. For '`d`', '`i`', '`o`', '`u`', '`x`', and '`X`' conversions, the converted value is padded on the left with zeros rather than blanks. If the '`0`' and '`-`' flags both appear, the '`0`' flag is ignored. If a precision is given, the '`0`' flag is ignored. Other conversions are not affected by this flag.

'`-`' The converted value is to be left adjusted on the field boundary. (The default is right justification.) The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A '`-`' overrides a '`0`' if both are given.

'` ` (a space)'
A blank should be left before a positive number (or empty string) produced by a signed conversion.

'`+`' A sign ('`+`' or '`-`') always be placed before a number produced by a signed conversion. By default a sign is used only for negative numbers. A '`+`' overrides a space if both are used.

Field width

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write '`*`' or '`*`*m*`$`' (for some decimal integer *m*) to specify that the field width is given in the next argument, or in the *m*-th argument, respectively, which must be of numeric type. A negative field width is taken as a '`-`' flag followed by a positive field width. In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Precision

An optional precision, in the form of a period ('`.`') followed by an optional decimal digit string. Instead of a decimal digit string one may write '`*`' or '`*`*m*`$`' (for some decimal integer *m*) to specify that the precision is given in the next argument, or in the *m*-th argument, respectively, which must be of numeric type. If the precision is given as just '`.`', or the precision is negative, the precision is taken to be zero. This gives the minimum number of digits to appear for '`d`', '`i`', '`o`', '`u`', '`x`', and '`X`' conversions, or the maximum number of characters to be printed from a string for the '`s`' conversion.

Conversion specifier

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d
i
: The numeric argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is '`1`'. When '`0`' is printed with an explicit precision '`0`', the output is empty.

o
u
x
X
: The numeric argument is converted to unsigned octal ('`o`'), unsigned decimal ('`u`'), or unsigned hexadecimal ('`x`' and '`X`') notation. The letters '`abcdef`' are used for '`x`' conversions; the letters '`ABCDEF`' are used for '`X`' conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is '`1`'. When '`0`' is printed with an explicit precision 0, the output is empty.

s
: The string argument is written to the output. If a precision is specified, no more than the number specified of characters are written.

%
: A '`%`' is written. No argument is converted. The complete conversion specification is '`%%`'.

5.6 Character Type

These functions check whether all characters of *str* fall into a certain character class according to the '`C`' ('`POSIX`') locale[2]. '`True`' (1) is returned if they do, '`false`' (0) is returned otherwise. In the latter case, the global variable `ctype_mismatch` is set to the index of the first character that is outside of the character class (characters are indexed from 0).

`boolean isalnum (string str)` [Built-in Function]
Checks for alphanumeric characters:

```
isalnum("a123") ⇒ 1
isalnum("a.123") ⇒ 0 (ctype_mismatch = 1)
```

`boolean isalpha (string str)` [Built-in Function]
Checks for an alphabetic character:

```
isalnum("abc") ⇒ 1
isalnum("a123") ⇒ 0
```

`boolean isascii (string str)` [Built-in Function]
Checks whether all characters in *str* are 7-bit ones, that fit into the ASCII character set.

```
isascii("abc") ⇒ 1
isascii("ab\0200") ⇒ 0
```

`boolean isblank (string str)` [Built-in Function]
Checks if *str* contains only blank characters; that is, spaces or tabs.

`boolean iscntrl (string str)` [Built-in Function]
Checks for control characters.

`boolean isdigit (string str)` [Built-in Function]
Checks for digits (0 through 9).

`boolean isgraph (string str)` [Built-in Function]
Checks for any printable characters except spaces.

`boolean islower (string str)` [Built-in Function]
Checks for lower-case characters.

`boolean isprint (string str)` [Built-in Function]
Checks for printable characters including space.

`boolean ispunct (string str)` [Built-in Function]
Checks for any printable characters which are not a spaces or alphanumeric characters.

`boolean isspace (string str)` [Built-in Function]
Checks for white-space characters, i.e.: space, form-feed ('`\f`'), newline ('`\n`'), carriage return ('`\r`'), horizontal tab ('`\t`'), and vertical tab ('`\v`').

`boolean isupper (string str)` [Built-in Function]
Checks for uppercase letters.

[2] Support for other locales is planned for future versions.

`boolean isxdigit (string str)` [Built-in Function]
Checks for hexadecimal digits, i.e. one of '`0`', '`1`', '`2`', '`3`', '`4`', '`5`', '`6`', '`7`', '`8`', '`9`', '`a`', '`b`', '`c`', '`d`', '`e`', '`f`', '`A`', '`B`', '`C`', '`D`', '`E`', '`F`'.

5.7 I/O functions

MFL provides a set of functions for writing to disk files, pipes or sockets and reading from them. The idea behind them is the same as in most other programming languages: first you open the resource with a call to `open` which returns a *descriptor* i.e. an integer number uniquely identifying the resource. Then you can write or read from it using this descriptor. Finally, when the resource is no longer needed, you can close it with a call to `close`.

The number of available resource descriptors is limited. The default limit is 1024. You can tailor it to your needs using the `max-streams` runtime configuration statement. See Section 7.12 [conf-runtime], page 231, for a detailed description.

By default, all I/O operations are unbuffered. This can be changed by setting the following global variables:

`number io_buffering` [Built-in variable]
Sets the default buffering type. Allowed values are (symbolic names are defined in `status.mfl` module):

`0`
`BUFFER_NONE`
No buffering. This is the default.

`1`
`BUFFER_FULL`
Full buffering. Size of the buffer is set by the `io_buffer_size` global variable (see below).

`2`
`BUFFER_LINE`
Line buffering. When reading, it is pretty much the same as `BUFFER_FULL`. When writing, the data are accumulated in buffer and actually sent to the underlying transport stream when the newline character is seen.

The initial size of the buffer is set by the `io_buffer_size` variable. It will grow as needed during the I/O.

`number io_buffer_size` [Built-in variable]
Set the buffer size if `io_buffering` is set to `BUFFER_FULL` or `BUFFER_LINE`. By default, this variable is set to the size of the system page.

`number open (string name)` [Built-in Function]
The *name* argument specifies the name of a resource to open and the access rights you need to have on it. The function returns a descriptor of the opened stream, which can subsequently be used as an argument to other I/O operations.

Buffering mode for the opened stream is defined by the `io_buffering` and `io_buffer_size` global variables. It can be changed using the `setbuf` function (see [setbuf], page 142).

First symbols of *name* determine the type of the resource to be opened and the access mode:

'`>`' The rest of *name* is a name of a file. Open the file for read-write access. If the file exists, truncate it to zero length, otherwise create the file.

'`>>`' The rest of *name* is a name of a file. Open the file for appending (writing at end of file). The file is created if it does not exist.

'`|`' Treat the rest of *name* as the command name and its arguments. Run this command and open its standard input for writing. The standard error is closed before launching the program. This can be altered by using the following versions of this construct:

`|2>`null: *command*
Standard error is redirected to `/dev/null`.

`|2>`file:*name command*
Execute *command* with its standard error redirected to the file *name*. If the file exists, it will be truncated.

`|2>>`file:*name command*
Standard error of the *command* is appended to the file *name*. If file does not exist, it will be created.

The '`|2>null:`' construct described above is a shortcut for

```
|2>>file:/dev/null command
```

`|2>`syslog:*facility*[.*priority*] *command*
Standard error is redirected to the given syslog *facility* and, optionally, *priority*. If the latter is omitted, '`LOG_ERR`' is assumed.

Valid values for *facility* are: '`user`', '`daemon`', '`auth`', '`authpriv`', '`mail`', and '`local0`' through '`local7`'. Valid values for *priority* are: '`emerg`', '`alert`', '`crit`', '`err`', '`warning`', '`notice`', '`info`', '`debug`'. Both *facility* and *priority* may be given in upper, lower or mixed cases.

Notice, that no whitespace characters are allowed between '`|`' and '`2>`'.

'`|<`' Treat the rest of *name* as the command name and its arguments. Run this command with its stdin closed and stdout open for reading.

The standard error is treated as described above (see '`|`').

'`|&`' Treat the rest of *name* as the command name and its arguments. Run this command and set up for two-way communication with it, i.e writes to the descriptor returned by `open` will send data to the program's standard input, reads from the descriptor will get data from the program's standard output.

The standard error is treated as described above (see '`|`'). For example, the following redirects it to syslog '`mail.debug`':

```
|&2>syslog:mail.debug command
```

'@' Treat the rest of *name* as the URL of a socket to connect to. Valid URL forms are described in [milter port specification], page 220.

If none of these prefixes is used, *name* is treated as a name of an existing file and `open` will attempt to open this file for reading.

The `open` function will signal exception `e_failure` if it is unable to open the resource or get the required access to it.

`number spawn (string` *cmd* `[, number` *in*`, number` *out*`, number` *err*`])` [Built-in Function]

Runs the supplied command *cmd*. The syntax of the *cmd* is the same as for the *name* argument to `open` (see above), which begins with '`|`', excepting that the '`|`' sign is optional. That is:

```
spawn("/bin/cat")
```

has exactly the same effect as

```
open("|/bin/cat")
```

Optional arguments specify file stream descriptors to be used for the program standard input, output and error streams, correspondingly. If supplied, these should be the values returned by a previous call to `open` or `tempfile`. The value '`-1`' means no redirection.

Buffering mode for the opened stream is defined by the `io_buffering` and `io_buffer_size` global variables. It can be changed using the `setbuf` function (see [setbuf], page 142).

The example below starts the `awk` program with a simple expression as its argument and redirects the content of the file `/etc/passwd` to its standard input. The returned stream descriptor is bound to the command's standard output (see the description of '`|<`' prefix above). The standard error is closed:

```
number fd spawn("<awk -F: '{print $1}'", open("/etc/passwd"))
```

`void close (number` *rd*`)` [Built-in Function]

The argument *rd* is a resource descriptor returned by a previous call to `open`. The function `close` closes the resource and deallocates any memory associated with it.

`close` will signal `e_range` exception if *rd* lies outside of allowed range of resource descriptors. See Section 7.12 [conf-runtime], page 231.

Notice that you are not required to close resources opened by `open`. Any unclosed resource will be closed automatically upon the termination of the filtering program.

`void shutdown (number` *rd*`, number` *how*`)` [Built-in Function]

This function causes all or part of a full-duplex connection to be closed. The *rd* must be either a socket descriptor (returned by `open(@...)`) or a two-way pipe socket descriptor (returned by `open(|&...)`), otherwise the call to `shutdown` is completely equivalent to `close`.

The `how` argument identifies which part of the connection to shut down:

SHUT_RD
Read connection. All further receptions will be disallowed.

SHUT_WR
: Write connection. All further transmissions will be disallowed.

SHUT_RDWR
: Shut down both read and write parts.

`number tempfile ([string tmpdir])` [Built-in Function]
: Creates a nameless temporary file and returns its descriptor. Optional *tmpdir* supplies the directory where to create the file, instead of the default `/tmp`.

`void rewind (number rd)` [Built-in Function]
: Rewinds the stream identified by *rd* to its beginning.

`number copy (number dst, number src)` [Built-in Function]
: Copies all data from the stream *src* to *dst*. Returns number of bytes copied.

The following functions provide basic read/write capabilities.

`void write (number rd, string str [, number size])` [Built-in Function]
: Writes the string *str* to the resource descriptor *rd*. If the *size* argument is given, writes this number of bytes.

  This function always attempts to write the requested amount of data. It will signal `e_range` exception if *rd* lies outside of allowed range of resource descriptors, `e_io` exception if an I/O error occurs, and `e_eof` exception if it wrote 0 bytes (e.g. because the underlying device is full).

`void write_body (number rd, pointer bp , number size)` [Built-in Function]
: Write the body segment of length *size* from pointer *bp* to the stream *rd*. This function can be used only in `prog body` (see [body handler], page 73). Its second and third arguments correspond exactly to the parameters of the `body` handler, so the following construct writes the message body to the resource `fd`, which should have been open prior to invoking the `body` handler:

```
prog body
do
  write_body(fd, $1, $2)
done
```

`string read (number rd, number n)` [Built-in Function]
: Read and return at most *n* bytes from the resource descriptor *rd*.

  If there are less than *n* bytes in the stream, the remaining bytes will be returned. Use `length()` to obtain the actual size of the returned data. If there are no bytes left, the `e_eof` exception will be signalled.

  The function may signal the following exceptions:

  e_range — *rd* lies outside of allowed range of resource descriptors.

  e_eof — End of file encountered.

  e_io — An I/O error occurred.

`string getdelim (number rd, string delim)` [Built-in Function]
Read and return the next string terminated by *delim* from the resource descriptor *rd*.

The terminating *delim* string will be removed from the return value.

When using this function, it is highly recommended to enable full buffering for *fd*, either by setting `io_buffering` before `open` (see [io_buffering], page 137) or by calling `setbuf` after it (see [setbuf], page 142). See [getline], page 141, for an example.

This function may signal the following exceptions:

e_range *rd* lies outside of allowed range of resource descriptors.

e_eof End of file encountered.

e_io An I/O error occurred.

`string getline (number rd)` [Built-in Function]
Read and return the next *line* from the resource descriptor *rd*. A line is any sequence of characters terminated with the default *line delimiter*. The default delimiter is a property of *rd*, i.e. different descriptors can have different line delimiters. The default value is '`\n`' (ASCII 10), and can be changed using the `fd_set_delimiter` function (see below).

When using this function, it is highly recommended to enable full buffering for *fd*, either by setting `io_buffering` before `open` (see [io_buffering], page 137) or by calling `setbuf` after it (see [setbuf], page 142), e.g.:

```
set fd open(input)
setbuf(fd, BUFFER_FULL)
set line getline(fd)
...
```

This function may signal the following exceptions:

e_range *rd* lies outside of allowed range of resource descriptors.

e_eof End of file encountered.

e_io An I/O error occurred.

`void fd_set_delimiter (number fd, string delim)` [Built-in Function]
Set new line delimiter for the descriptor *fd*, which must be in opened state.

Default delimiter is a newline character (ASCII 10). The following example shows how to change it to CRLF sequence:

```
fd_set_delimiter(fd, "\r\n")
```

`string fd_delimiter (number fd)` [Built-in Function]
Returns the line delimiter string for *fd*.

The following example shows how `mailfromd` I/O functions can be used to automatically add IP addresses to an RBL zone:

```
set nsupdate_cmd
  "/usr/bin/nsupdate -k /etc/bind/Kmail.+157+14657.private"

func block_address(string addr)
do
  number fd
  string domain

  set fd open "|%nsupdate_cmd"

  set domain reverse_ipstr(addr) . ".rbl.myzone.come"
  write(fd, "prereq nxrrset %domain A\n"
            "update add %domain 86400 A %addr\n\n"
done
```

The function `reverse_ipstr` is defined in Section 5.23 [Internet address manipulation functions], page 164.

`void setbuf(number` *fd*`, [number` *type*`, number` *size*`])` [Built-in Function]
Changes the buffering mode of *fd* according to the remaining two arguments. The *type* specifies buffering type (see [io_buffering], page 137), and *size* supplies the buffer size for buffering types `BUFFER_FULL` and `BUFFER_LINE`. If *size* is omitted, it defaults to `io_buffer_size` (see [io_buffer_size], page 137). Omitted *type* defaults to `io_buffering` (see [io_buffering], page 137).

`number getbuftype(number` *fd*`)` [Built-in Function]
Returns the type of buffering currently in effect for the descriptor *fd*. See [io_buffering], page 137, for a list of possible return values.

If this function returns `BUFFER_FULL` or `BUFFER_LINE`, you can use `getbufsize` to get the associated buffer size.

`number getbufsize(number` *fd*`)` [Built-in Function]
Returns the buffer size for the descriptor *fd*.

5.8 Filtering functions

This section describes functions that transform data using Mailutils *filter pipes*. Filter pipe is a string defining data flow between several *filters*. Each filter takes input, transforms it according to certain rules and produces the transformed data on its output. As in shell, multiple filters are connected using pipe characters (‘`|`’). For example, the `crlf` filter inserts a carriage return character before each newline character. A filter doing that kind of transformation is defined as:

```
"crlf"
```

Another filter, `base64`, converts its input to a BASE64 encoded string. To transform each newline into carriage return + newline pair and encode the resulting stream in BASE64, one would write:

```
"crlf | base64"
```

Some filters take one or more *arguments*. These are specified as a comma-delimited list in parentheses after the filter name. For example, the `linelen` filter limits the length of each output line to the given number of octets. The following filter pipe will limit the length of base64 lines in the filter above to 62 octets:

```
"crlf | base64 | linelen(62)"
```

Many filters operate in two modes: *encode* and *decode*. By default all MFL functions apply filters in encode mode. The desired mode can be stated explicitly in the filter string by using `encode()` and `decode()` functions. They take a filter pipe line as their argument. For example, the following will decode the stream produced by the example filter above:

```
"decode(base64 | crlf)"
```

See Section 5.8.1 [Filters], page 143, for a discussion of available filters and their arguments.

`string filter_string (string` *`input`*`, string` *`filter_pipe`*`)` [Built-in Function]

Transforms the string *input* using filters in *filter_pipe* and returns the result. Example:

```
set input "test\ninput\n"
filter_string(input, "crlf|base64") ⇒ "dGVzdA0KaW5wdXQNCg=="
```

`void filter_fd (number` *`src_fd`*`, number` *`dst_fd`*`, string` *`filter_pipe`*`)` [Built-in Function]

Given two I/O descriptors, reads data from *src_fd*, transforms it using *filter_pipe* and writes the result to descriptor *dst_fd*.

Both descriptors must be obtained using functions described in Section 5.7 [I/O functions], page 137.

5.8.1 Filters and Filter Pipes

A *filter pipe* is a string consisting of filter invocations delimited by pipe characters ('`|`'). Each invocation is a filter name optionally followed by a comma-separated list of parameters. Most filters can operate in two modes: *encode* and *decode*. Unless specified otherwise, filters are invoked in encode mode. To change the mode, the `encode` and `decode` meta-filters are provided. Argments to these filters are filter pipes that will be executed in the corresponding mode.

The following Mailutils filters are available:

`7bit` [Filter]

In encode mode, converts its input into 7-bit ASCII, by clearing the 8th bit on each processed byte.

In decode mode, it operates exactly as the 8bit filter, i.e. copies its input to the output verbatim.

The filter takes no arguments.

`8bit` [Filter]

`binary` [Filter]

Copies its input to output verbatim.

`base64` [Filter]
`B` [Filter]

Encodes or decodes the input using the `base64` encoding.

The only difference between `BASE64` and `B` is that, in encode mode, the former limits each ouput line length to 76 octets, whereas the latter produces a contiguous stream of base64 data.

In decode mode, both filters operate exactly the same way.

`charset (`*`cset`*`)` [Filter]
`charset (`*`cset`*`, `*`fallback`*`)` [Filter]

A convenience interface to the `iconv` filter, available for use only in the `message_body_to_stream` function. It decodes the part of a MIME message from its original character set, which is determined from the value of the `Content-Type` header, to the destination character set *cset*. Optional *fallback* parameter specifies the representation fallback to be used for octets that cannot be converted between the charater sets. Its use is described in See [iconv], page 145.

This filter is normally takes its input from the `mimedecode` filter, as in:

```
message_body_to_stream(fd, msg, 'mimedecode|charset(utf-8)')
```

See [mimedecode], page 159, for a detailed discussion.

`crlf` [Filter]
`rfc822` [Filter]

Converts line separators from LF (ASCII 10) to CRLF (ASCII 13 10) and vice-versa.

In decode mode, translates each CRLF to LF. Takes no arguments.

In encode mode, translates each LF to CRLF. If an optional argument '`-n`' is given, produces a *normalized* output, by preserving each input CRLF sequence untouched (otherwise such sequences will be are translated to CR CR LF).

`crlfdot` [Filter]

In encode mode, replaces each LF ('`\n`' or ASCII 10) character with CRLF ('`\r\n`', ASCII 13 10), and *byte-stuffs* the output by producing an additional '`.`' in front of any '`.`' appearing at the beginning of a line in input. Upon end of input, it outputs additional '`.\r\n`', if the last output character was '`\n`', or '`\r\n.\r\n`' otherwise.

If supplied the '`-n`' argument, it preserves each CRLF input sequence untranslated (see the `CRLF` above).

In decode mode, the reverse is performed: each CRLF is replaced with a single LF byte, and additional dots are removed from beginning of lines. A single dot on a line by itself marks the end of the stream and causes the filter to return EOF.

`dot` [Filter]

In encode mode, *byte-stuffs* the input by outputting an additional dot ('`.`') in front of any dot appearing at the beginning of a line. Upon encountering end of input, it outputs additional '`.\n`'.

In decode mode, the reverse is performed: additional dots are removed from beginning of lines. A single dot on a line by itself (i.e. the sequence '`\n.\n`') marks the end of the stream and causes the filter to return EOF.

This filter doesn't take arguments.

`from` [Filter]

Performs a traditional UNIX processing of lines starting with a '`From`' followed by a space character.

In encode mode, each '`From ` at the beginning of a line is replaced by '`>From `'.

In decode mode, the reverse operation is performed: initial greater-then sign ('`>`') is removed from any line starting with '`>From `'.

The filter takes no arguments.

`fromrd` [Filter]

MBOXRD-compatible processing of envelope lines.

In encode mode, each '`From ` optionally preceded by any number of contiguous '`>`' characters and appearing at the beginning of a line is prefixed by another '`>`' character on output.

In decode mode, the reverse operation is performed: initial greater-then sign ('`>`') is removed from any line starting with one or more '`>`' characters followed by '`From `'.

`header` [Filter]

This filter treats its input as a RFC-2822 email message. It extracts its header part (i.e. everything up to the first empty line) and copies it to the output. The body of the message is ignored.

The filter operates only in decode mode and takes no arguments.

`iconv` (*`src`*, *`dst`* [, *`fallback`*]) [Filter]

Converts input from character set *src* to *dst*. The filter works the same way in both decode and encode modes.

It takes two mandatory arguments: the names of the input (*src*) and output (*dst*) charset. Optional third argument specifies what to do when an illegal character sequence is encountered in the input stream. Its possible values are:

`none` Raise a `e_ilseq` exception.

`copy-pass`
: Copy the offending octet to the output verbatim and continue conversion from the next octet.

`copy-octal`
: Print the offending octet to the output using the C octal conversion and continue conversion from the next octet.

The default is `copy-octal`.

The following example creates a `iconv` filter for converting from `iso-8859-2` to `utf-8`, raising the `e_ilseq` exception on the first conversion error:

```
iconv(iso-8859-2, utf-8, none)
```

`inline-comment` [Filter]

`inline-comment` (*`str`*, [*options*]) [Filter]

In decode mode, the filter removes from the input all lines beginning with a given *inline comment sequence str*. The default comment sequence is '`;`' (a semicolon).

The following options modify the default behavior:

`-i, str` Emit line number information after each contiguous sequence of removed lines. The argument *str* supplies an *information starter* – a sequence of characters which is output before the actual line number.

`-r` Remove empty lines, i.e. the lines that contain only whitespace characters.

`-s` Squeeze whitespace. Each sequence of two or more whitespace characters encountered on input is replaced by a single space character on output.

`-S` A *whitespace-must-follow* mode. A comment sequence is recognized only if followed by a whitespace character. The character itself is retained on output.

In encode mode the `inline-comment` filter adds a comment-starter sequence at the beginning of each line. The default comment-starter is '`;`' and can be changed by specifying the desired comment starter as the first argument.

The only option supported in this mode is `-S`, which enables the whitespace-must-follow mode, in which a single space character (ASCII 20) is output after each comment sequence.

`linecon` [Filter]
`linecon` (*-i*, *str*) [Filter]

Implements a familiar UNIX line-continuation facility. The filter removes from itsinput stream any newline character immediately preceded by a backslash. This filter operates only in decode mode.

If given the arguments ('`-i`', *str*), enables the *line number information facility*. This facility emits current input line number (prefixed with *str*) after each contiguous sequence of one or more removed newline characters. It is useful for implementing parsers which are normally supposed to identify eventual erroneous lines with their input line numbers.

`linelen` (*n*) [Filter]

Limits the length of each output line to a certain number of octets. It operates in encode mode only and requires a single parameter: the desired output length in octets. This filter makes no attempt to analyze the lexical structure of the input: the newline caracters are inserted when the length of the output line reaches a predefined maximum. Any newline characters present in the input are taken into account when computing the input line length.

`mimedecode` [Filter]

This is a domain-specific filter available for use only with the `message_body_to_stream` function. It decodes the part of a MIME message from whatever encoding that was used to store it in the message to a stream of bytes. See [mimedecode], page 159.

`quoted-printable` [Filter]
`Q` [Filter]

Encodes or decodes the input using the *quoted-printable* encoding.

`XML` [Filter]

In encode mode, the `xml` filter converts input stream (which must contain valid UTF-8 characters) into a form suitable for inclusion into a XML or HTML document, i.e. it replaces '`<`', '`>`', and '`&`' with '`<`', '`>`', and '`&`', correspondingly, and replaces invalid characters with their numeric character reference representation.

In decode mode, a reverse operation is performed.

The filter does not take arguments.

5.9 Email processing functions

`number email_map (string` *email*`)` [Built-in Function]

Parses *email* and returns a bitmap, consisting of zero or more of the following flags:

'`EMAIL_MULTIPLE`'
: *email* has more than one email address.

'`EMAIL_COMMENTS`'
: *email* has comment parts.

'`EMAIL_PERSONAL`'
: *email* has personal part.

'`EMAIL_LOCAL`'
: *email* has local part.

'`EMAIL_DOMAIN`'
: *email* has domain part.

'`EMAIL_ROUTE`'
: *email* has route part.

These constants are declared in the `email.mfl` module. The function `email_map` returns 0 if its argument is not a valid email address.

`boolean email_valid (string` *email*`)` [Library Function]

Returns '`True`' (1) if *email* is a valid email address, consisting of local and domain parts only. E.g.:

```
email_valid("gray@gnu.org") ⇒ 1
email_valid("gray") ⇒ 0
email_valid('"Sergey Poznyakoff <gray@gnu.org>') ⇒ 0
```

This function is defined in `email.mfl` (see Section 4.24 [Modules], page 108).

5.10 Envelope Modification Functions

Envelope modification functions set sender and add or delete recipient addresses from the message envelope. This allows MFL scripts to redirect messages to another addresses.

`void set_from (string` *email* `[, string` *args*`])` [Built-in Function]

Sets envelope sender address to *email*, which must be a valid email address. Optional *args* supply arguments to ESMTP '`MAIL FROM`' command.

`void rcpt_add (string address)` [Built-in Function]
Add the e-mail *address* to the envelope.

`void rcpt_delete (string address)` [Built-in Function]
Remove *address* from the envelope.

The following example code uses these functions to implement a simple alias-like capability:

```
prog envrcpt
do
   string alias dbget(aliasdb, $1, "NULL", 1)
   if alias != "NULL"
     rcpt_delete($1)
     rcpt_add(alias)
   fi
done
```

5.11 Header Modification Functions

There are two ways to modify message headers in a MFL script. First is to use header actions, described in Section 4.19.1 [Actions], page 92, and the second way is to use message modification functions. Compared with the actions, the functions offer a series of advantages. For example, using functions you can construct the name of the header to operate upon (e.g. by concatenating several arguments), something which is impossible when using actions. Moreover, apart from three basic operations (add, modify and remove), as supported by header actions, header functions allow to insert a new header into a particular place.

`void header_add (string name, string value)` [Built-in Function]
Adds a header '*name*: *value*' to the message.

In contrast to the `add` action, this function allows to construct the header name using arbitrary MFL expressions.

`void header_add (string name, string value, number idx)` [Built-in Function]
This syntax is preserved for backward compatibility. It is equivalent to `header_insert`, which see.

`void header_insert (string name, string value, number idx)` [Built-in Function]
This function inserts a header '*name*: 'value'' at *idx*th header position in the internal list of headers maintained by the MTA. That list contains headers added to the message either by the filter or by the MTA itself, but not the headers included in the message itself. Some of the headers in this list are conditional, e.g. the ones added by the 'H?*cond*?' directive in `sendmail.cf`. MTA evaluates them after all header modifications have been done and removes those of headers for which they yield false. This means that the position at which the header added by `header_insert` will appear in the final message will differ from *idx*.

`void header_delete (string` *name* `[, number` *index*`])` [Built-in Function]

Delete header *name* from the envelope. If *index* is given, delete *index*th instance of the header *name*.

Notice the differences between this function and the `delete` action:

1. It allows to construct the header name, whereas `delete` requires it to be a literal string.
2. Optional *index* argument allows to select a particular header instance to delete.

`void header_replace (string` *name*`, string` *value* `[,` `number` *index*`])` [Built-in Function]

Replace the value of the header *name* with *value*. If *index* is given, replace *index*th instance of header *name*.

Notice the differences between this function and the `replace` action:

1. It allows to construct the header name, whereas `replace` requires it to be a literal string.
2. Optional *index* argument allows to select a particular header instance to replace.

`void header_rename (string` *name*`, string` *newname*`[,` `number` *idx*`])` [Library Function]

Defined in the module `header_rename.mfl`.
Available only in the '`eom`' handler.

Renames the *idx*th instance of header *name* to *newname*. If *idx* is not given, assumes 1.

If the specified header or the *idx* instance of it is not present in the current message, the function silently returns. All other errors cause run-time exception.

The position of the renamed header in the header list is not preserved.

The example below renames '`Subject`' header to '`X-Old-Subject`':

```
require 'header_rename'

prog eom
do
  header_rename("Subject", "X-Old-Subject")
done
```

`void header_prefix_all (string` *name* `[, string` *prefix*`])` [Library Function]

Defined in the module `header_rename.mfl`.
Available only in the '`eom`' handler.

Renames all headers named *name* by prefixing them with *prefix*. If *prefix* is not supplied, removes all such headers.

All renamed headers will be placed in a continuous block in the header list. The absolute position in the header list will change. Relative ordering of renamed headers will be preserved.

`void header_prefix_pattern (string pattern, string prefix)` [Library Function]

Defined in the module `header_rename.mfl`.
Available only in the '`eom`' handler.

Renames all headers with names matching *pattern* (in the sense of `fnmatch`, see Section 4.17.7 [Special comparisons], page 86) by prefixing them with *prefix*.

All renamed headers will be placed in a continuous block in the header list. The absolute position in the header list will change. Relative ordering of renamed headers will be preserved.

If called with one argument, removes all headers matching *pattern*.

For example, to prefix all headers beginning with '`X-Spamd-`' with an additional '`X-`':

```
require 'header_rename'

prog eom
do
  header_prefix_pattern("X-Spamd-*", "X-")
done
```

5.12 Body Modification Functions

Body modification is an experimental feature of MFL. The version 9.2 provides only one function for that purpose.

`void replbody (string text)` [Built-in Function]

Replace the body of the message with *text*. Notice, that *text* must not contain RFC 822 headers. See the previous section if you want to manipulate message headers.

Example:

```
replbody("Body of this message has been removed by the mail filter.")
```

No restrictions are imposed on the format of *text*.

`void replbody_fd (number fd)` [Built-in Function]

Replaces the body of the message with the content of the stream *fd*. Use this function if the body is very big, or if it is returned by an external program.

Notice that this function starts reading from the current position in *fd*. Use `rewind` if you wish to read from the beginning of the stream.

The example below shows how to preprocess the body of the message using external program `/usr/bin/mailproc`, which is supposed to read the body from its standard input and write the processed text to its standard output:

```
number fd   # Temporary file descriptor

prog data
do
  # Open the temporary file
  set fd tempfile()
```

```
done

prog body
do
  # Write the body to it.
  write_body(fd, $1, $2)
done

prog eom
do
  # Use the resulting stream as the stdin to the mailproc
  # command and read the new body from its standard output.
  rewind(fd)
  replbody_fd(spawn("</usr/bin/mailproc", fd))
done
```

5.13 Message Modification Queue

Message modification functions described in the previous subsections do not take effect immediately, in the moment they are called. Instead they store the requested changes in the internal *message modification queue*. These changes are applied at the end of processing, before '`eom`' stage finishes (see Figure 3.1).

One important consequence of this way of operation is that calling any MTA action (see Section 4.19.1 [Actions], page 92), causes all prior modifications to the message to be ignored. That is because after receiving the action command, MTA will not call filter for that message any more. In particular, the '`eom`' handler will not be called, and the message modification queue will not be flushed. While it is logical for such actions as `reject` or `tempfail`, it may be quite confusing for `accept`. Consider, for example, the following code:

```
prog envfrom
do
  if $1 == ""
    header_add("X-Filter", "foo")
    accept
  fi
done
```

Obviously, the intention was to add a '`X-Filter`' header and accept the message if it was sent from the null address. What happens in reality, however, is a bit different: the message is accepted, but no header is added to it. If you need to accept the message and retain any modifications you have done to it, you need to use an auxiliary variable, e.g.:

```
number accepted 0
prog envfrom
do
  if $1 == ""
    header_add("X-Filter", "foo")
    set accepted 1
  fi
done
```

Then, test this variable for non-zero value at the beginning of each subsequent handler, e.g.:

```
prog data
do
  if accepted
    continue
  fi
  ...
done
```

To help you trace such problematic usages of `accept`, `mailfromd` emits the following warning:

```
RUNTIME WARNING near /etc/mailfromd.mfl:36: `accept' causes previous
message modification commands to be ignored; call mmq_purge() prior
to `accept', to suppress this warning
```

If it is OK to lose all modifications, call `mmq_purge`, as suggested in this message.

`void mmq_purge ()` [Built-in Function]

Remove all modification requests from the queue. This function undoes the effect of any of the following functions, if they had been called previously: `rcpt_add`, `rcpt_delete`, `header_add`, `header_insert`, `header_delete`, `header_replace`, `replbody`, `quarantine`.

5.14 Mail Header Functions

`string message_header_encode (string text, [string enc, string charset])` [Built-in Function]

Encode *text* in accordance with RFC 2047. Optional arguments:

enc Encoding to use. Valid values are '`quoted-printable`', or '`Q`' (the default) and '`base64`', or '`B`'.

charset Character set. By default '`UTF-8`'.

If the function is unable to encode the string, it raises the exception `e_failure`.

For example:

```
set string "Keld Jørn Simonsen <keld@dkuug.dk>"
message_header_encode(string, "ISO-8859-1")
  ⇒ "=?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>"
```

`string message_header_decode (string text, [string charset])` [Built-in Function]

text must be a header value encoded in accordance with RFC 2047. The function returns the decoded string. If the decoding fails, it raises `e_failure` exception. The optional argument *charset* specifies the character set to use (default – '`UTF-8`').

```
set string "=?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>"
message_header_decode(string)
 ⇒ "Keld Jørn Simonsen <keld@dkuug.dk>"
```

`string unfold (string text)` [Built-in Function]

If *text* is a "folded" multi-line RFC 2822 header value, unfold it. If *text* is a single-line string, return its unchanged copy.

For example, suppose that the message being processed contained the following header:

```
List-Id: Sent bugreports to
  <some-address@some.net>
```

Then, applying `unfold` to its value[3] will produce:

```
Sent bugreports to <some-address@some.net>
```

5.15 Mail Body Functions

`string body_string (pointer text, number count)` [Built-in Function]

Converts first *count* bytes from the memory location pointed to by *text* into a regular string.

This function is intended to convert the `$1` argument passed to a `body` handler to a regular MFL string. For more information about its use, see [body handler], page 73.

`bool body_has_nulls (pointer text, number count)` [Built-in Function]

Returns '`True`' if first *count* bytes of the string pointed to by *text* contain ASCII NUL characters.

Example:

```
prog body
do
  if body_has_nulls($1, $2)
    reject
  fi
done
```

5.16 EOM Functions

The following function is available only in the '`eom`' handler:

`void progress ()` [Built-in Function]

Notify the MTA that the filter is still processing the message. This causes MTA to restart its timeouts and allows additional amount of time for execution of '`eom`'.

Use this function if your '`eom`' handler needs additional time for processing the message (e.g. for scanning a very big MIME message). You may call it several times, if the need be, although such usage is not recommended.

---

[3] For example:

```
prog header
do
  echo unfold($2)
done
```

5.17 Current Message Functions

`number current_message ()` [Built-in Function]
This function can be used in `eom` handlers only. It returns a message descriptor referring to the current message. See Section 5.19 [Message functions], page 156, for a description of functions for accessing messages.

The functions below access the headers from the current message. They are available in the following handlers: `eoh`, `body`, `eom`.

`number current_header_count ([string name])` [Built-in Function]
Return number of headers in the current message. If *name* is specified, return number of headers that have this name.

```
current_header_count() ⇒ 6
current_header_count("Subject") ⇒ 1
```

`string current_header_nth_name (number n)` [Built-in Function]
Return the name of the *n*th header. The index *n* is 1-based.

`string current_header_nth_value (number n)` [Built-in Function]
Return the value of the *n*th header. The index *n* is 1-based.

`string current_header (string name [, number n])` [Built-in Function]
Return the value of the named header, e.g.:

```
set s current_header("Subject")
```

Optional second argument specifies the header instance, if there are more than 1 header of the same name, e.g.:

```
set s current_header("Received", 2)
```

Header indices are 1-based.

All current_header function raise the `e_not_found` exception if the requested header was not found.

5.18 Mailbox Functions

A set of functions is provided for accessing mailboxes and messages within them. In this subsection we describe the functions for accessing mailboxes.

A mailbox is opened using `mailbox_open` function:

`number mailbox_open (string url [, string mode,` [Built-in Function]
`string perms])`
Open a mailbox identified by *url*. Return a *mailbox descriptor*: a unique numeric identifier that can subsequently be used to access this mailbox.

The optional *mode* argument specifies the access mode for the mailbox. Its valid values are:

Value	Meaning
r	Open mailbox for reading. This is the default.

w	Open mailbox for writing. If the mailbox does not exist, it is created.
rw	Open mailbox for reading and writing. If the mailbox does not exist, it is created.
wr	Same as '`rw`'.
w+	Open mailbox for reading and writing. If the mailbox does not exist, it is created.
a	Open mailbox for appending messages to it. If the mailbox does not exist, an exception is signalled.
a+	Open mailbox for appending messages to it. If the mailbox does not exist, it is created.

The optional *perms* argument specifies the permissions to use in case a new file (or files) is created. It is a comma-separated list of:

```
[go](+|=)[wr]+
```

The initial letter controls which users' access is to be set: users in the file's group ('`g`') or other users not in the file's group ('`o`'). The following character controls whether the permissions are added to the default ones ('`+`') or applied instead of them ('`=`'). The remaining letters specify the permissions: '`r`' for read access and '`w`' for write access. For example:

```
g=rw,o+r
```

The number of mailbox descriptors available for simultaneous opening is 64. This value can be changed using the `max-open-mailboxes` runtime configuration statement (see Section 7.12 [conf-runtime], page 231).

`number mailbox_messages_count (number` *`nmbx`*`)` [Built-in Function]
Return the number of messages in mailbox. The argument *nmbx* is a valid mailbox descriptor as returned by a previous call to `mailbox_open`.

`number mailbox_get_message (number` *`mbx`*`, number` *`n`*`)` [Built-in Function]
Retrieve *n*th message from the mailbox identified by descriptor *mbx*. On success, the function returns a *message descriptor*, an integer number that can subsequently be used to access that message (see Section 5.19 [Message functions], page 156). On error, an exception is raised.

Messages in a mailbox are numbered starting from 1.

`void mailbox_close (number` *`nmbx`*`)` [Built-in Function]
Close a mailbox previously opened by `mailbox_open`.

`void mailbox_append_message (number` *`nmbx`*`, number` *`nmsg`*`)` [Built-in Function]
Append message *nmsg* to mailbox *nmbx*. The message descriptor *nsmg* must be obtained from a previous call to `mailbox_get_message` or `current_message` (see [current_message], page 154).

5.19 Message Functions

The functions described below retrieve information from RFC822 messages. The message to operate upon is identified by its *descriptor*, an integer number returned by the previous call to `mailbox_get_message` (see Section 5.18 [Mailbox functions], page 154) or `current_message` (see [current_message], page 154) function. The maximum number of message descriptors is limited by 1024. You can change this limit using the `max-open-messages` runtime configuration statement (see Section 7.12 [conf-runtime], page 231).

`number message_size (number nmsg)` [Built-in Function]
Return the size of the message *nmsg*, in bytes. *Notice*, that if *nmsg* refers to current message (see [current_message], page 154), the returned value is less than the size seen by the MTA, because `mailfromd` recodes CR-LF sequences to LF, i.e. removes carriage returns (ASCII 13) occurring before line feeds (ASCII 10. To obtain actual message length as seen by the MTA, add the number of lines in the message:

```
set actual_length message_size(nmsg) + message_lines(nmsg)
```

`boolean message_body_is_empty (number nmsg)` [Built-in Function]
Returns `true` if the body of message *nmsg* has zero size or contains only whitespace characters. If the '`Content-Transfer-Encoding`' header is present, it is used to decode body before processing.

`void message_close (number nmsg)` [Built-in Function]
Close the message identified by descriptor *nmsg*.

`number message_lines (number nmsg)` [Built-in Function]
Return total number of lines in message *nmsg*. The following relation holds true:

```
message_lines(x) = message_body_lines(x)
                           + message_header_lines(x) + 1
```

`string message_read_line (number nmsg)` [Built-in Function]
Read and return next line from the message *nmsg*. If there are no more lines to read, raise the `eof` exception.

Use `message_rewind` to rewind the message stream and read its contents again.

`void message_rewind (number nmsg)` [Built-in Function]
Rewind the stream associated with message referred to by descriptor *nmsg*.

`number message_from_stream (number fd; string filter_chain)` [Built-in Function]
Converts contents of the stream identified by *fd* to a mail message. Returns identifier of the created message.

Optional *filter_chain* supplies the name of a *Mailutils filter chain*, through which the data will be passed before converting. See `http://mailutils.org/wiki/Filter_chain`, for a description of filter chains.

`void message_to_stream (number fd,number nmsg; string filter_chain)` [Built-in Function]
Copies message *nsmg* to stream descriptor *fd*. The descriptor must be obtained by a previous call to `open`.

Optional *filter_chain* supplies the name of a *Mailutils filter chain*, through which the data will be passed before writing them to *fd*. See `http://mailutils.org/wiki/Filter_chains`, for a description of filter chains.

5.19.1 Header functions

`number message_header_size (number nmsg)` [Built-in Function]
Return the size, in bytes of the headers of message *nmsg*. See the note to the `message_size`, above.

`number message_header_lines (number nmsg)` [Built-in Function]
Return number of lines occupied by headers in message *nmsg*.

`number message_header_count (number nmsg, [string name])` [Built-in Function]
Return number of headers in message *nmsg*.

If *name* is supplied, count only headers with that name.

`string message_find_header (number nmsg, string name [, number idx])` [Built-in Function]
Return value of header *name* from the message *nmsg*. If the message contains several headers with the same name, optional parameter *idx* may be used to select one of them. Headers are numbered from '`1`'.

If no matching header is not found, the `not_found` exception is raised. If another error occurs, the `failure` exception is raised.

The returned string is a verbatim copy of the message contents (except for eventual CR-LF -> LF translation, see above). You might need to apply the `unfold` function to it (see Section 5.14 [Mail header functions], page 152).

`string message_nth_header_name (number nmsg, number n)` [Built-in Function]
Returns the name of the *n*th header in message *nmsg*. If there is no such header, `e_range` exception is raised.

`string message_nth_header_value (number msg, number n)` [Built-in Function]
Returns the value of the *n*th header in message *nmsg*. If there is no such header, `e_range` exception is raised.

`boolean message_has_header (number nmsg,string name [, number idx])` [Built-in Function]
Return `true` if message *nmsg* contains header with the given *name*. If there are several headers with the same name, optional parameter *idx* may be used to select one of them.

5.19.2 Message body functions

`number message_body_size (number nmsg)` [Built-in Function]
Return the size, in bytes, of the body of message *nmsg*. See the note to the `message_size`, above.

`number message_body_lines (number nmsg)` [Built-in Function]
Return number of lines in the body of message referred to by descriptor *nmsg*.

`void message_body_rewind (number nmsg)` [Built-in Function]
Rewind the stream associated with the body of message referred to by descriptor *nmsg*.

A call to `message_body_read_line` (see below) after calling this function will return the first line from the message body.

`string message_read_body_line (number nmsg)` [Built-in Function]
Read and return next line from the body of the message *nmsg*. If there are no more lines to read, raise the `eof` exception.

Use `message_body_rewind` (see above) to rewind the body stream and read its contents again.

`void message_body_to_stream (number fd,number nmsg;` [Built-in Function]
`string filter_pipe)`
Copies the body of the message *nsmg* to stream descriptor *fd*. The descriptor must be obtained by a previous call to `open`.

Optional *filter_pipe* supplies a sequence of *Mailutils filters*, through which the data will be passed before writing them to *fd*. See Section 5.8 [Filtering functions], page 142, for a discussion of filter pipe syntax.

In addition to filters described in See Section 5.8.1 [Filters], page 143, two special filters are provided for use with this function: `mimedecode` and `charset`. The `mimedecode` filter instructs the function to decode the message body by reverting the encoding specified by its `Content-Transfer-Encoding` header. It is normally used as the very first filter in chain. The `charset` filter recodes the message body from it original character set to the character set specified as its argument.

See [mimedecode], page 159, for a detailed discussion of this feature.

5.19.3 MIME functions

`boolean message_is_multipart (number nmsg)` [Built-in Function]
Return `true` if message *nmsg* is a multipart (MIME) message.

`number message_count_parts (number nmsg)` [Built-in Function]
Return number of parts in message *nmsg*, if it is a multipart (MIME) message. If it is not, return '`1`'.

Use `message_is_multipart` to check whether the message is a multipart one.

`number message_get_part (number nmsg, number n)` [Built-in Function]
Extract *n*th part from the multipart message *nmsg*. Numeration of parts begins from '`1`'. Return message descriptor referring to the extracted part. Message parts are regarded as messages, so any message functions can be applied to them.

`string message_content_type (number nmsg)` [Built-in Function]
Returns content type for the message *nmsg*. The returned string is composed of content type and subtype, delimited by slash.

If *nmsg* is not a multipart message, the function returns '`text/plain`'.

Several functions are provided for decoding multi-part messages. Such decoding is governed by `Content-Transfer-Encoding` and `Content-Type` headers of the message. The `Content-Transfer-Encoding` header defines the method used to encode the message. The value of `Content-Type` header is used to determine the character set the body is written in.

Basic MIME decoding facilities are provided by the built-in function `message_body_to_stream`, described in the previous subsection. To instruct it to decode the content, pass it the *filter_chain* argument beginning with the word `mimedecode`. The usual sequence is:

```
set fd open("> outfile")
message_body_to_stream(fd, msg, "mimedecode")
```

To ensure that the produced stream is represented in a specific character set, use the `charset` special filter. Its argument is the name of the character set to recode the text to:

```
set fd open("> outfile")
message_body_to_stream(fd, msg, "mimedecode|charset(utf-8)")
```

The `charset` filter takes also an optional second argument – a *fallback* method, specifying what to do when an octet sequence is encountered that cannot be represented in the requested character set. Possible values for this argument are:

'`none`' Stop further conversion and signal the `e_ilseq` exception.

'`copy-pass`'
Copy the offending character to the output verbatim.

'`copy-octal`'
Represent the offending character as a C octal sequence ('`\`*nnn*', where *n* is an octal digit). This is the default.

To decode a particular part of the message, first extract it using the `message_get_part` function. Recall that message parts are messages as well, and as such can be passed to `message_body_to_stream`. For example, the following code fragment extracts all top-level parts of a multi-part message to files named '`part.`*N*':

```
if message_is_multipart(msg)
  set n message_count_parts(msg)
  loop for set i 1, while i <= n, set i i + 1
  do
    set fd open("> part.%i")
    message_body_to_stream(fd, message_get_part(msg, i), "mimedecode")
    close(fd)
  done
fi
```

The `mime.mfl` module provides additional functions for decoding multi-part messages:

`number message_body_decode (number` *`nmsg`*`; string` *`charset`*`, string` *`fallback`*`)` [Library Function]
Decodes the body of the message (or message part) *nmsg*, optionally converting it to the given *charset*. The *fallback* argument specifies what to do if a byte sequence cannot be converted to the specified character set. See [iconv fallback], page 145, for a detailed discussion.

The function returns a descriptor of the I/O stream that contains the decoded material. See Section 5.7 [I/O functions], page 137, for a discussion of functions available for reading from it.

`number message_part_decode(number nmsg, number part; string charset, string fallback)` [Library Function]

Decodes the body of the given part of a MIME message *nmsg*. The argument *part* is a 1-based index of the part in the message. Optional arguments *charset* and *fallback* have the same meaning as in `message_body_decode` (see above).

Returns a descriptor of the I/O stream that contains the decoded material.

This function is equivalent to:

```
message_body_decode(message_get_part(nmsg, part, charset,
                    fallback))
```

5.19.4 Message digest functions

Message digests are specially formatted messages that contain certain number of mail messages, encapsulated using the method described in RFC 934. Such digests are often used in mailing lists to reduce the frequency of sending mails. Messages of this format are also produced by the *forward* function in most MUA's.

The usual way to handle a message digest in MFL is to convert it first to a MIME message, and then to use functions for accessing its parts (see Section 5.19.3 [MIME functions], page 158).

`number message_burst (number nmsg ; number flags)` [Built-in Function]

Converts the message identified by the descriptor *nmsg* to a multi-part message. Returns a descriptor of the created message.

Optional argument *flags* controls the behavior of the bursting agent. It is a bitwise OR of error action and bursting flags.

Error action defines what to do if a part of the digest is not in RFC822 message format. If it is '`BURST_ERR_FAIL`' (the default), the function will raise the '`e_format`' exception. If *onerr* is '`BURST_ERR_IGNORE`', the improperly formatted part will be ignored. Finally, the value '`BURST_ERR_BODY`' instructs `message_burst` to create a replacement part with empty headers and the text of the offending part as its body.

Bursting flags control various aspects of the agent behavior. Currently only one flag is defined, '`BURST_DECODE`', which instructs the agent to decode any MIME parts (according to the '`Content-Transfer-Encoding`' header) it encounters while bursting the message.

Parts of a message digest are separated by so-called *encapsulation boundaries*, which are in essence lines beginning with at least one dash followed by a non-whitespace character. A dash followed by a whitespace serves as a *byte-stuffing* character, a sort of escape for lines which begin with a dash themselves. Unfortunately, there are mail agents which do not follow byte-stuffing rules and pass lines beginning with dashes unmodified into resulting digests. To help handle such cases a global variable is provided which controls how much dashes should the line begin with for it to be recognized as an encapsulation boundary.

`number burst_eb_min_length` [Built-in variable]
Minimal number of consecutive dashes an encapsulation boundary must begin with.
The default is 2.

The following example shows a function which saves all parts of a digest message to separate disk files. The argument *orig* is a message descriptor. The resulting files are named by concatenating the string supplied by the *stem* argument and the ordinal number (1-based) of the message part.

```
func burst_digest(number orig, string stem)
do
  number msg message_burst(orig)
  number nparts message_count_parts(msg)

  loop for number i 1,
       while i <= nparts,
       set i i + 1
  do
    number part message_get_part(msg, i)
    number out open(sprintf('>%s%02d', stem, i))
    message_to_stream(out, part)
  done
  message_close(msg)
done
```

5.20 Quarantine Functions

`void quarantine (string text)` [Built-in Function]
Place the message to the quarantine queue, using *text* as explanatory reason.

5.21 SMTP Callout Functions

`number callout_open (string url)` [Library Function]
Opens connection to the callout server listening at *url*. Returns the descriptor of the connection.

`void callout_close (number fd)` [Library Function]
Closes the connection. *fd* is the file descriptor returned by the previous call to `callout_open`.

`number callout_do (number fd, string email [, string rest])` [Library Function]
Instructs the callout server identified by *fd* (a file descriptor returned by a previous call to `callout_open`) to verify the validity of the *email*. Optional *rest* argument supplies additional parameters for the server. It is formatted as a sequence of parameter settings delimited by whitespaces. Each setting is a parameter name and value separated by a '=' sign. See [callout parameters], page 254, for a discussion of available callout parameters.

Possible return values:

`0` Success. The *email* is found to be valid.

`e_not_found`
email does not exist.

`e_temp_failure`
The email validity cannot be determined right now, e.g. because remote SMTP server returned temporary failure. The caller should retry verification later.

`e_failure`
Some error occurred.

The function will throw the `e_callout_proto` exception if the remote host doesn't speak the correct callout protocol.

Upon return, `callout_do` modifies the following variables:

`last_poll_host`
Host name or IP address of the last polled SMTP server.

`last_poll_greeting`
Initial SMTP reply from the last polled host.

`last_poll_helo`
The reply to the `HELO` (`EHLO`) command, received from the last polled host.

`last_poll_sent`
Last SMTP command sent to the polled host. If nothing was sent, `last_poll_sent` contains the string '`nothing`'.

`last_poll_recv`
Last SMTP reply received from the remote host. In case of multi-line replies, only the first line is stored. If nothing was received the variable contains the string '`nothing`'.

The *default callout server* is defined by the `callout-url` statement in the configuration file, or by the `callout` statement in the `server milter` section (see [configuring default callout server], page 224. The following functions operate on that server.

`string default_callout_server_url ()` [Built-in Function]
Returns URL of the default callout server.

`number callout (string email)` [Library Function]
Verifies the validity of the *email* using the default callout server.

5.22 Compatibility Callout Functions

The following functions are wrappers over the callout functions described in the previous section. They are provided for backward compatibility.

These functions are defined in the module `poll.mfl`, which you must require prior to using any of them.

`boolean _pollhost (string ip, string email, string domain, string mailfrom)` [Library Function]

Poll SMTP host *ip* for email address *email*, using *domain* as `EHLO` domain and *mailfrom* as `MAIL FROM`. Returns 0 or 1 depending on the result of the test. In contrast to the `strictpoll` function, this function does not use cache database and does not fall back to polling MX servers if the main poll tempfails. The function can throw one of the following exceptions: `e_failure`, `e_temp_failure`.

`boolean _pollmx (string ip, string email, string domain, string mailfrom)` [Library Function]

Poll MXs of the *domain* for email address *email*, using *domain* as `EHLO` domain and *mailfrom* as `MAIL FROM` address. Returns 0 or 1 depending on the result of the test. In contrast to the `stdpoll` function, `_pollmx` does not use cache database and does not fall back to polling the *ip* if the poll fails. The function can throw one of the following exceptions: `e_failure`, `e_temp_failure`.

`boolean stdpoll (string email, string domain, string mailfrom)` [Library Function]

Performs standard poll for *email*, using *domain* as `EHLO` domain and *mailfrom* as `MAIL FROM` address. Returns 0 or 1 depending on the result of the test. Can raise one of the following exceptions: `e_failure`, `e_temp_failure`.

In `on` statement context, it is synonymous to `poll` without explicit *host*.

`boolean strictpoll (string host, string email, string domain, string mailfrom)` [Library Function]

Performs strict poll for *email* on host *host*. See the description of `stdpoll` for the detailed information.

In `on` context, it is synonymous to `poll host host`.

The *mailfrom* argument can be a comma-separated list of email addresses, which can be useful for servers that are unusually picky about sender addresses. It is advised, however, that this list always contain the '<>' address. For example:

```
_pollhost($client_addr, $f, "domain", "postmaster@my.net,<>")
```

See also Section 7.9 [conf-callout], page 227.

Before returning, all described functions set the following built-in variables:

Variable	Contains
last_poll_host	Host name or IP address of the last polled host.
last_poll_sent	Last SMTP command, sent to this host. If nothing was sent, it contains literal string '`nothing`'.
last_poll_recv	Last SMTP reply received from this host. In case of multi-line replies, only the first line is stored. If nothing was received the variable contains the string '`nothing`'.
cache_used	`1` if cached data were used instead of polling, `0` otherwise. This variable is set by `stdpoll` and `strictpoll`. If it equals `1`, none of the above variables are modified. See [cache_used example], page 67, for an example.

Table 5.1: Variables set by polling functions

5.23 Internet address manipulation functions

Following functions operate on IPv4 addresses in numeric form.

`number ntohl (number n)` [Built-in Function]
Converts the number *n*, from host to network byte order. The argument *n* is treated as an unsigned 32-bit number.

`number htonl (number n)` [Built-in Function]
Converts the number *n*, from network to host byte order. The argument *n* is treated as an unsigned 32-bit number.

`number ntohs (number n)` [Built-in Function]
The argument *n* is treated as an unsigned 16-bit number. The function converts this number from network to host order.

`number htons (number n)` [Built-in Function]
The argument *n* is treated as an unsigned 16-bit number. The function converts this number from host to network order.

`number inet_aton (string s)` [Built-in Function]
Converts the Internet host address *s* from the standard numbers-and-dots notation into the equivalent integer in host byte order.

```
inet_aton("127.0.0.1") ⇒ 2130706433
```

The numeric data type in MFL is signed, therefore on machines with 32 bit integers, this conversion can result in a negative number:

```
inet_aton("255.255.255.255") ⇒ -1
```

However, this does not affect arithmetical operations on IP addresses.

`string inet_ntoa (number n)` [Built-in Function]
Converts the Internet host address *n*, given in host byte order to string in standard numbers-and-dots notation:

```
inet_ntoa(2130706433) ⇒ "127.0.0.1"
```

`number len_to_netmask (number n)` [Built-in Function]
Convert number of masked bits *n* to IPv4 netmask:

```
inet_ntoa(len_to_netmask(24)) ⇒ 255.255.255.0
inet_ntoa(len_to_netmask(7)) ⇒ 254.0.0.0
```

If *n* is greater than 32 the function raises `e_range` exception.

`number netmask_to_len (number mask)` [Built-in Function]
Convert IPv4 netmask *mask* into netmask length (number of bits preserved by the mask):

```
netmask_to_len(inet_aton("255.255.255.0")) ⇒ 24
netmask_to_len(inet_aton("254.0.0.0")) ⇒ 7
```

The following functions operate on string representation of IPv4 and IPv6 addresses.

`string reverse_ipstr(string ip)` [Built-in Function]
Converts the IP address ip to reverse format suitable for use in DNS labels. That is, if *ip* is an IPv4 address, the return value is obtained by reversing the order of octets in the input:

```
reverse_ipstr("192.0.2.10") ⇒ "10.2.0.192"
```

If *IP* is an IPv6 address, the return string is formed as a sequence of nibbles separated by dots, encoded in reverse order, i.e. the low-order nibble is encoded first, followed by the next low-order nibble and so on. Each nibble is represented by a hexadecimal digit appending each byte from the IP represented in hex from the last to first, delimited by dots, e.g.:

```
reverse_ipstr("2001:db8:0:0:1::2")
⇒
"2.0.0.0.0.0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2"
```

`number is_ipstr(string s)` [Built-in Function]
Returns 1 if *s* is a string representation of an IP address (IPv4 or IPv6) and 0 otherwise.

`number is_ip4str(string s)` [Built-in Function]
Returns 1 if *s* is a string representation of an IPv4 address, and 0 otherwise.

`number is_ip6str(string s)` [Built-in Function]
Returns 1 if *s* is a string representation of an IPv6 address, and 0 otherwise.

The following functions operate on IP addresses or CIDRs represented as strings:

`boolean match_cidr (string ip, string cidr)` [Built-in Function]
It returns `true` if the IP address *ip* pertains to the IP range *cidr*. The first argument, *ip*, is a string representation of an IP address (IPv4 or IPv6). The second argument, *cidr*, is a string representation of a IP range in CIDR notation, i.e. '`addr/N`', where *addr* is IP address and *N* specifies address *prefix length* – the number of meaningful initial bits, counting from the left side of the address.

The following example will reject the mail if the IP address of the sending machine does not belong to the block `192.0.2.0/24`:

```
if not match_cidr(${client_addr}, "192.0.2.0/24")
  reject
fi
```

The following example does the same for a IPv6 CIDR `2001:DB8::/56`:

```
if not match_cidr(${client_addr}, "2001:DB8::/56")
  reject
fi
```

Notice, that in previous versions of `mailfromd` this function was implemented as MFL function and required the use of `match_cidr.mfl` module (see Section C.2 [8170-9000], page 287. This is no longer the case. The module itself is retained in the distribution for compatibility reasons. If your code uses it, you will see the following warning during compilation phase (split in two lines for typesetting reasons):

```
mailfromd: match_cidr.mfl:18.1-78: warning: This module is
deprecated. The match_cidr function is now built-in.
```

Just remove the `require 'match_cidr'` from your code to make this warning disappear.

5.24 DNS Functions

MFL offers two sets of functions for querying the Domain Name System. The `dns_query` function and associated `dns_reply_` functions provide a generalized DNS API.

Other functions provide a simplified API.

5.24.1 dns_query

`number dns_query (number` *`type`*`, string` *`domain`*`; number` *`sort`*`, number` *`resolve`*`)` [Built-in Function]

This function looks up the domain name *name*. The *type* argument specifies type of the query to perform. On success, the function returns *DNS reply descriptor*, a non-negative integer number identifying the reply. It can then be passed to any of the '`dns_reply_`' functions discussed below in order to retrieve the information from it.

If no matching records were found, the function returns '`-1`'.

On error, it throws a corresponding exception.

The *type* argument is one of the following constants (defined in the module '`dns`'):

`DNS_TYPE_A`
: Query the '`A`' record. The *domain* should be the hostname to look up.

`DNS_TYPE_NS`
: Query the '`NS`' records.

`DNS_TYPE_PTR`
: Query the '`PTR`' record. The *domain* address should be the IP address in dotted-quad form.

`DNS_TYPE_MX`
: Query the '`MX`' records.

`DNS_TYPE_TXT`
: Query the '`TXT`' records.

If the query returns multiple RR sets, the optional argument *sort* controls whether they should be returned in the same order as obtained from the DNS (0, the default), or should be sorted (1).

Optional argument *resolve* is consulted if *type* is `DNS_TYPE_MX` or `DNS_TYPE_NS`. By default, queries for these types return hostnames. The *resolve* argument controls whether to return IP addresses instead. Its possible values (defined in module `status.mfl` are:

`RESOLVE_NONE`
: Don't resolve hostnames to IP addresses. This is the default.

`RESOLVE_DFL`
: Resolve hostnames to IP addresses according to the address family of the SMTP session. That is, use '`A`' records if the client connected using the INET family (i.e. connected to the IPv4 address), and use '`AAAA`' records if the client connected to the IPv6 address.

`RESOLVE_IP4`
: Resolve hostnames to IPv4 addresses ('`A`' records).

`RESOLVE_IP6`
: Resolve hostnames to IPv6 addresses ('`AAAA`' records).

To extract actual data from the `dns_query` return value, use the functions `dns_reply_count` and `dns_reply_string`. The usual processing sequence is:

```
require dns

# Send the query and save the reply descriptor
set n dns_query(DNS_TYPE_NS, domain_name)

if n >= 0
  # If non-empty set is returned, iterate over each value in it:
  loop for set i 0,
       while i < dns_reply_count(n),
       set i i + 1
  do
    # Get the actual data:
    echo dns_reply_string(n, i)
  done
  # Release the memory associated with the reply.
  dns_reply_release(n)
fi
```

`void dns_reply_release (number` *`rd`*`)` [Built-in Function]

Release the memory associated with the reply *rd*. If *rd* is -1, the function does nothing.

`number dns_reply_count (number rd)` [Built-in Function]
Return the number of records in the reply *rd*. For convenience, if *rd* is -1, the function returns 0. If *rd* is negative (excepting -1), a '`e_failure`' exception is thrown.

`string dns_reply_string (number rd, number n)` [Built-in Function]
Returns *n*th record from the DNS reply *rd*.

`number dns_reply_ip (number rd, number n)` [Built-in Function]
Returns *n*th record from the DNS reply *rd*, if the reply contains IPv4 addresses.

5.24.2 Simplified DNS functions

These functions are implemented in two layers: *primitive* built-in functions which raise exceptions if the lookup fails, and library calls that are warranted to always return meaningful value without throwing exceptions.

The built-in layer is always available. The library calls become available after requesting the `dns` module (see Section 4.24 [Modules], page 108):

```
require dns
```

`string dns_getaddr (string domain)` [Library Function]
Returns a whitespace-separated list of IP addresses (`A` records) for *domain*.

`string dns_getname (string ipstr)` [Library Function]
Returns a whitespace-separated list of domain names (`PTR` records) for the IPv4 address *ipstr*.

`string getmx (string domain [, boolean ip])` [Library Function]
Returns a whitespace-separated list of '`MX`' names (if *ip* is not given or if it is `0`) or '`MX`' IP addresses (if `ip!=0`)) for *domain*. Within the returned string, items are sorted in order of increasing '`MX`' priority. If *domain* has no '`MX`' records, an empty string is returned. If the DNS query fails, `getmx` raises an appropriate exception.

Examples:

```
getmx("mafra.cz") ⇒ "smtp1.mafra.cz smtp2.mafra.cz relay.iol.cz"
getmx("idnes.cz") ⇒ "smtp1.mafra.cz smtp2.mafra.cz relay.iol.cz"
getmx("gnu.org")  ⇒ "mx10.gnu.org mx20.gnu.org"
getmx("org.pl") ⇒ ""
```

Notes:

1. Number of items returned by `getmx(domain)` can differ from that obtained from `getmx(domain, 1)`, e.g.:

```
getmx("aol.com")
  ⇒ mailin-01.mx.aol.com mailin-02.mx.aol.com
            mailin-03.mx.aol.com mailin-04.mx.aol.com
getmx("aol.com", 1)
  ⇒ 64.12.137.89 64.12.137.168 64.12.137.184
            64.12.137.249 64.12.138.57 64.12.138.88
            64.12.138.120 64.12.138.185 205.188.155.89
            205.188.156.185 205.188.156.249 205.188.157.25
            205.188.157.217 205.188.158.121 205.188.159.57
            205.188.159.217
```

2. This function is a wrapper over `dns_query`.

   If you intend to iterate over returned values, better use `dns_query` directly, e.g. instead of doing

   ```
   string_list_iterate(getmx(domain), ` ', MX, `do_something(MX)')
   ```

   use

   ```
   set n dns_query(DNS_TYPE_MX, domain)
   if n >= 0
     loop for set i 0,
          while i < dns_reply_count(n),
          set i i + 1
     do
       do_something(dns_reply_string(n, i))
     done
     dns_reply_release(n)
   fi
   ```

   See Section 5.24.1 [dns_query], page 166, for details about the `dns_query` function and associated `dns_reply_*` calls.

3. This interface is semi-deprecated.

   It will most probably be removed in future releases, when array data types are implemented.

`boolean primitive_hasmx (string` *`domain`*`)` [Built-in Function]

Returns `true` if the domain name given by its argument has any '`MX`' records.

If the DNS query fails, this function throws `failure` or `temp_failure`.

`boolean hasmx (string` *`domain`*`)` [Library Function]

Returns `true` if the domain name given by its argument has any '`MX`' records.

Otherwise, if *domain* has no '`MX`'s or if the DNS query fails, `hasmx` returns `false`.

`string primitive_hostname (string` *`ip`*`)` [Built-in Function]

The *ip* argument should be a string representing an IP address in *dotted-quad* notation. The function returns the canonical name of the host with this IP address obtained from DNS lookup. For example

```
primitive_hostname (${client_addr})
```

returns the fully qualified domain name of the host represented by Sendmail variable '`client_addr`'.

If there is no '`PTR`' record for *ip*, `primitive_hostname` raises the exception `e_not_found`.

If DNS query fails, the function raises `failure` or `temp_failure`, depending on the character of the failure.

`string hostname (string` *`ip`*`)` [Library Function]

The *ip* argument should be a string representing an IP address in *dotted-quad* notation. The function returns the canonical name of the host with this IP address obtained from DNS lookup.

If there is no '`PTR`' record for *ip*, or if the lookup fails, the function returns *ip* unchanged.

The previous `mailfromd` versions used the following paradigm to check if an IP address resolves:

```
if hostname(ip) != ip
  ...
```

`boolean primitive_ismx (string domain, string host)` [Built-in Function]

The *domain* argument is any valid domain name, the *host* is a host name or IP address.

The function returns `true` if *host* is one of the '`MX`' records for the *domain*.

If *domain* has no '`MX`' records, `primitive_ismx` raises exception `e_not_found`.

If DNS query fails, the function raises `failure` or `temp_failure`, depending on the character of the failure.

`boolean ismx (string domain, string host)` [Library Function]

The *domain* argument is any valid domain name, the *host* is a host name or IP address.

The function returns `true` if *host* is one of the '`MX`' records for the *domain*. Otherwise it returns `false`.

If *domain* has no '`MX`' records, or if the DNS query fails, the function returns `false`.

`string primitive_resolve (string host, [string domain, number family])` [Built-in Function]

Reverse of `primitive_hostname`. The `primitive_resolve` function returns the IP address for the host name specified by its *host* argument. If the SMTP session uses IPv4 protocol, '`A`' record is queried. It it uses IPv6, '`AAAA`' record is queried. A particular record type can be requested via optional *family*, which can have one of the following values (defined in `status.mfl`):

`RESOLVE_DFL`
: Look for '`A`' or '`AAAA`', depending on the connection type. This is the default.

`RESOLVE_IP4`
: Resolve to IPv4 addresses ('`A`' records).

`RESOLVE_IP6`
: Resolve to IPv6 addresses ('`AAAA`' records).

If *host* has no records of the requested type, the function raises the exception `e_not_found`.

If DNS lookup fails, the function raises `failure` or `temp_failure`, depending on the character of the failure.

Optional *domain* argument is deprecated. If a non-empty string is given as *domain*, the function works as follows:

1. If *domain* is '`in-addr.arpa`' (case-insensitive)

   The *host* must be a string representation of an IPv4 address in dotted-quad form. If it is not, a `e_inval` exception is thrown. The octets in the IPv4 are reversed,

a dot and '`domain`' are appended to it, and a '`PTR`' record is queried for the resulting name.

Thus, the call

```
primitive_resolve("192.0.2.1", "in-addr.arpa")
```

is equivalent to

```
primitive_hostname("192.0.2.1")
```

2. *domain* is '`ip6.arpa`' (case-insensitive)

   The *host* must be a string representation of an IPv6. If it is not, a `e_inval` exception is thrown. The octets of this IPv6 address are reversed, a dot and '`domain`' are appended to it, and a '`PTR`' record is queried for the resulting name.

   Thus, the call

   ```
   primitive_resolve("2001:DB8::1", "ip6.arpa")
   ```

   is equivalent to

   ```
   primitive_hostname("2001:DB8::1")
   ```

3. *host* is a string representation of an IPv4 address

   The address is reversed as in (1), then a dot and '`domain`' are appended to it. Finally, the DNS is queried for an '`A`' record of the resulting name.

   Thus,

   ```
   primitive_resolve("192.0.2.1", "rev.example.com")
   ```

   is equivalent to

   ```
   primitive_resolve("1.2.0.192.rev.example.com")
   ```

4. *host* is a string representation of an IPv6 address

   The address is reversed as in (2), then a dot and '`domain`' are appended to it. Finally, the DNS is queried for an '`AAAA`' record of the resulting name.

5. None of the above. Same as `primitive_hostname('host.domain','',resolve)`.

`string resolve (string host, [string domain, number family])` [Library Function]

Reverse of `hostname`. The `resolve` function returns IP address for the host name specified by *host* argument. If the host name cannot be resolved, or a DNS failure occurs, the function returns '`"0"`'.

This function is entirely equivalent to `primitive_resolve` (see above), except that it never raises exceptions.

`string ptr_validate (string ip)` [Built-in Function]

Tests whether the DNS reverse-mapping for *ip* exists and correctly points to a domain name within a particular domain.

First, it obtains all '`PTR`' records for *ip*. Then, for each record returned, a look up for '`A`' (or '`AAAA`', if *ip* is an IPv6 address) records is performed and IP addresses of each record are compared against *ip*. The function returns true if a matching '`A`' (or '`AAAA`') record is found.

This function can raise the following exceptions:

`e_not_found`
: Unable to resolve IP address or hostname.

`e_failure`
: Fatal error while resolving.

`e_temp_failure`
: Temporary error in DNS resolution.

`e_too_many`
: Too many CNAME records (see [CNAME chains], page 223).

`boolean primitive_hasns (string` *`domain`*`)` [Built-in Function]
Returns '`True`' if the domain *domain* has at least one '`NS`' record. Throws exception if DNS lookup fails.

`boolean hasns (string` *`domain`*`)` [Library Function]
Returns '`True`' if the domain *domain* has at least one '`NS`' record. Returns '`False`' if there are no '`NS`' records or if the DNS lookup fails.

`string getns (string` *`domain`* `; boolean` *`resolve`*`, boolean` *`sort`*`)` [Library Function]
Returns a whitespace-separated list of all the '`NS`' records for the domain *domain*. Optional parameters *resolve* and *sort* control the formatting. If *resolve* is 0 (the default), the resulting string will contain IP addresses of the NS servers. If *resolve* is not 0, hostnames will be returned instead. If *sort* is 1, the returned items will be sorted.

If the DNS query fails, `getns` raises an appropriate exception.

Notes:

1. This function is a wrapper over `dns_query`.

   If you intend to iterate over returned values, better use `dns_query` directly, e.g. instead of doing

   ```
   string_list_iterate(getns(domain), ` ', NS, `do_something(NS)')
   ```

   use

   ```
   set n dns_query(DNS_TYPE_NS, domain)
   if n >= 0
     loop for set i 0,
          while i < dns_reply_count(n),
          set i i + 1
     do
       do_something(dns_reply_string(n, i))
     done
     dns_reply_release(n)
   fi
   ```

   See Section 5.24.1 [dns_query], page 166, for details about the `dns_query` function and associated `dns_reply_*` calls.

2. This interface is semi-deprecated.

   It will most probably be removed in future releases, when array data types are implemented.

5.25 Geolocation functions

The *geolocation functions* allow you to identify the country where the given IP address or host name is located. These functions are available only if the `libmaxminddb` library is installed and `mailfromd` is compiled with the '`GeoIP2`' support.

The `libmaxminddb` library is distributed by '`MaxMind`' under the terms of the *Apache License* Version 2.0. It is available from `https://dev.maxmind.com/geoip/geoip2/downloadable/#MaxMind_APIs`.

`void geoip2_open (string filename)` [Built-in Function]

Opens the geolocation database file *filename*. The database must be in GeoIP2 format.

If the database cannot be opened, `geoip2_open` throws the `e_failure` exception.

If this function is not called, geolocation functions described below will try to open the database file '`/usr/share/GeoIP/GeoLite2-City.mmdb`'.

`string geoip2_dbname (void)` [Built-in Function]

Returns the name of the geolocation database currently in use.

The geolocation database for each IP address, which serves as a look up key, stores a set of items describing this IP. This set is organized as a map of key-value pairs. Each key is a string value. A value can be a scalar, another map or array of values. Using JSON notation, the result of a look up in the database might look as:

```
{
    "country":{
        "geoname_id":2921044,
        "iso_code":"DE",
        "names":{
            "en": "Germany",
            "de": "Deutschland",
            "fr":"Allemagne"
        },
    },
    "continent":{
        "code":"EU",
        "geoname_id":6255148,
        "names":{
            "en":"Europe",
            "de":"Europa",
            "fr":"Europe"
        }
    },
    "location":{
        "accuracy_radius":200,
        "latitude":49.4478,
        "longitude":11.0683,
        "time_zone":"Europe/Berlin"
    },
    "city":{
        "geoname_id":2861650,
        "names":{
            "en":"Nuremberg",
            "de":"Nürnberg",
            "fr":"Nuremberg"
        }
    },
    "subdivisions":[{
        "geoname_id":2951839,
        "iso_code":"BY",
        "names":{
            "en":"Bavaria",
            "de":"Bayern",
            "fr":"Bavière"
        }
    }
}
```

Each particular data item in such structure is identified by its *search path*, which is a dot-delimited list of key names leading to that value. For example, using the above map, the name of the city in English can be retrieved using the key `city.names.en`.

`string geoip2_get (string ip, string path)` [Built-in Function]
Looks up the IP address *ip* in the geolocation database. If found, returns data item identified by the search path *path*.

The function can throw the following exceptions:

e_not_found
: The *ip* was not found in the database.

e_range
: The *path* does not exist the returned map.

e_failure
: General error occurred. E.g. the database cannot be opened, *ip* is not a valid IP address, etc.

`string geoip2_get_json (string ip [; number indent)` [Built-in Function]
Looks up the *ip* in the database and returns entire data set associated with it, formatted as a JSON object. If the optional parameter *indent* is supplied and is greater than zero, it gives the indentation for each nesting level in the JSON object.

Applications may test whether the GeoIP2 support is present and enable the corresponding code blocks conditionally, by testing if the '`WITH_GEOIP2`' m4 macro is defined. For example, the following code adds to the message the '`X-Originator-Country`' header, containing the 2 letter code of the country where the client machine is located. If `mailfromd` is compiled without the '`GeoIP2`' support, it does nothing:

```
m4_ifdef(`WITH_GEOIP2',`
  try
  do
    header_add("X-Originator-Country", geoip2_get($client_addr,
                                                  'country.iso_code'))
  done
  catch e_not_found or e_range
  do
    pass
  done
')
```

5.26 Database Functions

The functions described below provide a user interface to DBM databases.

Each DBM database is a separate disk file that keeps *key/value pairs*. The interface allows to retrieve the value corresponding to a given key. Both '`key`' and '`value`' are null-terminated character strings. To lookup a key, it is important to know whether its length includes the terminating null byte. By default, it is assumed that it does not.

Another important database property is the *file mode* of the database file. The default file mode is '`640`' (i.e. '`rw-r----`', in symbolic notation).

These and other properties can be configured using the `dbprop` pragma:

```
#pragma dbprop pattern prop [prop]
```

The *pattern* is the database name or shell-style globbing pattern. Properties defined by that pragma apply to each database whose name matches this pattern. If several `dbprop` pragmas match the database name, the one that matches exactly is preferred.

The rest of arguments define properties for that database. The valid values for *prop* are:

1. The word '`null`', meaning that the terminating null byte is included in the key length.

   Setting '`null`' property is necessary, for databases created with `makemap -N hash` command.

2. File mode for the disk file. It can be either an octal number, or a symbolic mode specification in ls-like format. E.g., the following two formats are equivalent:

   ```
   640
   rw-r----
   ```

3. *DBM scheme*, which specifies the type of the DBM (see Section 3.15.1 [Database Formats], page 31) to use for this database. The scheme consists of DBM type name followed by a colon and two slashes. E.g. '`gdbm://`' means a GDBM database. See [DBM scheme], page 229, for a detailed discussion.

For example, consider the following pragmas:

```
#pragma dbprop /etc/mail/whitelist.db 640
```

It tells that the database file `whitelist.db` has privileges '`640`' and do not include null in the key length.

Similarly, the following pragma:

```
#pragma dbprop `/etc/mail/*.db' null 600 bdb://
```

declares that all database files in directory `/etc/mail` are Berkeley DB files, have privileges '`640`', and include null terminator in the key length. *Notice*, the use of `m4` quoting characters in the example below. Without them, the sequence '`/*`' would have been taken as the beginning of a comment.

Additionally, for compatibility with previous versions (up to 5.0), the terminating null property can be requested via an optional argument to the database functions (in description below, marked as *null*).

`boolean dbmap (string` *`db`*`, string` *`key`*`, [boolean` *`null`*`])` [Built-in Function]

Looks up *key* in the DBM file *db* and returns `true` if it is found.

See above for the meaning of *null*.

See [whitelisting], page 28, for an example of using this function.

`string dbget (string` *`db`*`, string` *`key`* `[, string` *`default`*`, boolean` *`null`*`])` [Built-in Function]

Looks up *key* in the database *db* and returns the value associated with it. If the key is not found returns *default*, if specified, or empty string otherwise.

See above for the meaning of *null*.

`void dbput (string` *`db`*`, string` *`key`*`, string` *`value`* `[, boolean` *`null`*`, number` *`mode`* `])` [Built-in Function]

Inserts in the database a record with the given *key* and *value*. If a record with the given *key* already exists, its value is replaced with the supplied one.

See above for the meaning of *null*. Optional *mode* allows to explicitly specify the file mode for this database. See also `#pragma dbprop`, described above.

`void dbinsert (string db, string key, string value [, boolean replace, boolean null, number mode ])` [Built-in Function]

This is an improved variant of `dbput`, which provides a better control on the actions to take if the *key* already exists in the database. Namely, if *replace* is '`True`', the old value is replaced with the new one. Otherwise, the '`e_exists`' exception is thrown.

`void dbdel (string db, string key [, boolean null, number mode])` [Built-in Function]

Delete from the database the record with the given *key*. If there are no such record, return without signalling error.

If the optional *null* argument is given and is not zero, the terminating null character will be included in *key* length.

Optional *mode* allows to explicitly specify the file mode for this database. See also `#pragma dbprop`, described above.

The functions above have also the corresponding exception-safe interfaces, which return cleanly if the '`e_dbfailure`' exception occurs. To use these interfaces, request the `safedb` module:

```
require safedb
```

The exception-safe interfaces are:

`number safedbmap (string db, string key [, number default, boolean null])` [Library Function]

This is an exception-safe interface to `dbmap`. If a database error occurs while attempting to retrieve the record, `safedbmap` returns *default* or '`0`', if it is not defined.

`string safedbget (string db, string key [, string default, boolean null])` [Library Function]

This is an exception-safe interface to `dbget`. If a database error occurs while attempting to retrieve the record, `safedbget` returns *default* or empty string, if it is not defined.

`void safedbput (string db, string key, string value [, boolean null])` [Library Function]

This is an exception-safe interface to `dbput`. If a database error occurs while attempting to retrieve the record, the function returns without raising exception.

`void safedbdel (string db, string key [, boolean null])` [Library Function]

This is an exception-safe interface to `dbdel`. If a database error occurs while attempting to delete the record, the function returns without raising exception.

The verbosity of '`safedb`' interfaces in case of database error is controlled by the value of `safedb_verbose` variable. If it is '`0`', these functions return silently. This is the default behavior. Otherwise, if `safedb_verbose` is not '`0`', these functions log the detailed diagnostics about the database error and return.

The following functions provide a sequential access to the contents of a DBM database:

`number dbfirst (string name)` [Built-in Function]
Start sequential access to the database *name*. The return value is an opaque identifier, which is used by the remaining sequential access functions. This number is '`0`' if the database is empty.

`number dbnext (number dn)` [Built-in Function]
Select next record form the database. The argument *dn* is the access identifier, returned by a previous call to `dbfirst` or `dbnext`.

Returns new access identifier. This number is '`0`' if all records in the database have been visited.

The usual approach for iterating over all records in a database *dbname* is:

```
loop for number dbn dbfirst(dbname)
do
  ...
done while dbnext(dbn)
```

The following two functions can be used to access values of the currently selected database record. Their argument, *dn*, is the access identifier, returned by a previous call to `dbfirst` or `dbnext`.

`string dbkey (number dn)` [Built-in Function]
Return the key from the selected database record.

`string dbvalue (number dn)` [Built-in Function]
Return the value from the selected database record.

`void dbbreak (number dn)` [Built-in Function]
Stop sequential access to the database and deallocate all associated resources. Use this function if you need to break from the sequential access loop, as in the example below:

```
loop for number dbn dbfirst(dbname)
do
  if some_condition
    dbbreak(dbn)
    break
  fi
done while dbnext(dbn)
```

`number db_expire_interval (string fmt)` [Built-in Function]
The *fmt* argument is a database format identifier (see Section 3.15.1 [Database Formats], page 31). If it is valid, the function returns the expiration interval for that format. Otherwise, `db_expire_interval` raises the `e_not_found` exception.

`string db_name (string fmtid)` [Built-in Function]
The *fmtid* argument is a database format identifier (see Section 3.15.1 [Database Formats], page 31). The function returns the file name for that format. If *fmtid* does not match any known format, `db_name` raises the `e_not_found` exception.

`number db_get_active (string fmtid)` [Built-in Function]
Returns the flag indicating whether the cache database *fmtid* is currently enabled. If *fmtid* does not match any known format, `db_name` raises the `e_not_found` exception.

`void db_set_active (string fmtid, boolean enable)` [Built-in Function]
Enables the cache database *fmtid* if *enable* is '`True`', or disables it otherwise. For example, to disable DNS caching, do:

```
db_set_active("dns", 0)
```

`boolean relayed (string domain)` [Built-in Function]
Returns `true` if the string *domain* is found in one of relayed domain files (see Section 7.2 [conf-base], page 220). The usual construct is:

```
if relayed(hostname(${client_addr}))
  ...
```

which yields `true` if the IP address from `Sendmail` variable '`client_addr`' is relayed by the local machine.

5.27 Control Database

Control database is a DBM file whose records define actions to be taken for mails coming from particular IP or email addresses. Functions and variables for using the control database are defined in module `cdb.mfl`.

`string cdb_name` [cdb variable]
Name of the database file to use as control database. By default it is `/etc/mail/mfctl.db`.

`number cdb_greylist_interval` [cdb variable]
Greylisting interval, for database records that prescribe greylisting (see below). Defaults to 900 seconds.

`void cdb_check (string prefix, string addr)` [Library Function]
Perform a look up in the database. If the value is found, take the action it indicates.

The key to use for the look up depends on the value of *prefix*:

ip
: The *addr* argument must be an IP address. The look up key is '`ip:addr`'.

email
: The *addr* argument is an email address. The key is '`email:cemail`', where *cemail* is *addr* in *canonical form*, obtained by converting its domain part to lower case.

domain
: The *addr* argument is an email address. The key is formed as '`domain:dom`', where *dom* is domain part of *addr* converted to lower case.

subdomain
: Initial key value is obtained as for '`domain`'. If the key is found, the requested action is performed. Otherwise, the shortest hostname prefix (sequence of characters up to and including the first dot) is stripped off the domain and the process is retried.

Thus, the action is determined by the longest subdomain of *addr*, *x*, for which the key '`domain:x`' exists in the database.

mx If *addr* is an email address, its domain part is selected, otherwise it is used as is. The list of MX servers for this domain is obtained. For each *host* from that list the key '`mx:host`' is looked up. First entry found is used.

The function `cdb_check` returns if the key was not found in the database or if the value found was '`OK`' (case-insensitive) or empty. Otherwise, the looked up value determines the action (see [reply actions], page 92), as described in the following table. The action is returned and execution of the filter program stops.

CONTINUE
: Continue to the next milter state. See [continue], page 13.

ACCEPT Accept the mail. See [accept], page 13.

REJECT Reject the mail. See [reject], page 13.

TEMPFAIL
: Return a temporary failure. See [tempfail], page 13.

GREYLIST
: Greylist the mail using the interval defined by the `cdb_greylist_interval` variable. If called in `envrcpt` handler, the action is taken immediately. Otherwise, if called in `connect`, `helo` or `envfrom`, the action is delayed until `envrcpt` is invoked. Otherwise, if called in any other handler an error is reported.

code xcode text
code text
text Here, *code* is SMTP response code, *xcode* is extended STMP response code, and *text* is an arbitrary text. If *code* is given, it must begin with '`4`' or '`5`'. Its first digit defines the action to be taken: temporary failure (if '`4`') or reject (if '`5`'). If *xcode* is given, its first digit must match that of *code*.

If only *text* is supplied, it is equivalent to

```
reject(550, 5.1.0, text)
```

5.28 System functions

`boolean access (string pathname, number mode)` [Built-in Function]

Checks whether the calling process can access the file *pathname*. If *pathname* is a symbolic link, it is dereferenced. The function returns '`True`' if the file can be accessed and '`False`' otherwise[4].

Symbolic values for *mode* are provided in module `status`:

F_OK Tests for the existence of the file.

[4] *Note*, that the return code is inverted in respect to the system function '`access(2)`'.

R_OK Tests whether the file exists and grants read permission.

W_OK Tests whether the file exists and grants write permission.

X_OK Tests whether the file exists and grants execute permission.

`string getenv (string name)` [Built-in Function]
Searches the environment list for the variable *name* and returns its value. If the variable is not defined, the function raises the exception '`e_not_found`'.

`string gethostname ([bool fqn])` [Built-in Function]
Return the host name of this machine.

If the optional *fqn* is given and is '`true`', the function will attempt to return fully-qualified host name, by attempting to resolve it using DNS.

`string getdomainname ()` [Built-in Function]
Return the domain name of this machine. Note, that it does not necessarily coincide with the actual machine name in DNS.

Depending on the underlying '`libc`' implementation, this call may return empty string or the string '`(none)`'. Do not rely on it to get the real domain name of the box `mailfromd` runs on, use `localdomain` (see below) instead.

`string localdomain ()` [Library Function]
Return the local domain name of this machine.

This function first uses `getdomainname` to make a first guess. If it does not return a meaningful value, `localdomain` calls `gethostname(1)` to determine the fully qualified host name of the machine, and returns its domain part.

To use this function, require the `localdomain` module (see Section 4.24 [Modules], page 108), e.g.: `require localdomain`.

`number time ()` [Built-in Function]
Return the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

`string strftime (string fmt, number timestamp)` [Built-in Function]
`string strftime (string fmt, number timestamp, boolean gmt)` [Built-in Function]
Formats the time *timestamp* (seconds since the Epoch) according to the format specification *format*. Ordinary characters placed in the format string are copied to the output without conversion. Conversion specifiers are introduced by a '`%`' character. See Appendix B [Time and Date Formats], page 283, for a detailed description of the conversion specifiers. We recommend using single quotes around *fmt* to prevent '`%`' specifiers from being interpreted as `Mailfromd` variables (See Section 4.8 [Literals], page 59, for a discussion of quoted literals and variable interpretation within them).

The *timestamp* argument can be a return value of `time` function (see above).

For example:

```
strftime('%Y-%m-%d %H:%M:%S %Z', 1164477564)
 ⇒ 2006-11-25 19:59:24 EET
strftime('%Y-%m-%d %H:%M:%S %Z', 1164477564, 1)
 ⇒ 2006-11-25 17:59:24 GMT
```

`string uname (string format)` [Built-in Function]

This function returns system information formatted according to the format specification *format*. Ordinary characters placed in the format string are copied to the output without conversion. Conversion specifiers are introduced by a '`%`' character. The following conversions are defined:

%s Name of this system.

%n Name of this node within the communications network to which this node is attached. Note, that it does not necessarily coincide with the actual machine name in DNS.

%r Kernel release.

%v Kernel version.

%m Name of the hardware type on which the system is running.

For example:

```
uname('%n runs %s, release %r on %m')
  ⇒ "Trurl runs Linux, release 2.6.26 on i686"
```

Notice the use of single quotes.

`void unlink (string name)` [Built-in Function]

Unlinks (deletes) the file *name*. On error, throws the `e_failure` exception.

`number system (string str)` [Built-in Function]

The function `system` executes a command specified in *str* by calling `/bin/sh -c string`, and returns -1 on error or the return status of the command otherwise.

`void sleep (number secs[, usec])` [Built-in Function]

Sleep for *secs* seconds. If optional *usec* argument is given, it specifies additional number of microseconds to wait for. For example, to suspend execution of the filter for 1.5 seconds:

```
sleep(1,500000)
```

This function is intended mostly for debugging and experimental purposes.

`number umask (number mask)` [Built-in Function]

Set the umask to *mask* & 0777. Return the previous value of the mask.

5.29 System User Database

`string getpwnam (string name)` [Built-in Function]
`string getpwuid (number uid)` [Built-in Function]

Look for the user *name* (`getpwnam`) or user ID *uid* (`getpwuid`) in the system password database and return the corresponding record, if found. If not found, raise the '`e_not_found`' exception.

The returned record consists of six fields, separated by colon sign:

```
uname:passwd:uid:gid:gecos:dir:shell
```

Field	Meaning
uname	user name
passwd	user password
uid	user ID
gid	group ID
gecos	real name
dir	home directory
shell	shell program

For example:

```
getpwnam("gray")
⇒ "gray:x:1000:1000:Sergey Poznyakoff:/home/gray:/bin/bash"
```

Following two functions can be used to test for existence of a key in the user database:

`boolean mappwnam (string` *`name`*`)` [Built-in Function]
`boolean mappwuid (number` *`uid`*`)` [Built-in Function]
Return '`true`' if *name* (or *uid*) is found in the system user database.

5.30 Sieve Interface

'`Sieve`' is a powerful mail filtering language, defined in RFC 3028. `Mailfromd` supports an extended form of this language. For a description of the language and available extensions, see Section "Sieve Language" in *GNU Mailutils Manual*.

`boolean sieve (number` *`msg`*`, string` *`script`* `[, number` *`flags`*`, string` *`file`*`, number` *`line`*`])` [Built-in Function]
Compile the Sieve program *script* and execute it over the message identified by the descriptor *nmsg*.

Optional *flags* modify the behavior of the function. It is a bit-mask field, consisting of a bitwise `or` of one or more of the following flags, defined in `sieve.mfl`:

`MF_SIEVE_FILE`
The *script* argument specifies the name of a Sieve program file. This is the default.

`MF_SIEVE_TEXT`
The *script* argument is a string containing entire Sieve program. Optional arguments *file* and *line* can be used to fix source locations in Sieve diagnostic messages (see below).

`MF_SIEVE_LOG`
Log every executed '`Sieve`' action.

`MF_SIEVE_DEBUG_TRACE`
Trace execution of '`Sieve`' tests.

`MF_SIEVE_DEBUG_INSTR`
Log every instruction, executed in the compiled '`Sieve`' code. This produces huge amounts of output and is rarely useful, unless you suspect some bug in '`Sieve`' implementation and wish to trace it.

For example, `MF_SIEVE_LOG|MF_SIEVE_DEBUG_TRACE` enables logging '`Sieve`' actions and tests.

The `sieve` function returns `true` if the message was accepted by the *script* program, and `false` otherwise. Here, the word *accepted* means that some form of '`KEEP`' action (see Section "Actions" in *GNU Mailutils Manual*) was executed over the message.

While executing the Sieve script, Sieve environment (*RFC 5183*) is initialized as follows:

domain The domain name of the server Sieve is running on.

host Host name of the server Sieve is running on.

location The string '`MTA`'.

name The string '`GNU Mailutils`'.

phase The string '`pre`'.

remote-host
Defined to the value of '`client_ptr`' macro, if it was required.

remote-ip Defined to the value of '`client_addr`' macro, if it was required.

version The version of GNU Mailutils.

The following example discards each message not accepted by the '`Sieve`' program `/etc/mail/filter.siv`:

```
require 'sieve'

group eom
do
  if not sieve(current_message(), "/etc/mail/filter.siv", MF_SIEVE_LOG)
     discard
  fi
done
```

The Sieve program can be embedded in the MFL filter, as shown in the example below:

```
require 'sieve'

prog eom
do
  if not sieve(current_message(),
               "require \"fileinto\";\n"
               "fileinto \"/tmp/sieved.mbox\";",
               MF_SIEVE_TEXT | MF_SIEVE_LOG)
     discard
  fi
done
```

In such cases, any Sieve diagnostics (error messages, traces, etc.) will be marked with the locations relative to the line where the call to `sieve` appears. For example, the above program produces the following in the log:

```
prog.mfl:7: FILEINTO; delivering into /tmp/sieved.mbox
```

Notice, that the line number correctly refers to the line where the `fileinto` action appears in the source. However, there are cases where the reported line number is incorrect. This happens, for instance, if *script* is a string variable defined elsewhere. To handle such cases, `sieve` accepts two optional parameters which are used to compute the location in the Sieve program. The *file* parameter specifies the file name where the definition of the program appears, and the *line* parameter gives the number of line in that file where the program begins. For example:

```
require 'sieve'

const sieve_prog_line __line__ + 2
string sieve_prog <<EOT
require "fileinto";
fileinto "/tmp/sieved.mbox";
EOT

prog eom
do
  if not sieve(current_message(),
               sieve_prog, MF_SIEVE_TEXT | MF_SIEVE_LOG,
               __file__, sieve_prog_line)
     discard
  fi
done
```

The actual Sieve program begins two lines below the `sieve_prog_line` constant definition, which is reflected in its initialization.

5.31 Interfaces to Third-Party Programs

A set of functions is defined for interfacing with other filters via TCP. Currently implemented are interfaces with `SpamAssassin spamd` daemon and with `ClamAV` anti-virus.

Both interfaces work much the same way: the remote filter is connected and the message is passed to it. If the remote filter confirms that the message matches its requirements, the function returns `true`. Notice that in practice that means that such a message *should be rejected or deferred*.

The address of the remote filter is supplied as the second argument in the form of a standard URL:

proto`://`*path*`[:`*port*`]`

The *proto* part specifies the *connection protocol*. It should be '`tcp`' for the TCP connection and '`file`' or '`socket`' for the connection via UNIX socket. In the latter case the *proto* part can be omitted. When using TCP connection, the *path* part gives the remote host name or IP address and the optional *port* specifies the port number or service name to use. For example:

```
# connect to 'remote.filter.net' on port 3314:
tcp://remote.filter.net:3314

# the same, using symbolic service name (must be defined in
```

```
# /etc/services):
tcp://remote.filter.net:spamd

# Connect via a local UNIX socket (equivalent forms):
/var/run/filter.sock
file:///var/run/filter.sock
socket:///var/run/filter.sock
```

The description of the interface functions follows.

5.31.1 SpamAssassin

`boolean spamc (number` *`msg`*`, string` *`url`*`, number` *`prec`*`,` [Built-in Function]
`number` *`command`*`)`

Send the message *msg*t to the SpamAssassin daemon (`spamd`) listening on the given *url*. The *command* argument identifies what kind of processing is needed for the message. Allowed values are:

SA_SYMBOLS
: Process the message and return 1 or 0 depending on whether it is diagnosed as spam or not. Store SpamAssassin keywords in the global variable `sa_keywords` (see below).

SA_REPORT
: Process the message and return 1 or 0 depending on whether it is diagnosed as spam or not. Store entire SpamAssassin report in the global variable `sa_keywords`.

SA_LEARN_SPAM
: Learn the supplied message as spam.

SA_LEARN_HAM
: Learn the supplied message as ham.

SA_FORGET
: Forget any prior classification of the message.

The second argument, *prec*, gives the precision, in decimal digits, to be used when converting SpamAssassin diagnostic data and storing them into `mailfromd` variables.

The floating point SpamAssassin data are converted to the integer `mailfromd` variables using the following relation:

```
var = int(sa-var * 10**prec)
```

where *sa-var* stands for the SpamAssassin value and *var* stands for the corresponding `mailfromd` one. `int()` means taking the integer part and '`**`' denotes the exponentiation operator.

The function returns additional information via the following variables:

`sa_score`
: The spam score, converted to integer as described above. To convert it to a floating-point representation, use `sa_format_score` function (see Section 5.4 [String manipulation], page 130). See also the example below.

`sa_threshold`
: The threshold, converted to integer form.

`sa_keywords`
: If *command* is '`SA_SYMBOLS`', this variable contains a string of comma-separated SpamAssassin keywords identifying this message, e.g.:

```
ADVANCE_FEE_1,AWL,BAYES_99
```

If *command* is '`SA_REPORT`', the value of this variable is a *spam report* message. It is a multi-line textual message, containing detailed description of spam scores in a tabular form. It consists of the following parts:

1. A preamble.
2. Content preview.

   The words '`Content preview`', followed by a colon and an excerpt of the message body.
3. Content analysis details.

   It has the following form:

```
Content analysis details: (score points, max required)
```

   where *score* and *max* are spam score and threshold in floating point.
4. Score table.

   The score table is formatted in three columns:

   pts
   : The score, as a floating point number with one decimal digit.

   rule name
   : SpamAssassin rule name that contributed this score.

   description
   : Textual description of the rule

   The score table can be extracted from `sa_keywords` using `sa_format_report_header` function (see Section 5.4 [String manipulation], page 130), as illustrated in the example below.

The value of this variable is undefined if *command* is '`SA_LEARN_SPAM`', '`SA_LEARN_HAM`' or '`SA_FORGET`'.

The `spamc` function can signal the following exceptions: `e_failure` if the connection fails, `e_url` if the supplied URL is invalid and `e_range` if the supplied port number is out of the range 1–65535.

An example of using this function:

```
prog eom
do
  if spamc(current_message(), "tcp://192.168.10.1:3333", 3,
           SA_SYMBOLS)
    reject 550 5.7.0
         "Spam detected, score %sa_score with threshold %sa_threshold"
  fi
done
```

Here is a more advanced example:

```
prog eom
do
  set prec 3
  if spamc(current_message(),
           "tcp://192.168.10.1:3333", prec, SA_REPORT)
    add "X-Spamd-Status" "SPAM"
  else
    add "X-Spamd-Status" "OK"
  fi
  add "X-Spamd-Score" sa_format_score(sa_score, prec)
  add "X-Spamd-Threshold" sa_format_score(sa_threshold, prec)
  add "X-Spamd-Keywords" sa_format_report_header(sa_keywords)
done
```

`boolean sa (string url, number prec; number command)` [Library Function]

Additional interface to the `spamc` function, provided for backward compatibility. It is equivalent to

```
spamc(current_message(), url, prec, command)
```

If *command* is not supplied, '`SA_SYMBOLS`' is used.

5.31.2 ClamAV

`boolean clamav (number msg, string url)` [Built-in Function]

Pass the message *msg* to the ClamAV daemon at *url*. Return `true` if it detects a virus in it. Return virus name in `clamav_virus_name` global variable.

The message is sent to `clamd` in chunks. The size of each chunk is defined by the variable `clamav_stream_max_length`. Its value should not be greater than that of the `StreamMaxLength` parameter in `clamd.conf`.

To stay on the safe side, `clamav_stream_max_length` defaults to 4096 bytes. To improve performance, adjust it to the actual value of `StreamMaxLength`.

The `clamav` function can signal the following exceptions:

`e_failure`
: Unable to connect to the server, or failed to send the message to it. In particular, this exception will be raised if the value of `clamav_stream_max_length` is greater than the `StreamMaxLength clamd` configuration parameter. In this case, the `clamd` server will report:

  ```
  WARNING: INSTREAM: Size limit reached, (requested: n, max: m)
  ```

  where *n* and *m* are decimal numbers. Adjust your `clamav_stream_max_length` value so that

  ```
  clamav_stream_max_length < m
  ```

`e_url`
: The supplied URL is invalid.

An example usage:

```
prog eom
do
  if clamav(current_message(), "tcp://192.168.10.1:6300")
    reject 550 5.7.0 "Infected with %clamav_virus_name"
  fi
done
```

5.32 Rate limiting functions

`number rate (string` *key*`, number` *sample-interval*`,` [Built-in Function]
`[number` *min-samples*`, number` *threshold*`])`

Returns the mail sending rate for *key* per *sample-interval*. Optional *min-samples*, if supplied, specifies the minimal number of mails needed to obtain the statistics. The default is 2. Optional *threshold* controls rate database updates. If the observed rate (per *sample-interval* seconds) is higher than the *threshold*, the hit counters for that key are not incremented and the database is not updated. Although the *threshold* argument is optional[5], its use is strongly encouraged. Normally, the value of *threshold* equals the value compared with the return from *rate*, as in:

```
if rate("$f-$client_addr", rate_interval, 4, maxrate) > maxrate
  tempfail 450 4.7.0 "Mail sending rate exceeded.  Try again later"
fi
```

This function is a low-level interface. Instead of using it directly, we advise to use the `rateok` function, described below.

`boolean rateok (string` *key*`, number` *sample-interval*`,` [Library Function]
`number` *threshold*`,`
[number *min-samples*])

To use this function, require the `rateok` module (see Section 4.24 [Modules], page 108), e.g.: `require rateok`.

The `rateok` function returns '`True`' if the mail sending rate for *key*, computed for the interval of *sample-interval* seconds is less than the *threshold*. Optional *min-samples* parameter supplies the minimal number of mails needed to obtain the statistics. It defaults to 4.

See Section 3.12 [Sending Rate], page 24, for a detailed description of the `rateok` and its use. The `interval` function (see [interval], page 130) is often used in the second argument to `rateok` or `rate`.

`boolean tbf_rate (string` *key*`, number` *cost*`, number` [Built-in Function]
sample-interval`, number` *burst-size*`)`

This function implements a classical token bucket filter algorithm. Tokens are added to the bucket identified by the *key* at constant rate of 1 token per *sample-interval* microseconds, to a maximum of *burst-size* tokens. If no bucket is found for the specified key, a new bucket is created and initialized to contain *burst-size* tokens. If

[5] It is made optional in order to provide backward compatibility with the releases of mailfromd prior to 5.0.93.

the bucket contains *cost* or more tokens, *cost* tokens are removed from it and `tbf_rate` returns '`True`'. Otherwise, the function returns '`False`'.

For a detailed description of the Token Bucket Algorithm and its use to limit mail rates, see [TBF], page 25.

5.33 Greylisting functions

`boolean greylist (string` *`key`*`, number` *`interval`*`)` [Built-in Function]
Returns '`True`' if the *key* is found in the greylist database (controlled by `database greylist` configuration file statement, see Section 7.11 [conf-database], page 229). The argument *interval* gives the greylisting interval in seconds. The function stores the number of seconds left to the end of greylisting period in the global variable `greylist_seconds_left`. See Section 3.13 [Greylisting], page 27, for a detailed explanation.

The function `greylist` can signal `e_dbfailure` exception.

`boolean is_greylisted (string` *`key`* [Built-in Function]
Returns '`True`' if the *key* is still greylisted. If '`true`' is returned, the function also stores the number of seconds left to the end of greylisting period in the global variable `greylist_seconds_left`.

This function is available only if Con Tassios implementation of greylisting is used. See [greylisting types], page 28, for a discussion of available greylisting implementations. See Section 4.5.5 [greylist], page 57, for a way to switch to Con Tassios implementation.

5.34 Special Test Functions

`boolean portprobe (string` *`host`*`, [number` *`port`*`])` [Library Function]
`boolean listens (string` *`host`*`, [number` *`port`*`])` [Library Function]
Returns `true` if the IP address or host name given by *host* argument listens on the port number *port* (default 25).

This function is defined in the module `portprobe`.

`boolean validuser (string` *`name`*`)` [Built-in Function]
Returns `true` if authenticity of the user *name* is confirmed using mailutils authentication system. See Section 3.14 [Local Account Verification], page 29, for more details.

`boolean valid_domain (string` *`domain`*`)` [Library Function]
Returns `true` if the domain name *domain* has a corresponding A record or if it has any '`MX`' records, i.e. if it is possible to send mail to it.

To use this function, require the `valid_domain` module (see Section 4.24 [Modules], page 108):

```
require valid_domain
```

`number heloarg_test (string arg, string remote_ip, string local_ip)` [Library Function]

Verify if an argument of '`HELO`' ('`EHLO`') command is valid. To use this function, require the `heloarg_test` module (see Section 4.24 [Modules], page 108).

Arguments:

arg '`HELO`' ('`EHLO`') argument. Typically, the value of `$s` Sendmail macro;

remote_ip IP address of the remote client. Typically, the value of `$client_addr` Sendmail macro;

local_ip IP address of this SMTP server;

The function returns a number describing the result of the test, as described in the following table.

Code	Meaning
HELO_SUCCESS	*arg* successfully passes all tests.
HELO_MYIP	*arg* is our IP address.
HELO_IPNOMATCH	*arg* is an IP, but it does not match the remote party IP address.
HELO_ARGNORESOLVE	*arg* is an IP, but it does not resolve.
HELO_ARGNOIP	*arg* is in square brackets, but it is not an IP address.
HELO_ARGINVALID	*arg* is not an IP address and does not resolve to one.
HELO_MYSERVERIP	*arg* resolves to our server IP.
HELO_IPMISMATCH	*arg* does not resolve to the remote client IP address.

5.35 Mail Sending Functions

The mail sending functions are new interfaces, introduced in version 3.1.

The underlying mechanism for sending mail, called *mailer*, is specified by `--mailer` command line option. This global setting can be overridden using the last optional argument to a particular function. In any case, the mailer is specified in the form of a URL.

Mailer URL begins with a protocol specification. Two protocol specifications are currently supported: '`sendmail`' and '`smtp`'. The former means to use a `sendmail`-compatible program to send mails. Such a program must be able to read mail from its standard input and must support the following options:

`-oi` Do not treat '`.`' as message terminator.

`-f addr` Use *addr* as the address of the sender.

`-t` Get recipient addresses from the message.

These conditions are met by most existing MTA programs, such as `exim` or `postfix` (to say nothing of `sendmail` itself).

Following the protocol specification is the *mailer location*, which is separated from it with a colon. For the '`sendmail`' protocol, the mailer location sets the full file name of the Sendmail-compatible MTA binary, for example:

```
sendmail:/usr/sbin/sendmail
```

A special form of a sendmail URL, consisting of protocol specification only (‘`sendmail:`’) is also allowed. It means “use the sendmail binary from the `_PATH_SENDMAIL` macro in your `/usr/include/paths.h` file”. This is the default mailer.

The ‘`smtp`’ protocol means to use an SMTP server directly. In this case the mailer location consists of two slashes, followed by the IP address or host name of the SMTP server, and, optionally, the port number. If the port number is present, it is separated from the rest of URL by a colon. For example:

```
smtp://remote.server.net
smtp://remote.server.net:24
```

`void send_mail (string msg [, string to, string from, string mailer])` [Built-in Function]

Sends message *msg* to the email address *to*. The value of *msg* must be a valid RFC 2822 message, consisting of headers and body. Optional argument *to* can contain several email addresses. In this case the message will be sent to each recipient specified in *to*. If it is not specified, recipient addresses will be obtained from the message headers.

Other optional arguments are:

from Sets the sender address. By default ‘`<>`’ is used.

mailer The URL of the mailer to use

Sample usage:

```
set message <<- EOT
      Subject: Test message
      To: Postmaster <postmaster@gnu.org.ua>
      From: Mailfromd <devnull@gnu.org.ua>
      X-Agent: %__package__ (%__version__)

      Dear postmaster,

      This is to notify you that our /etc/mailfromd.mfl
      needs a revision.
      --
      Mailfromd filter administrator
EOT
send_mail(message, "postmaster@gnu.org.ua")
```

`void send_text (string text, string headers [, string to, string from, string mailer])` [Built-in Function]

A more complex interface to mail sending functions.

Mandatory arguments:

text Text of the message to be sent.

headers Headers for the message.

Optional arguments:

to Recipient email addresses.

from Sender email address.

mailer URL of the mailer to use.

The above example can be rewritten using `send_text` as follows:

```
set headers << -EOT
      Subject: Test message
      To: Postmaster <postmaster@gnu.org.ua>
      From: Mailfromd <devnull@gnu.org.ua>
      X-Agent: %__package__ (%__version__)
EOT
set text <<- EOT
      Dear postmaster,

      This is to notify you that our /etc/mailfromd.mfl
      needs a revision.
      --
      Mailfromd filter administrator
EOT
send_text(text, headers, "postmaster@gnu.org.ua")
```

`void send_message (number` *`msg`* `[string` *`to`*`, string` *`from`*`, string` *`mailer`*`])` [Built-in Function]

Send the message identified by descriptor *msg* (see Section 5.19 [Message functions], page 156).

Optional arguments are:

to Recipient email addresses.

from Sender email address.

mailer URL of the mailer to use.

`void send_dsn (string` *`to`*`, string` *`sender`*`, string` *`rcpt`*`, string` *`text`* `[, string` *`headers`*`, string` *`from`*`, string` *`mailer`*`])` [Built-in Function]

This is an experimental interface which will change in the future versions. It sends a message disposition notification (RFC 2298, RFC 1894), of type '`deleted`' to the email address *to*. Arguments are:

to Recipient email address.

sender Original sender email address.

rcpt Original recipient email address.

text Notification text.

Optional arguments:

headers Message headers

from Sender address.

mailer URL of the mailer to use.

`void create_dsn (string sender, string rcpt, string text [, string headers, string from])` [Built-in Function]

Creates DSN message and returns its descriptor. Arguments are:

sender	Original sender email address.
rcpt	Original recipient email address.
text	Notification text.
headers	Message headers
from	Sender address.

5.36 Blacklisting Functions

The functions described in this subsection allow to check whether the given IP address is listed in certain *black list* DNS zone.

`boolean match_dnsbl (string address, string zone, string range)` [Library Function]

This function looks up *address* in the DNS blacklist zone *zone* and checks if the return falls into the given *range* of IP addresses.

It is intended as a replacement for the Sendmail macros '`dnsbl`' and '`enhdnsbl`'.

To use `match_dnsbl`, require the `match_dnsbl` module (see Section 4.24 [Modules], page 108).

Arguments:

address	IP address of the SMTP server to be tested.
zone	FQDN of the DNSbl zone to test against.
range	The range of IP addresses in CIDR notation or the word '`ANY`', which stands for '`127.0.0.0/8`'.

The function returns `true` if dns lookup for *address* in the zone *dnsbl* yields an IP that falls within the range, specified by *cidr*. Otherwise, it returns `false`.

This function raises the following exceptions: `e_invip` if *address* is invalid and `e_invcidr` if *cidr* is invalid.

`boolean match_rhsbl (string email, string zone, string range)` [Library Function]

This function checks if the IP address, corresponding to the domain part of *email* is listed in the RHS DNS blacklist zone *zone*, and if so, whether its record falls into the given range of IP addresses *range*.

It is intended as a replacement for the Sendmail macro '`rhsbl`' by Derek J. Balling.

To use this function, require the `match_rhsbl` module (see Section 4.24 [Modules], page 108).

Arguments:

email	E-mail address, whose domain name should be tested (usually, it is `$f`)
zone	Domain name of the RHS DNS blacklist zone.
range	The range of IP addresses in CIDR notation.

5.37 SPF Functions

Sender Policy Framework, or SPF for short, is an extension to SMTP protocol that allows to identify forged identities supplied with the `MAIL FROM` and `HELO` commands. The framework is explained in detail in RFC 4408 (`http://tools.ietf.org/html/rfc4408`) and on the SPF Project Site (`http://www.openspf.org/`). The following description is a short introduction only, and the users are encouraged to refer to the original specification for the detailed description of the framework.

The domain holder publishes an *SPF record* – a special DNS resource record that contains a set of rules declaring which hosts are, and are not, authorized to use a domain name for `HELO` and `MAIL FROM` identities. This resource record is usually of type `TXT`.[6]

The MFL script can verify if the identity matches the published SPF record by calling `check_host` function and analyzing its return code. The function can be called either in `helo` or in `envfrom` handler. Its arguments are:

ip The IP address of the SMTP client that is emitting the mail. Usually it is `$client_addr`.

domain The domain that provides the sought-after authorization information; Normally it is the domain portion of the `MAIL FROM` or `HELO` identity.

sender The `MAIL FROM` identity.

helo_domain
The `HELO` identity.

my_domain
The SMTP domain served by the local server.

The function returns a numeric result code. For convenience, all possible return values are defined as macros in module `spf.mfl`. The table below describes each value along with the recommended actions for it:

`None` A result of `None` means that no records were published by the domain or that no checkable sender domain could be determined from the given identity. The checking software cannot ascertain whether or not the client host is authorized. Such a message can be subject to further checks that will decide about its fate.

`Neutral` The domain owner has explicitly stated that he cannot or does not want to assert whether or not the IP address is authorized. This result must be treated exactly like `None`; the distinction between them exists only for informational purposes

`Pass` The client is authorized to send mail with the given identity. The message can be subject to further policy checks with confidence in the legitimate use of the identity or it can be accepted in the absence of such checks.

`Fail` The client is not authorized to use the domain in the given identity. The proper action in this case can be to mark the message with a header explicitly stating it is spam, or to reject it outright.

[6] Although RFC 4408 introduces a special `SPF` record type for this purpose, it is not yet widely used. As of version 9.2, MFL does not support `SPF` DNS records.

If you choose to reject such mails, we suggest to use `reject 550 5.7.1`, as recommended by RFC 4408. The reject can return either a default explanation string, or the one supplied by the domain that published the SPF records, as in the example below:

```
reject 550 5.7.1 "SPF check failed:\n%spf_explanation"
```

(for the description of `spf_explanation`, see [spf_explanation], page 197)

`SoftFail` The domain believes the host is not authorized but is not willing to make that strong of a statement. This result code should be treated as somewhere in between a `Fail` and a `Neutral`. It is not recommended to reject the message based solely on this result.

`TempError`
A transient error occurred while performing SPF check. The proper action in this case is to accept or temporarily reject the message. If you choose the latter, we suggest to use SMTP reply code of '`451`' and DSN code '`4.4.3`', for example:

```
tempfail 451 4.4.3
         "Transient error while performing SPF verification"
```

`PermError`
This result means that the domain's published records could not be correctly interpreted. This signals an error condition that requires manual intervention to be resolved, as opposed to the `TempError` result.

The following example illustrates the use of SPF verification in `envfrom` handler:

```
require 'status'
require 'spf'

prog envfrom
do
  switch check_host($client_addr, domainpart($f), $f, $s)
  do
  case Fail:
    string text ""
    if spf_explanation != ""
      set text "%text\n%spf_explanation"
    fi
    reject 550 5.7.1 "SPF MAIL FROM check failed: %text"

  case Pass:
    accept

  case TempError:
    tempfail 451 4.4.3
             "Transient error while performing SPF verification"

  default:
    on poll $f do
```

```
      when success:
        accept
      when not_found or failure:
        reject 550 5.1.0 "Sender validity not confirmed"
      when temp_failure:
        tempfail 450 4.7.0 "Temporary failure during sender verification"
      done
    done
done
```

The SPF support is implemented in MFL in two layers: a built-in layer that provides basic support, and a library layer that provides a convenience wrapper over the library function.

The library layer is implemented in the module `spf.mfl` (see Section 4.24 [Modules], page 108).

The rest of this node describes available SPF functions and variables.

`number spf_check_host (string` *`ip`*`, string` *`domain`*`,` [Built-in Function]
`string` *`sender`*`, string` *`helo_domain`*`, string` *`my_domain`*`)`

This function is the basic implementation of the `check_host` function, defined in RFC 4408, chapter 4. It fetches SPF records, parses them, and evaluates them to determine whether a particular host (*ip*) is or is not permitted to send mail from a given email address (*sender*). The function returns an *SPF result code*.

Arguments are:

ip The IP address of the SMTP client that is emitting the mail. Usually it is `$client_addr`.

domain The domain that provides the sought-after authorization information; Normally it is the domain portion of the `MAIL FROM` or `HELO` identity.

sender The `MAIL FROM` identity.

helo_domain
The `HELO` identity.

my_domain
The SMTP domain served by the local server.

Before returning the `spf_check_host` function stores additional information in global variables:

`spf_explanation`
If the result code is `Fail`, this variable contains the explanation string as returned by the publishing domain, prefixed with the value of the global variable `spf_explanation_prefix`.

For example, if `spf_explanation_prefix` contains '`The domain %{o} explains: `', and the publishing domain '`example.com`' returns the explanation string '`Please see`

`http://www.example.com/mailpolicy.html`', than the value of `spf_explanation` will be:

```
The domain example.com explains:
Please see http://www.example.com/mailpolicy.html
```

(see RFC 4408 (`http://tools.ietf.org/html/rfc4408`), chapter 8, for the description of SPF macro facility).

`spf_mechanism`
: Name of the SPF mechanism that decided about the result code of the SPF record. If one or more '`include`' or '`redirect`' mechanisms were traversed before arriving at that mechanism, their values are appended in the reverse order.

`number spf_test_record (string` *`record`*`, string` *`ip`*`, string` *`domain`*`, string` *`sender`*`, string` *`helo_domain`*`, string` *`my_domain`*`)` [Built-in Function]

Evaluate SPF record *record* as if it were published by *domain*. The rest of arguments are the same as for `spf_check_host` above.

This function is designed primarily for testing and debugging purposes. You would hardly need to use it.

The `spf_test_record` function sets the same global variables as `spf_check_host`.

`number check_host (string` *`ip`*`, string` *`domain`*`, string` *`sender`*`, string` *`helo`*`)` [Library Function]

This function implements the `check_host` function, defined in RFC 4408, chapter 4. It fetches SPF records, parses them, and evaluates them to determine whether a particular host (*ip*) is or is not permitted to send mail from a given email address (*sender*). The function returns an *SPF result code*.

This function is a wrapper over the built-in `spf_check_host`.

The arguments are:

ip
: The IP address of the SMTP client that is emitting the mail. Usually it is the same as the value of `$client_addr`.

domain
: The domain that provides the sought-after authorization information; Normally it is the domain portion of the `MAIL FROM` or `HELO` identity.

sender
: The `MAIL FROM` identity.

helo
: The `HELO` identity.

`string spf_status_string (number` *`code`*`)` [Library Function]

Converts numeric SPF result *code* to its string representation.

`string spf_explanation` [Built-in variable]

If `check_host` (or `spf_check_host` or `spf_test_record`) returned `Fail`, this variable contains the explanation string as returned by the publishing domain, prefixed with the value of the global variable `spf_explanation_prefix`.

For example, if `spf_explanation_prefix` contains '`The domain %{o} explains: `', and the publishing domain '`example.com`' returns the explanation string '`Please see`

`http://www.example.com/mailpolicy.html`', than the value of `spf_explanation` will be:

```
The domain example.com explains:
Please see http://www.example.com/mailpolicy.html
```

`string spf_mechanism` [Built-in variable]

Set to the name of a SPF mechanism that decided about the result code of the SPF record.

`string spf_explanation_prefix` [Built-in variable]

The prefix to be appended to the explanation string before storing it in the `spf_explanation` variable. This string can contain valid SPF macros (see RFC 4408 (`http://tools.ietf.org/html/rfc4408`), chapter 8), for example:

```
set spf_explanation_prefix "%{o} explains: "
```

The default value is '`""`' (an empty string).

5.38 DKIM

DKIM or *DomainKeys Identified Mail* is an email authentication method that allows recipients to verify if an email was authorized by the owner of the domain that email claims to originate from. It does so by adding a digital signature which is verified using a public key published as a DNS TXT record. For technical details about DKIM, please refer to RFC 6376 (`http://tools.ietf.org/html/rfc6376`).

MFL provides functions for DKIM signing and verification.

`number dkim_verify (number` *msg*`)` [Built-in Function]

Verifies the message *msg* (a message descriptor, obtained from a call to `current_message`, `mailbox_get_message`, `message_from_stream` or a similar function).

Return value (constants defined in the '`status`' module):

`DKIM_VERIFY_OK` [dkim_verify status]

The message contains one or more '`DKIM-Signature`' headers and one of them verified successfully.

`DKIM_VERIFY_PERMFAIL` [dkim_verify status]

The message contains one or more '`DKIM-Signature`' headers, all of which failed to verify.

`DKIM_VERIFY_TEMPFAIL` [dkim_verify status]

The message was not signed using DKIM, or the DNS query to obtain the public key failed, or an internal software error occurred during verification.

The following two global variables are always set upon return from this function: `dkim_explanation` and `dkim_explanation_code`. These can be used to clarify the verification result to the end user. The variable `dkim_signing_algorithm` is initialized with the name of the algorithm used to sign the message.

Upon successful return, the variable `dkim_verified_signature` is set to the value of the successfully verified DKIM signature.

`string dkim_signing_algorithm` [Built-in variable]

Name of the algorithm used to sign the message (either '`rsa-sha1`' or '`rsa-sha256`'). If the algorithm was not specified (e.g. the signature is malformed), this variable is assigned an empty value.

`string dkim_explanation` [Built-in variable]

An explanatory message clarifying the verification result.

`number dkim_explanation_code` [Built-in variable]

A numeric code corresponding to the '`dkim_explanation`' string. Its possible values are defined in '`status.mfl`':

`DKIM_EXPL_OK` [DKIM explanation code]

'`DKIM verification passed`'

`DKIM_EXPL_OK` [DKIM explanation code]

'`DKIM verification passed`'

`DKIM_EXPL_NO_SIG` [DKIM explanation code]

'`No DKIM signature`'

`DKIM_EXPL_INTERNAL_ERROR` [DKIM explanation code]

'`internal error`'

`DKIM_EXPL_SIG_SYNTAX` [DKIM explanation code]

'`signature syntax error`'

`DKIM_EXPL_SIG_MISS` [DKIM explanation code]

'`signature is missing required tag`'

According to the DKIM specification, required tags are: `a=`, `b=`, `bh=`, `d=`, `h=`, `s=`, `v=`.

`DKIM_EXPL_DOMAIN_MISMATCH` [DKIM explanation code]

'`domain mismatch`'

The domain part of the `i=` tag does not match and is not a subdomain of the domain listed in the `d=` tag.

`DKIM_EXPL_BAD_VERSION` [DKIM explanation code]

'`incompatible version`'

Incompatible DKIM version listed in the `v=` tag.

`DKIM_EXPL_BAD_ALGORITHM` [DKIM explanation code]

'`unsupported signing algorithm`'

Either the `a=` tag of the DKIM signature contains an unsupported algorithm (currently supported algorithms are: '`rsa-sha1`' and '`rsa-sha256`') or this algorithm, while being supported by `mailfromd`, is not listed in the `h=` tag of the public DKIM key.

`DKIM_EXPL_BAD_QUERY` [DKIM explanation code]

'`unsupported query method`'

The `q=` tag of the public DKIM key contains something other than '`dns/txt`'.

`DKIM_EXPL_FROM` [DKIM explanation code]
: '`From field not signed`'

`DKIM_EXPL_EXPIRED` [DKIM explanation code]
: '`signature expired`'

`DKIM_EXPL_DNS_UNAVAIL` [DKIM explanation code]
: '`public key unavailable`'

`DKIM_EXPL_DNS_NOTFOUND` [DKIM explanation code]
: '`public key not found`'

`DKIM_EXPL_KEY_SYNTAX` [DKIM explanation code]
: '`key syntax error`'

`DKIM_EXPL_KEY_REVOKED` [DKIM explanation code]
: '`key revoked`'

`DKIM_EXPL_BAD_BODY` [DKIM explanation code]
: '`body hash did not verify`'

`DKIM_EXPL_BAD_BASE64` [DKIM explanation code]
: '`can't decode b= tag`'

  Base64 decoding of the `b=` tag failed.

`DKIM_EXPL_BAD_SIG` [DKIM explanation code]
: '`signature did not verify`'

`DKIM_EXPL_BAD_KEY_TYPE` [DKIM explanation code]
: '`unsupported public key type`'

  The `k=` tag of the public DKIM signature contains a value, other than '`rsa`'.

`string dkim_verified_signature` [Built-in variable]
: Upon successful return from the `dkim_verify` function, this variable holds the value of the successfully verified DKIM header. This value is unfolded and all whitespace is removed from it.

An example of using the '`dkim_verify`' function:

```
require status
require dkim

prog eom
do
  string result
  switch dkim_verify(current_message())
  do
     case DKIM_VERIFY_OK:
         set result "pass; verified for " .
                    dkim_verified_signature_tag('i')
     case DKIM_VERIFY_PERMFAIL:
         set result "fail (%dkim_explanation)"
     case DKIM_VERIFY_TEMPFAIL:
         set result "neutral"
  done
  header_add("X-Verification-Result", "dkim=%result")
done
```

The ‘dkim’ module defines convenience functions for manipulating with DKIM signatures:

dkim_signature_tag (string *sig*, string *tag*) [Library Function]

Extracts the value of the tag *tag* from the DKIM signature *sig*. Signature must be normalized by performing the header unwrapping and removing whitespace characters.

If the tag was not found, returns empty string, unless *tag* is one of the tags listed in the table below. If any of these tags are absent, the following values are returned instead:

Tag	**Default value**
c	‘simple/simple’
q	‘dns/txt’
i	‘@’ + the value of the ‘d’ tag.

string dkim_verified_signature_tag (string *tag*) [Library Function]

Returns the value of tag *tag* from the ‘dkim_verified_signature’ variable.

void dkim_sign (string *d*, string *s*, string *keyfile*, [string *ch*, string *cb*, string *headers*, string *algo*]) [Built-in Function]

This function is available only in the eom handler.

Signs the current message. *Notice*, that no other modification should be attempted on the message after calling this function. Doing so would make the signature invalid.

Mandatory arguments:

d Name of the domain claiming responsibility for an introduction of a message into the mail stream. It is also known as the signing domain identifier (SDID).

s The selector name. This value, along with *d* identifies the location of the DKIM public key necessary for verifying the message. The public key is stored in the DNS TXT record for

```
s._domainkey.d
```

keyfile Name of the disk file that keeps the private key for signing the message. The file must be in PKCS#1 or PKCS#8 format (PEM formatted).

Optional arguments:

ch Canonicalization algorithm for message headers. Valid values are: '`simple`' and '`relaxed`'. '`simple`' is the default.

cb Canonicalization algorithm for message body. Valid and default values are the same as for *ch*.

headers A colon-separated list of header field names that identify the header fields that must be signed. Optional whitespace is allowed at either side of each colon separator. Header names are case-insensitive. This list must contain at least the '`From`' header.

It may contain names of headers that are not present in the message being signed. This provides a way to explicitly assert the absence of a header field. For example, if *headers* contained '`X-Mailer`' and that header is not present in the message being signed, but is added by a third party later, the signature verification will fail.

Similarly, listing a header field name once more than the actual number of its occurrences in a message allows you to prevent any further additions. For example, if there is a single '`Comments`' header field at the time of signing, putting '`Comments:Comments:`' in the *headers* parameter is sufficient to prevent any surplus '`Comments`' headers from being added later on.

Multiple instances of the same header name are allowed. They mean that multiple occurrences of the corresponding header field will be included in the header hash. When such multiple header occurrences are referenced, they will be presented to the hashing algorithm in the reverse order. E.g. if the *header* list contained '`Received:Received`') and the current message contained three '`Received`' headers:

```
Received: A
Received: B
Received: C
```

then these headers will be signed in the following order:

```
Received: C
Received: B
```

The default value for this parameter, split over multiple lines for readability, is as follows:

- "From:From:"
- "Reply-To:Reply-To:"

- "Subject:Subject:"
- "Date:Date:"
- "To:"
- "Cc:"
- "Resent-Date:"
- "Resent-From:"
- "Resent-To:"
- "Resent-Cc:"
- "In-Reply-To:"
- "References:"
- "List-Id:"
- "List-Help:"
- "List-Unsubscribe:"
- "List-Subscribe:"
- "List-Post:"
- "List-Owner:"
- "List-Archive"

algo Signing algorithm: either '`rsa-sha256`' or '`rsa-sha1`'. Default is '`rsa-sha256`'.

An example of using this function:

```
precious string domain "example.org"
precious string selector "s2048"
prog eom
do
  dkim_sign("example.org", "s2048", "/etc/pem/my-private.pem",
            "relaxed", "relaxed", "from:to:subject")
done
```

Note on interaction of dkim_sign with Sendmail

When sending a signed message, it is critical that no other modifications be applied to the message after it has been signed. Unfortunately, it is not always the case when `mailfromd` is used with `Sendmail`. Before sending the message over SMTP, `Sendmail` reformats the headers that contain a list of email addresses, by applying to them a procedure called in its parlance *commaization*. The following headers are modified: `Apparently-To`, `Bcc`, `Cc`, `Disposition-Notification-To`, `Errors-To`, `From`, `Reply-To`, `Resent-Bcc`, `Resent-Cc`, `Resent-From`, `Resent-Reply-To`, `Resent-Sender`, `Resent-To`, `Sender`, `To`. Thus, if your `dkim_sign` includes any of these in the signature (which is the default) and some of them happen to be formatted other way than the one `Sendmail` prefers, the DKIM signature would not verify on the recipient side. To prevent this from happening, `dkim_sign` mimics the Sendmail behavior and reformats those headers before signing the message. This should ensure that the message signed and the message actually sent are the same. This default behavior is controlled by the following global variable:

`number dkim_sendmail_commaize` [Built-in variable]

"Commaize" the address headers (see the list above) of the message the same way Sendmail does, and then sign the resulting message.

The default value is 1 (`true`). You can set it to 0 (`false`) if this behavior is not what you want (e.g. if you are using `postfix` or some other MTA).

Note on interaction of dkim_sign with MMQ

The functions `header_add` and `header_insert` (see Section 5.11 [Header modification functions], page 148) as well as the `add` action (see [header manipulation], page 93) cannot interact properly with `dkim_sign` due to the shortcomings of the Milter API. If any of these was called, `dkim_sign` will throw the `e_badmmq` exception with the following diagnostics:

`MMQ incompatible with dkim_sign: `*`op`*` on `*`h`*`, value `*`v`*

where *op* is the operation code ('`ADD HEADER`' or '`INSERT HEADER`'), *h* is the header name and *v* is its value.

The following example shows one graceful way of handling such exception:

```
prog eom
do
    try
    do
        dkim_sign("example.org", "s2048", "/etc/pem/my-private.pem")
    done
    catch e_badmmq
    do
        # Purge the message modification queue
        mmq_purge()
        # and retry
        dkim_sign("example.org", "s2048", "/etc/pem/my-private.pem")
    done
done
```

See Section 5.13 [Message modification queue], page 151, for a discussion of the message modification queue.

5.38.1 Setting up a DKIM record

Follow these steps to set up your own DKIM record:

1. Generate a key pair:

   Use the `openssl genrsa` command. Run:

   ```
   openssl genrsa -out private.pem 2048
   ```

   The last argument is the size of the private key to generate in bits.

2. Extract the public key:

   ```
   openssl rsa -in private.pem -pubout -outform PEM -out public.pem
   ```

3. Set up a DKIM record in your domain:

   A DKIM record is a TXT type DNS record that holds the public key part for verifying messages. Its format is defined in RFC 4871[7]. The label for this record is composed as follows:

   ```
   s._domainkey.d
   ```

   where *d* is your domain name, and *s* is the selector you chose to use. You will use these two values as parameters to the `dkim_sign` function in your `eom` handler. E.g. if your domain in '`example.com`' and selector is '`s2048`', then the DKIM TXT record label is '`s2048._domainkey.example.com`'.

   The public key file generated in step 2 will have the following contents:

   ```
   -----BEGIN PUBLIC KEY-----
   base64
   -----END PUBLIC KEY-----
   ```

   where *base64* is the key itself in base64 encoding. The minimal DKIM TXT record will be:

   ```
   "v=DKIM1; p=base64"
   ```

   The only mandatory *tag* is in fact '`p=`'. The use of '`v=`' is recommended. More tags can be added as needed. In particular, while testing the DKIM support, it is advisable to add the '`t=y`' tag.

5.39 Sockmap Functions

Socket map (*sockmap* for short) is a special type of database used in Sendmail and MeTA1. It uses a simple server/client protocol over INET or UNIX stream sockets. The server listens on a socket for queries. The client connects to the server and sends it a query, consisting of a *map name* and a *key* separated by a single space. Both map name and key are sequences of non-whitespace characters. The map name serves to identify the type of the query. The server replies with a response consisting of a *status indicator* and *result*, separated by a single space. The result part is optional.

For example, following is the query for key '`smith`' in map '`aliases`':

```
11:aliases news,
```

A possible reply is:

```
18:OK root@domain.net,
```

This reply means that the key '`news`' was found in the map, and the value corresponding to that key is '`root@domain.net`'.

The following reply means the key was not found:

```
8:NOTFOUND,
```

For a detailed description of the sockmap protocol, see Section "Protocol" in *Smap manual*.

The MFL library provides two primitives for dealing with sockmaps. Both primitives become available after requiring the `sockmap` module.

[7] `https://tools.ietf.org/html/rfc4871`

`string sockmap_lookup (number fd, string map, string key)` [Library Function]

This function look ups the *key* in the *map*. The *fd* refers to the sockmap to use. It must be obtained as a result of a previous call to `open` with the URL of the sockmap as its first argument (see Section 5.7 [I/O functions], page 137). For example:

```
number fd open("@ unix:///var/spool/meta1/smap/socket")
string ret sockmap_query(fd, "aliases", $rcpt_to)
if ret matches "OK (.+)"
  set alias \1
fi
close(fd)
```

`string sockmap_single_lookup (string url, string map, string key)` [Library Function]

This function connects to the sockmap identified by the *url*, queries for *key* in *map* and closes the connection. It is useful when you need to perform only a single lookup on the sockmap.

5.40 National Language Support Functions

The *National Language Support* functions allow you to write your scripts in such a way, that any textual messages they display are automatically translated to your native language, or, more precisely, to the language required by your current locale.

This section assumes the reader is familiar with the concepts of program *internationalization* and *localization*. If not, please refer to Section "The Purpose of GNU `gettext`" in *GNU gettext manual*, before reading further.

In general, internationalization of any MFL script follows the same rules as described in the *GNU gettext manual*. First of all, you select the program *message domain*, i.e. the identifier of a set of translatable messages your script contain. This identifier is then used to select appropriate translation. The message domain is set using `textdomain` function. For the purposes of this section, let's suppose the domain name is '`myfilter`'. All NLS functions are provided in the `nls` module, which you need to require prior to using any of them.

To find translations of textual message to the current locale, the underlying `gettext` mechanism will look for file *dirname*`/`*locale*`/LC_MESSAGES/`*domainname*`.mo`, where *dirname* is the message catalog hierarchy name, *locale* is the locale name, and *domainname* is the name of the message domain. By default *dirname* is `/usr/local/share/locale`, but you may change it using `bindtextdomain` function. The right place for this initial NLS setup is in the '`begin`' block (see Section 4.15.1 [begin/end], page 76). To summarize all the above, the usual NLS setup will look like:

```
require nls

begin
do
  textdomain("myfilter")
  bindtextdomain("myfilter", "/usr/share/locale");
```

```
done
```

For example, given the settings above, and supposing the environment variable `LC_ALL` is set to '`pl`', translations will be looked in file `/usr/share/locale/pl/LC_MESSAGES/myfilter.mo`.

Once this preparatory work is done, you can request each message to be translated by using `gettext` function, or `_` (underscore) macro. For example, the following statement will produce translated textual description for '`450`' response:

```
tempfail 450 4.1.0 _("Try again later")
```

Of course it assumes that the appropriate `myfile.mo` file already exists. If it does not, nothing bad happens: in this case the macro `_` (as well as `gettext` function) will simply return its argument unchanged, so that the remote party will get the textual message in English.

The '`mo`' files are binary files created from '`po`' source files using `msgfmt` utility, as described in Section "Producing Binary MO Files" in *GNU gettext manual*. In turn, the format of '`po`' files is described in Section "The Format of PO Files" in *GNU gettext manual*.

`string bindtextdomain (string` *`domain`*`, string` *`dirname`*`)` [Built-in Function]

This function sets the base directory of the hierarchy containing message catalogs for a given message domain.

domain is a string identifying the textual domain. If it is not empty, the base directory for message catalogs belonging to domain *domain* is set to *dirname*. It is important that *dirname* be an absolute pathname; otherwise it cannot be guaranteed that the message catalogs will be found.

If *domain* is '`""`', `bindtextdomain` returns the previously set base directory for domain *domain*.

The rest of this section describes the NLS functions supplied in the `nls` module.

`string dgettext (string` *`domain`*`, string` *`msgid`*`)` [Built-in Function]

`dgettext` attempts to translate the string *msgid* into the currently active locale, according to the settings of the textual domain *domain*. If there is no translation available, `dgettext` returns *msgid* unchanged.

`string dngettext (string` *`domain`*`, string` *`msgid`*`,` `string` *`msgid_plural`*`, number` *`n`*`)` [Built-in Function]

The `dngettext` functions attempts to translate a text string into the language specified by the current locale, by looking up the appropriate singular or plural form of the translation in a message catalog, set for the textual domain *domain*.

See Section "Additional functions for plural forms" in *GNU gettext utilities*, for a discussion of the plural form handling in different languages.

`string textdomain (string` *`domain`*`)` [Library Function]

The `textdomain` function sets the current message domain to *domain*, if it is not empty. In any case the function returns the current message domain. The current

domain is '`mailfromd`' initially. For example, the following sequence of `textdomain` invocations will yield:

```
textdomain("") ⇒ "mailfromd"
textdomain("myfilter") ⇒ "myfilter"
textdomain("") ⇒ "myfilter"
```

`string gettext (string` *`msgid`*`)` [Library Function]

`gettext` attempts to translate the string *msgid* into the currently active locale, according to the settings of the current textual domain (set using `textdomain` function). If there is no translation available, `gettext` returns *msgid* unchanged.

`string ngettext (string` *`msgid`*`, string` *`msgid_plural`*`,` [Library Function]
`number` *`n`*`)`

The `ngettext` functions attempts to translate a text string into the language specified by the current locale, by looking up the appropriate singular or plural form of the translation in a message catalog, set for the current textual domain.

See Section "Additional functions for plural forms" in *GNU gettext utilities*, for a discussion of the plural form handling in different languages.

5.41 Syslog Interface

The basic means for outputting diagnostic messages is the '`echo`' instruction (see Section 4.19.4 [Echo], page 95), which sends its arguments to the currently established logging channel. In daemon mode, the latter is normally connected to syslog, so any echoed messages are sent there with the facility selected in mailfromd configuration and priority '`info`'.

If you want to send a message to another facility and/or priority, use the '`syslog`' function:

`void syslog (number` *`priority`*`, string` *`text`*`)` [Built-in Function]

Sends *text* to syslog. The priority argument is formed by ORing the facility and the level values (explained below). The facility level is optional. If not supplied, the currently selected logging facility is used.

The facility specifies what type of program is logging the message, and the level indicates its relative severity. The following symbolic facility values are declared in the `syslog` module: '`LOG_KERN`', '`LOG_USER`', '`LOG_MAIL`', '`LOG_DAEMON`', '`LOG_AUTH`', '`LOG_SYSLOG`', '`LOG_LPR`', '`LOG_NEWS`', '`LOG_UUCP`', '`LOG_CRON`', '`LOG_AUTHPRIV`', '`LOG_FTP`' and '`LOG_LOCAL0`' through '`LOG_LOCAL7`'

The declared severity levels are: '`LOG_EMERG`', '`LOG_ALERT`', '`LOG_CRIT`', '`LOG_ERR`', '`LOG_WARNING`', '`LOG_NOTICE`', '`LOG_INFO`' and '`LOG_DEBUG`'.

5.42 Debugging Functions

These functions are designed for debugging the MFL programs.

`void debug (string` *`spec`*`)` [Built-in Function]

Enable debugging. The value of *spec* sets the debugging level. See [debugging level specification], page 45, for a description of its format.

For compatibility with previous versions, this function is also available under the name '`mailutils_set_debug_level`'.

`number debug_level ([string srcname])` [Built-in Function]

This function returns the debugging level currently in effect for the source module *srcname*, or the global debugging level, if called without arguments.

For example, if the program was started with `--debug='all.trace5; engine.trace8'` option, then:

```
debug_level() ⇒ 127
debug_level("engine") ⇒ 1023
debug_level("db") ⇒ 0
```

`boolean callout_transcript ([boolean value])` [Built-in Function]

Returns the current state of the callout SMTP transcript. The result is 1 if the transcript is enabled and 0 otherwise. The transcript is normally enabled either by the use of the `--transcript` command line option (see [SMTP transcript], page 48) or via the '`transcript`' configuration statement (see Section 7.5 [conf-server], page 223).

The optional *value*, supplies the new state for SMTP transcript. Thus, calling '`callout_transcript(0)`' disables the transcript.

This function can be used in bracket-like fashion to enable transcript for a certain part of MFL program, e.g.:

```
number xstate callout_transcript(1)
on poll $f do
  ...
done
set xstate callout_transcript(0)
```

Note, that the use of this function (as well as the use of the `--transcript` option) makes sense only if callouts are performed by the `mailfromd` daemon itself. It will not work if a dedicated callout server is used for that purpose (see Chapter 10 [calloutd], page 249).

`string debug_spec ([string catnames, bool showunset])` [Built-in Function]

Returns the current debugging level specification, as given by `--debug` command line option or by the `debug` configuration statement (see Section 7.7 [conf-debug], page 225).

If the argument *srcnames* is specified, it is treated as a semicolon-separated list of categories for which the debugging specification is to be returned.

For example, if `mailfromd` was started with `--debug=all.trace5;spf.trace1; engine.trace8;db.trace0`, then:

```
debug_spec() ⇒ "all.trace5,engine.trace8"
debug_spec("all;engine") ⇒ "all.trace5,engine.trace8"
debug_spec("engine;db") ⇒ "db.trace0;engine.trace8"
debug_spec("prog") ⇒ ""
```

When called without arguments, `debug_spec` returns only those categories which have been set, as shown in the first example above.

Optional *showunset* parameters controls whether to return unset module specifications. To print all debugging specifications, whether set or not, use

```
debug_spec("", 1)
```

These three functions are intended to complement each other. The calls to `debug` can be placed around some piece of code you wish to debug, to enable specific debugging information for this code fragment only. For example:

```
/* Save debugging level for dns.c source */
set dlev debug_spec("dns", 1)
/* Set new debugging level */
debug("dns.trace8")
.
.
.
/* Restore previous level */
debug(dlev)
```

`void program_trace (string` *`module`*`)` [Built-in Function]
Enable tracing for a set of modules given in *module* argument. See [–trace-program], page 238, for a description of its format.

`void cancel_program_trace (string` *`module`*`)` [Built-in Function]
Disable tracing for given modules.

This pair of functions is also designed to be used together in a bracket-like fashion. They are useful for debugging `mailfromd`, but are not advised to use otherwise, since tracing slows down the execution considerably.

`void stack_trace ()` [Built-in Function]
Generate a stack trace in this point. See [tracing runtime errors], page 50, for the detailed description of stack traces.

`string stack_trace_format (string` *`fmt`*`)` [Built-in Function]
Format each frame in stack trace to a string according to *fmt*, and return the concatenated strings. The format string *fmt* consists of regular characters, which are copied to the resulting string verbatim, and *conversion specifiers*, prefixed with percent sign, which expand to parts of stack trace. The following conversion specifiers are defined:

`%d` Frame number, counting from the innermost (last) frame, 0-based.

`%f` Name of the function.

`%n` Line number in the source file.

`%s` Name of the source file.

`%p` Value of the PC register.

`%%` Single percent sign.

A '`%`' followed by any other letter is reproduced verbatim.

`string stack_trace_top (string fmt)` [Built-in Function]
Format the top-level stack frame according to *fmt*. See above for a discussion of format string.

The functions below are intended mainly for debugging MFL run-time engine and for use in `mailfromd` testsuite. You will hardly need to use them in your programs.

`void _expand_dataseg (number n)` [Built-in Function]
Expands the run-time data segment by at least *n* words.

`number _reg (number r)` [Built-in Function]
Returns the value of the register *r* at the moment of the call. Symbolic names for run-time registers are provided in the module `_register`:

Name	**Register**
REG_PC	Program counter
REG_TOS	Top of stack
REG_TOH	Top of heap
REG_BASE	Frame base
REG_REG	General-purpose accumulator
REG_MATCHSTR	Last matched string pointer

`number _stack_free ()` [Built-in Function]
Returns number of words available for use in stack. This is the same as

```
_reg(REG_TOS) - _reg(REG_TOH)
```

`number _heap_reserve (number n)` [Built-in Function]
Use up next *n* words in the heap. Return the address of the first word.

`void _wd ([number n])` [Built-in Function]
Enters a time-consuming loop and waits there for *n* seconds (by default – indefinitely). The intention is to facilitate attaching to `mailfromd` with the debugger. Before entering the loop, a diagnostic message is printed on the '`crit`' facility, informing about the PID of the process and suggesting the command to be used to attach to it, e.g.:

```
mailfromd: process 21831 is waiting for debug
mailfromd: to attach: gdb -ex 'set variable mu_wd::_count_down=0'
 /usr/sbib/mailfromd 21831
```

5.43 Informative Functions

These functions convert numeric identifiers of various MFL entities to strings and vice-versa.

`string milter_state_name (number code)` [Built-in Function]
Convert the numeric identifier of a *milter state* to textual form. It is normally used to convert the `milter_state` variable (see [milter state], page 13) to string, e.g.:

```
milter_state_name(5) ⇒ "connect"
```

If *code* does not refer to a valid milter state, the `e_inval` exception is raised.

`number milter_state_code (string name)` [Built-in Function]
Returns numeric code of the milter state *name*. If *name* does not refer to a valid milter state, returns 0 (`milter_state_none` from the `milter.mfl` module).

`string milter_action_name (number code)` [Built-in Function]
Convert the numeric identifier of a reply action (see [reply actions], page 92) to textual name.

If *code* does not refer to a valid reply action, the `e_inval` exception is raised.

```
milter_state_code("connect") ⇒ 5
```

This function is useful in action hooks. See Section 4.15.3 [action hook], page 77, for details.

`number milter_action_name (string name)` [Built-in Function]
Returns numeric code of the reply action identified by *name*. If *name* does not refer to a valid action, returns -1.

5.44 Mfmod Interface Functions

The calls described in this section provide interface for invoking functions defined in a *dynamically loaded library* and retrieving their return values. For a detailed description of this interface and a discussion of its use, see Section 4.25 [mfmod], page 111.

`number dlopen (filename)` [Built-in Function]
Loads the dynamically loaded library *filename* and returns a numeric handle that can be used to call functions from that library.

Unless *filename* is an absolute pathname, it will be looked up in *mfmod search path*, defined by the configuration variable `runtime.mfmod-path` (see [mfmod-path], page 231).

Maximum number of dynamic libraries that can be loaded simultaneously is limited by the configuration variable `runtime.max-mfmods`.

Once open, the library remains loaded until `mailfromd` exits. There is no `dlclose` function, since it is not deemed necessary (at the time of this writing, at least). Therefore, the common practice is to call this function in a `begin` section (see Section 4.15.1 [begin/end], page 76) and assign its return value to a global or static variable, which will then be used by further `dl*` calls in this module.

`any dlcall (number dlh, string symbol, string types, ...)` [Built-in Function]
Calls a *mfmod function symbol* from the library identified by handle *dlh* (a value returned by a previous call to `dlopen`). The *types* parameter defines types of the remaining arguments. It is a list of type letters, one for each argument:

‘`s`’ String value.

‘`n`’
‘`d`’ Numeric value.

‘`m`’ Message.

An example usage:

```
set dlh dlopen("mfmod_test.so")
string x dlcall(dlh, "runtest", "sn", "input string", 3)
```

This example calls the function '`runtest`' from the `mfmod_test.so` library with two arguments: a string '`input string`' and numeric value '`3`' and assings the return value to the variable `x`.

Type of the return value from `dlcall` is determined by the value of `retval.type` upon return from the underlying library function. See Section 4.25.1 [Loadable Library], page 111, for a detailed description.

For more details about using this function, Section 4.25.2 [Interface Module], page 114.

6 Using the GNU Emacs MFL Mode

MFL sources are usual ASCII files and you may edit them with any editor you like. However, the best choice for this job (as well as for many others) is, without doubt, GNU Emacs. To ease the work of editing script files, the `mailfromd` package provides a special Emacs mode, called *MFL mode*.

The elisp source file providing this mode, `mfl-mode.el`, is installed automatically, provided that GNU Emacs is present on your machine. To enable the mode, add the following lines to your Emacs setup file (either system-wide `site-start.el`, or your personal one, `~/.emacs`):

```
(autoload 'mfl-mode "mfl-mode")
(setq auto-mode-alist (append auto-mode-alist
                      '(("\\.mfl$" . mfl-mode))))
```

The first directive loads the MFL mode, and the second one tells Emacs to apply it to any file whose name ends in '`.mfl`' suffix.

MFL mode provides automatic indentation and syntax highlighting for MFL sources. The default indentation setup is the same as the one used throughout this book:

- Handler and function definitions start at column 1;
- A block statement, i.e. '`do`', '`done`', '`if`', '`else`', '`elif`' and '`fi`', occupies a line by itself, with the only exception that '`do`' after an '`on`' statement is located on the same line with it;
- A '`do`' statement that follows function or handler definition is placed in column 1.
- Each subsequent level of nesting is indented two columns to the right (see [mfl-basic-offset], page 216).
- A closing statement ('`done`', '`else`', '`elif`', '`fi`') is placed at the same column as the corresponding opening statement;
- Branch statements ('`case`' and '`when`') are placed in the same column as their controlling keyword ('`switch`' and '`on`', correspondingly (see [mfl-case-line-offset], page 216).
- Loop substatements (see Section 4.21 [Loops], page 96) are offset 5 columns to the right from the controlling `loop` keyword. (see [mfl-loop-statement-offset], page 217). Continuation statements within loop header are offset 5 columns from the indentation of their controlling keyword, either `for` or `while` (see [mfl-loop-continuation-offset], page 217).

The mode provides two special commands that help navigate through the complex filter scripts:

`C-M-a` Move to the beginning of current function or handler definition.

`C-M-e` Move to the end of current function or handler definition.

Here, *current function or handler* means the one within which your cursor currently stays.

You can use `C-M-e` repeatedly to walk through all function and handler definitions in your script files. Similarly, repeatedly pressing `C-M-a` will visit all the definitions in the opposite direction (from the last up to the very first one).

Another special command, *C-c C-c*, allows to verify the syntax of your script file. This command runs `mailfromd` in syntax check mode (see Section 3.16 [Testing Filter Scripts], page 33) and displays its output in a secondary window, which allows to navigate through eventual diagnostic messages and to jump to source locations described by them.

All MFL mode settings are customizable. To change any of them, press *M-x customize* and visit '`Environment/Unix/Mfl`' customization group. This group offers two subgroups: '`Mfl Lint group`' and '`Mfl Indentation group`'.

'`Mfl Lint group`' controls invocation of mailfromd by *C-c C-c*. This group contains two variables:

`mfl-mailfromd-command` [MFL-mode setting]
: The `mailfromd` to be invoked. By default, it is '`mailfromd`'. You will have to change it, if `mailfromd` cannot be found using `PATH` environment variable, or if you wish to pass it some special options. However, do not include `--lint` or `-I` options in this variable. The `--lint` option is given automatically, and include paths are controlled by `mfl-include-path` variable (see below).

`mfl-include-path` [MFL-mode setting]
: A list of directories to be appended to `mailfromd` include search path (see [include search path], page 53). By default it is empty.

'`Mfl Indentation group`' controls automatic indentation of MFL scripts. This group contains the following settings:

`mfl-basic-offset` [MFL-mode setting]
: This variable sets the basic indentation increment. It is set to 2, by default, which corresponds to the following indentation style:

```
prog envfrom
do
  if $f = ""
    accept
  else
    ...
  fi
done
```

`mfl-case-line-offset` [MFL-mode setting]
: Indentation offset for `case` and `when` lines, relative to the column of their controlling keyword. The default is 0, i.e.:

```
switch x
do
case 0:
  ...
default:
  ...
done
```

`mfl-returns-offset` [MFL-mode setting]
Indentation offset of `returns` and `alias` statements, relative to the controlling `func` keyword. The default value is 2, which corresponds to:

```
func foo()
  alias bar
  returns string
```

`mfl-comment-offset` [MFL-mode setting]
Indentation increment for multi-line comments. The default value is 1, which makes:

```
 /* first comment line
    second comment line */
```

`mfl-loop-statement-offset` [MFL-mode setting]
Indentation increment for parts of a `loop` statement. The default value is 5, which corresponds to the following style:

```
loop for stmt,
     while cond,
     incr
do
```

`mfl-loop-continuation-offset` [MFL-mode setting]
If any of the `loop` parts occupies several lines, the indentation of continuation lines relative to the first line is controlled by `mfl-loop-continuation-offset`, which defaults to 5:

```
loop for set n 0
          set z 1,
     while n != 10
          or z != 2,
     set n n + 1
```

7 Configuring `mailfromd`

Upon startup, `mailfromd` checks if the file `/etc/mailfromd.conf` exists.[1] If it does, the program attempts to retrieve its configuration settings from that file.

The `mailfromd.conf` file must be written in the *GNU mailutils configuration format*, as described in Section "conf-syntax" in *GNU Mailutils Manual*. This format can be summarized as follows:

Comments
: Inline comments begin with '`//`' or '`#`' and end at the end of the line. Multiline comments are delimited by '`/*`' and '`*/`'. Multiline comments cannot be nested, but can contain inline comment markers.

Empty lines and whitespace
: Empty lines are ignored. Whitespace characters (i.e. horizontal, vertical space, and newline) are ignored, except as they serve to separate tokens.

Statements
: A statement consists of a keyword and a value, separated by whitespace. Statements terminate with a semicolon. E.g.

```
pidfile /var/run/mailfromd.pid;
```

Block statements
: A block statement consists of a keyword and a list of statements enclosed in '`{`' and '`}`' characters. Optional *label* can appear between the keyword and opening curly brace. E.g.:

```
logging {
  syslog on;
  facility mail;
}
```

Block statement is not required to terminate with a semicolon, although it is allowed to.

File Inclusion
: The `include` statement causes inclusion of the file listed as its value:

```
include /usr/share/mailfromd/config.inc;
```

The `mailfromd.conf` file is used by all programs that form the '`mailfromd`' package, i.e. `mailfromd`, `calloutd`, `mfdbtool`, and `pmult`. Since the sets of statements understood by each of them differ, special syntactic means are provided to separate program-specific configurations from each other.

First of all, if the argument to `include` is a directory, then the program will search that directory for a file with the same name as the base name of the program itself. If found, this file will be loaded after finishing parsing the `mailfromd.conf` file. Otherwise, this statement is ignored.

[1] The exact location is determined at compile time: the `/etc` directory is the system configuration directory set when building `mailfromd` (see Chapter 2 [Building], page 9).

Secondly, the special block statement `program tag` is processed only if *tag* matches the base name of the program being run. Again, it is processed after the main `mailfromd.conf` file.

Thus, if you need to provide configuration for the `calloutd` component, there are two ways of doing so. First, you can place it to a file named `calloutd` placed in a separate directory (say, `/etc/mailfromd.d`), and use the name of that directory in a `include` statement in the main configuration file:

```
include /etc/mailfromd.d;
```

Secondly, you can use the `program` statement as follows:

```
program calloutd {
  ...
}
```

7.1 Special Configuration Data Types

In addition to the usual data types (see Section "Statements" in *GNU Mailutils Manual*), `mailfromd` configuration introduces the following two special ones:

time interval specification

The *time interval specification* is a string that defines an interval, much the same way we do this in English: it consists of one or more pairs '`number`'-'`time unit`'. For example, the following are valid interval specifications:

```
1 hour
2 hours 35 seconds
1 year 7 months 2 weeks 2 days 11 hours 12 seconds
```

The pairs can occur in any order, however unusual it may sound to a human ear, e.g. '`2 days 1 year`'. If the '`time unit`' is omitted, seconds are supposed.

Connection URL

unix://*file*
unix:*file*
local://*file*
local:*file* A named pipe (socket).

inet://*address*:*port*
inet:*port*@*address*
An IPv4 connection to host *address* at port *port*. Port must be specified either as a decimal number or as a string representing the port name in `/etc/services`.

inet6:*port*@*address*
An IPv6 connection to host *address* at port *port*. This port type is not yet supported.

7.2 Base Mailfromd Configuration

`script-file` *file* [Mailfromd Conf]

Read filter script from *file*. By default it is `sysconfdir/mailfromd.mfl`.

`setvar` *name value* [Mailfromd Conf]
Initialize MFL variable *name* to *value*.

`include-path` *path* [Mailfromd Conf]
Add directories to the list of directories to be searched for include files. See [include search path], page 53.

Argument is a list of directory names separated by colons. These names are added to the search path before the default search path directories, but after any directories added by the previous `include-path` statements, if any.

See [–include-path], page 234,

`module-path` *path* [Mailfromd Conf]
Add directories to the module search path See [module search path], page 109.

Argument is a list of directory names separated by colons. These names are added to the search path before the default search path directories, but after any directories added by the previous `module-path` statements, if any.

See [–module-path], page 234,

`state-directory` *dir* [Mailfromd Conf]
Set program state directory. See [statedir], page 11.

`relayed-domain-file` *file* [Mailfromd Conf]
Append domain names from the named *file* to the list of relayed domains. This list can be inspected from MFL script using the '`relayed`' function (see [relayed], page 179).

The *file* argument is either a single file name or a list of file names, e.g.:

```
relayed-domain-file /etc/mail/sendmail.cw;
relayed-domain-file (/etc/mail/sendmail.cw, /etc/mail/relay-domains);
```

`source-ip` *ipaddr* [Mailfromd Conf]
Set source IP address for outgoing TCP connections.

`pidfile` *file* [Mailfromd Conf]
Set the name of the file to store PID value in. The file must be writable for the user or group `mailfromd` runs as (see Section 7.10 [conf-priv], page 229).

7.3 Preprocessor Configuration

MFL preprocessor is configured using the `preprocessor` statement (default values shown):

```
preprocessor {
  # Enable preprocessor
  enable yes;
  # Preprocessor command line stub.
  command "m4 -s";
  # Pass current include path to the preprocessor via -I options.
  pass-includes false;
  # Pass to the preprocessor the feature definitions via -D options
  # as well as any -D/-U options from the command line
  pass-defines true;
  # Name of the preprocessor setup file.  Unless absolute, it is
  # looked up in the include path.
  setup-file "pp-setup";
}
```

Its substatements are:

`enable` *bool* [preprocessor]
: Enable or disable preprocessor. If disabled in configuration file, preprocessor can be forcefully enabled in the command line by using the `--preprocessor` option.

`command` *string* [preprocessor]
: The stub for the preprocessor command line. Before use, it will be eventually augmented by `-I` and `-D` options, depending on the `pass-includes` and `pass-defines` settings in this section. This is described in detail in Section 4.26.1 [Configuring Preprocessor], page 117.

`pass-includes` *bool* [preprocessor]
: If `true` the directories from include search path (see [include search path], page 53, will be passed to the preprocessor via the `-I` options.

`pass-defines` *bool* [preprocessor]
: If `true`, the -D options defining optional features will be passed to the preprocessor (e.g. `-DWITH_DKIM`), as well as any `-D` and `-U` options from the `mailfromd` command line.

`setup-file` *string* [preprocessor]
: Name of the preprocessor setup file (see Section 4.26 [Preprocessor], page 116). Unless *string* begins with a slash, the file with this name is looked up in the current include search path (see [include search path], page 53). If found, its absolute name is passed to the preprocessor as its first argument.

  If *string* begins with a slash, it is passed to the preprocessor as is.

7.4 DNS Resolver Configuration

DNS resolver settings are configured using the `resolver` compound statement:

```
resolver {
  config filename;
  max-cname-chain num;
}
```

`config` *filename* [resolver]
: Name of the resolver configuration file to use, instead of the default `/etc/resolv.conf`.

`max-cname-chain` *num* [resolver]
: Maximum allowed length of a DNS *CNAME chain* that will be followed. A *CNAME chain* is a sequence of `CNAME` records pointing to another `CNAME`s. Although CNAME chains are not considered a good practice, many sites still use them. By default the `mailfromd` resolver allows at most one CNAME record pointing to a CNAME (this corresponds to `max-cname-chain 2`). If you need to follow longer chains, raise this value. Note however, that using values greater than 5 is not a good idea, anyway.

7.5 Server Configuration

A single `mailfromd` daemon can run several *servers*. These are configured in the following statement:

```
server type {
  id name;
  listen url;
  backlog num;
  max-instances num;
  single-process bool;
  reuseaddr bool;
  option list;
  default bool;
  callout url;
  acl { ... }
}
```

`server` *type* [Mailfromd Conf]
: Define a server. The *type* is either '`milter`' or '`callout`'. See Section 3.7 [SMTP Timeouts], page 19, for a description of various types of servers.

The substatements in the `server` block provide parameters for configuring this server.

`id` *name* [server]
: Assign an identifier to this server. This identifier is used as a suffix to syslog tag (see [syslog tag], page 44) in messages related to this server. For example, if a server block had the following statement in it:

```
id main;
```

then all messages related to this server will be marked with tag '`mailfromd#main`'.

The part before the '`#`' is set using the `tag` statement in `logging` block (see Section "Logging Statement" in *GNU Mailutils Manual*).

This identifier can be inspected from the MFL code using the `milter_server_id` variable (see [milter_server_id], page 67).

`listen` *url* [server]
: Listen for connections on the given URL. See [milter port specification], page 220, for a description of allowed *url* formats.

Example:

```
listen inet://10.10.10.1:3331;
```

The MFL code can access its address using the `milter_server_address` and `milter_server_family` variables.

`backlog` *num* [server]
Configures the size of the queue of pending connections. Default value is 8.

`max-instances` *number* [server]
Sets the maximum number of instances allowed for this server.

`single-process` *bool* [server]
When set to '`yes`', this server will run in *single-process* mode, i.e. it will not fork subprocesses to serve requests. This option is meant exclusively to assist in debugging `mailfromd`. Don't use it for anything else but for debugging!

`reuseaddr` *bool* [server]
When set to '`yes`', `mailfromd` will attempt to reuse existing socket addresses. This is the default behavior.

If the server *type* is '`callout`', the following statement is also allowed:

`option` *list* [server]
Configures server options. As of version 9.2 only one option is defined:

default Mark this server as the default one. This means it will be used by every milter server that doesn't define the `callout-url` statement.

`default` *bool* [server]
When set to '`yes`', this server is marked as a default callout server for all milter servers declared in the configuration. This is equivalent to `option default`.

if the server *type* is '`milter`', you can use the following statement to query a remote callout server:

`callout` *url* [server]
Use a callout server at *url* (see [milter port specification], page 220).

You can also set a *global callout server*, which will be used by all milter servers that do not set the `callout` statement:

`callout-url` *url* [Mailfromd Conf]
Set global callout server. See [milter port specification], page 220, for allowed *url* formats.

7.6 Milter Connection Configuration

`milter-timeout` *time* [Mailfromd Conf]

Sets the timeout value for connection between the filter and the MTA. Default value is 7210 seconds. You normally do not need to change this value.

`acl` [Mailfromd Conf]

This block statement configures *access control list* for incoming Milter connections. See Section "ACL Statement" in *GNU Mailutils Manual*, for a description of its syntax. E.g.:

```
acl {
  allow from 10.10.10.0/24;
  deny from any;
}
```

7.7 Logging and Debugging configuration

`logger` *mech* [Mailfromd Conf]

Set default logger mechanism. Allowed values for *mech* are:

stderr Log everything to the standard error.

syslog Log to syslog.

syslog:async
Log to syslog using the asynchronous syslog implementation.

See Section 3.19 [Logging and Debugging], page 43, for a detailed discussion. See also [syslog-async], page 11, for information on how to set default syslog implementation at compile time.

`debug` *spec* [Mailfromd Conf]

Set `mailfromd` debug verbosity level. The *spec* must be a valid debugging level specification (see [debugging level specification], page 45).

`stack-trace` *bool* [Mailfromd Conf]

Enables stack trace dumps on runtime errors. This feature is useful for locating the source of an error, especially in complex scripts. See [tracing runtime errors], page 50, for a detailed description.

`trace-actions` *bool* [Mailfromd Conf]

Enable action tracing. If *bool* is '`true`', `mailfromd` will log all executed actions. See Section 3.19 [Logging and Debugging], page 43, for a detailed description of action tracing.

`trace-program` *modlist* [Mailfromd Conf]

Enable program instruction tracing for modules in *modlist*, a comma-separated list of source code modules, e.g.:

```
trace-program (bi_io,bi_db);
```

This statement enables tracing for functions from modules `bi_io.c` and `bi_db.c` (notice, that you need not give file suffixes).

This tracing is useful for debugging `mailfromd`, but is not advised to use otherwise, since it is very time-costly.

`transcript` *bool* [Mailfromd Conf]
Enable transcripts of call-out SMTP sessions. See [SMTP transcript], page 48, for a detailed description of SMTP transcripts.

7.8 Timeout Configuration

The SMTP timeouts used in callout sessions are configured via `smtp-timeout` statement:

Syntax

```
smtp-timeout type {
  connection interval;
  initial-response interval;
  helo interval;
  mail interval;
  rcpt interval;
  rset interval;
  quit interval;
}
```

`smtp-timeout` *type* [Mailfromd Conf]
Declare SMTP timeouts of the given *type*, which may be '`soft`' or '`hard`'.

Callout SMTP sessions initiated by polling functions, are controlled by two sets of timeouts: '`soft`' and '`hard`'. *Soft timeouts* are used by the mailfromd milter servers. *Hard timeouts* are used by callout servers (see [callout server], page 19). When a soft timeout is exceeded, the calling procedure is delivered an '`e_temp_failure`' exception and the session is scheduled for processing by a callout server. The latter re-runs the session using hard timeouts. If a hard timeout is exceeded, the address is marked as '`not_found`' and is stored in the cache database with that status.

Normally, soft timeouts are set to shorter values, suitable for use in MFL scripts without causing excessive delays. Hard timeouts are set to large values, as requested by RFC 2822 and guarantee obtaining a definite answer (see below for the default values).

Statements

The *time* argument for all `smtp-timeout` sub-statements is expressed in time interval units, as described in [time interval specification], page 220.

`connection` *time* [smtp-timeout]
Sets initial connection timeout for callout tests. If the connection is not established within this time, the corresponding callout function returns temporary failure.

`initial-response` *time* [smtp-timeout]
Sets the time to wait for the initial SMTP response.

`helo` *time* [smtp-timeout]
Timeout for a response to '`HELO`' (or '`EHLO`') command.

`mail` *time* [smtp-timeout]
Timeout for a response to '`MAIL`' command.

`rcpt` *time* [smtp-timeout]
Timeout for a response to '`RCPT`' command.

`rset` *time* [smtp-timeout]
Timeout for a response to '`RSET`' command.

`quit` *time* [smtp-timeout]
Timeout for a response to '`QUIT`' command.

Default Values

The default timeout settings are:

Timeout	**Soft**	**Hard**
connection	10s	5m
initial-response	30s	5m
helo	I/O	5m
mail	I/O	10m
rcpt	I/O	5m
rset	I/O	5m
quit	I/O	2m

Table 7.1: Default SMTP timeouts

`io-timeout` *time* [Mailfromd Conf]
Sets a general SMTP I/O operation timeout. This timeout is used as the default for entries marked with '`I/O`' in the above table. The default is 3 seconds.

7.9 Call-out Configuration

`ehlo-domain` *string* [Mailfromd Conf]
Sets default domain used in '`EHLO`' (or '`HELO`') SMTP command when probing the remote host. This value can be overridden by '`from`' parameter to `poll` command (see [poll], page 107).

This statement assigns the value *string* to the '`ehlo_domain`' variable (see [ehlo_domain], page 68), and is therefore equivalent to

```
setvar ehlo_domain string;
```

`mail-from-address` *string* [Mailfromd Conf]
Sets default email addresses used in '`MAIL FROM:`' SMTP command when probing the remote host. This value can be overridden by '`as`' parameter to `poll` command (see [poll], page 107).

This statement assigns the value *string* to the '`mailfrom_address`' variable (see [mailfrom_address], page 69), and is therefore equivalent to

```
setvar mailfrom_address string;
```

`enable-vrfy` *bool* [Mailfromd Conf]

Enables the use of SMTP VRFY statement prior to normal callout sequence. If VRFY is supported by the remote server, `mailfromd` relies on its reply and does not perform normal callout.

The use of this statement is not recommended, because many existing VRFY implementations always return affirmative result, no matter is the requested email handled by the server or not.

The default is `enable-vrfy no`, i.e. VRFY is disabled.

`smtp-starttls` *string* [Mailfromd Conf]

Configures whether to issue the `STARTTLS` command if the mail server offers such capability. Allowed values are:

`never` [smtp-starttls value]

Never use `STARTTLS`.

`always` [smtp-starttls value]

Always use `STARTTLS` if supported by the server.

`ondemand` [smtp-starttls value]

Use `STARTTLS` only if `MAIL FROM:` command failed with the code 530 (Authorization required).

Default is '`ondemand`'.

Notice, that the `smtp-starttls` feature depends on whether GnuTLS support is available in `libmailutils`. You can check whether it is available by inspecting the output of `mailfromd --show-defaults` (see Section 3.18 [Examining Defaults], page 40): if so, the '`optional features`' line will contain the word '`STARTTLS`'.

`tls` { ... } [Mailfromd Conf]

Configures TLS settings for the callout. This is a compound statement. The two most important statements in this compound are:

`ssl-priorities` *string* [Mailfromd TLS]

Configures the TLS session's handshake algorithms and options in a compact, easy-to-use format. See Section "Priority strings" in *GnuTLS*, for a detailed description of the priority string format.

Default value is '`NORMAL:%COMPAT`'. You may need to adjust it in order to work with older or misconfigured servers, e.g.:

```
tls {
  ssl-priorities "LEGACY:%COMPAT";
}
```

`handshake-timeout` *n* [Mailfromd TLS]

Sets the timeout for TLS handshake to *n* seconds.

The remaining three statements are not of much importance for callout. They are described here for completeness sake:

`ssl-ca-file` *file* [Mailfromd TLS]
: Specifies the pathname of the certificate authority file.

`ssl-certificate-file` *file* [Mailfromd TLS]
: Specifies the pathname of the certificate file.

`ssl-key-file` *file* [Mailfromd TLS]
: Specifies the pathname of the certificate key file.

7.10 Privilege Configuration

`user` *name* [Mailfromd Conf]
: Switch to this user's privileges after startup. See Section 8.2 [Starting and Stopping], page 239, for a discussion of the privileges `mailfromd` runs under and the options that affect them. See also `group` below.

`group` *name* [Mailfromd Conf]
: Retain the supplementary group *name* when switching to user privileges. By default `mailfromd` clears the list of supplementary groups when switching to user privileges, but this statement allows to retain the given group. It can be specified multiple times to retain several groups. This option may be necessary to maintain proper access rights for various files. See Section 8.2 [Starting and Stopping], page 239.

7.11 Database Configuration

Syntax

```
database dbname {
  file name;
  enable bool;
  expire-interval interval;
  positive-expire-interval interval;
  negative-expire-interval interval;
}
```

`database` *dbname* [Mailfromd Conf]
: The `database` statement controls run-time parameters of the DBM database identified by *dbname*. Allowed values for the latter are:

`cache`	Callout cache database.
`rate`	Sending rate database. See Section 3.12 [Sending Rate], page 24.
`tbf`	Token-bucket filter database. See [TBF], page 25.
`greylist`	Greylisting database. See Section 3.13 [Greylisting], page 27.

Statements

`file` *name* [database]
: Set the database file name. The file name can be prefixed with the *DBM scheme*, which indicates the desired DBM implementation to use. The scheme consists of

DBM type name followed by a colon and two slashes. To obtain the list of available DBM types, run `mailfromd --show-defaults` (see Section 3.18 [Examining Defaults], page 40) and examine the '`supported database types:`' line, e.g.:

```
$ mailfromd --show-defaults | grep '^supported database types:'
supported database types:gdbm, bdb
```

If no DBM scheme is supplied, the default DBM implementation will be used. The default implementation is set up on compile time and can be inspected using the following command:

```
mailfromd --show-defaults | grep '^default database type:'
```

The part that follows DBM scheme can be either absolute or relative file name. Absolute file name (begining with '`/`') is used verbatim. Relative file name (i.e. the name that begins with any character except '`/`') is modified by prepending it with the mailfromd local state directory (see [statedir], page 11).

To illustrate the above, suppose that the local state directory is `/var/mailfromd` and default DBM implementation is `gdbm`. Then:

Value of `file`	**Resulting filename**	**DBM type**
`tbf.db`	/var/mailfromd/tbf.db	GDBM
`bdb://tbf.db`	/var/mailfromd/tbf.db	Berkeley DB
`/run/tbf.db`	/run/tbf.db	GDBM
`bdb:///run/tbf.db`	/run/tbf.db	Berkeley DB

`enable` *bool* [database]

Enable or disable this database.

`expire-interval` *time* [database]

Set the expiration interval for this database *dbname*. See [time interval specification], page 220, for a description of *time* format.

`positive-expire-interval` *time* [database]

This statement is valid only for '`cache`' database. It sets the expiration interval for positive ('`success`') cache entries.

`negative-expire-interval` *time* [database]

This statement is valid only for '`cache`' database, where it sets expiration interval for negative ('`not_found`') cache entries.

Additional Statements

`database-type` *type* [Mailfromd Conf]

Set default database type. *type* is one of the database types supported by mailutils (i.e., for Mailutils 3.0: '`gdbm`', '`ndbm`', '`bdb`', '`kc`', and '`tc`'). Run

```
mailfromd --show-defaults | grep 'supported databases:'
```

to get a list of type names supported by your build of `mailfromd` (see Section 3.18 [Examining Defaults], page 40).

`database-mode` *mode* [Mailfromd Conf]

Defines file mode for newly created database files. *mode* must be a valid file mode in octal.

7.12 Runtime Constants Configuration

`runtime` { *statements* } [Mailfromd Conf]
The statements in the `runtime` section configure various values used by MFL builtin functions.

`max-streams` *number* [runtime]
Sets the maximum number of stream descriptors that can be opened simultaneously. Default is 1024. See Section 5.7 [I/O functions], page 137.

`max-open-mailboxes` *number* [runtime]
Sets the maximum number of available mailbox descriptors. This value is used by MFL mailbox functions (see Section 5.18 [Mailbox functions], page 154).

`max-open-messages` *number* [runtime]
Sets the maximum number of messages that can be opened simultaneously using the `mailbox_get_message` function. See Section 5.19 [Message functions], page 156, for details.

`max-mfmods` *number* [runtime]
Maximum number of dynamically loaded modules (*mfmod*s) that can be used simultaneously. See Section 4.25 [mfmod], page 111.

`mfmod-path` *string* [runtime]
Search path for dynamically loaded modules – a colon-delimited list of directories where to look for libraries loaded using the `dlopen` function (see [dlopen], page 213). Default `mfmod-path` value is *`prefix`*`/mailfromd/lib`, where *prefix* is the installation prefix directory. See Section 4.25 [mfmod], page 111, for details.

7.13 Standard Mailutils Statements

The following standard Mailutils statements are understood:

Statement	**Reference**
auth	See Section "auth statement" in *GNU Mailutils Manual*.
debug	See Section "debug statement" in *GNU Mailutils Manual*.
include	See Section "include" in *GNU Mailutils Manual*.
logging	See Section "logging statement" in *GNU Mailutils Manual*.
mailer	See Section "mailer statement" in *GNU Mailutils Manual*.
locking	See Section "locking statement" in *GNU Mailutils Manual*.

8 `Mailfromd` Command Line Syntax

The `mailfromd` binary is started from the command line using the following syntax (brackets indicate optional parts):

```
$ mailfromd [options] [asgn] [script]
```

The meaning of each invocation part is described in the table below:

options The command line options (see Section 8.1 [options], page 233).

asgn Sendmail macro assignments. These are currently meaningful only with the `--test` option (see [test mode], page 34), but this may change in the future. Each assignment has the form:

```
var=value
```

where *var* is the name of a Sendmail macro and *value* is the value to be assigned to it.

script The file name of the filter script, if other than the default one.

8.1 Command Line Options.

8.1.1 Operation Modifiers

`--daemon` Run in daemon mode (default).

`--run[=start]`

Load the script named in the command line and execute the function named *start*, or '`main`', if *start* is not given.

The full invocation syntax for this mode is:

```
mailfromd [options] --run[=start \
          [macro=value] [file args...]
```

Its parts are:

options Any mailfromd options needed to tune its behavior.

start Start symbol. Defaults to `main`.

macro=value

Any number of Sendmail macro definitions. *macro* is the macro name and *value* is the value to initialize it to.

file Script file name. If omitted, default script file will be used (see [default script file], page 10).

args... Command line arguments that will be pased to the *start* MFL function as parameters.

The *start* function must be defined as:

```
func start (...) returns number
```

See Section 3.17 [Run Mode], page 35, for a detailed discussion of this feature.

`-t[`*state*`]`
`--test[=`*state*`]`
: Run in test mode. See Section 3.16 [Testing Filter Scripts], page 33. Default *state* is '`envfrom`'. This option implies `--stderr` (see [–stderr], page 239).

8.1.2 General Settings

`--callout-socket=`*string*
: Set socket for the default callout server. This is mainly useful together with the `--mtasim` option.

`--echo`
`--echo=`*file*
: This option controls where the output of the `echo` statement (see Section 4.19.4 [Echo], page 95) goes when `mailfromd` is run in test mode, i.e. using the `--test` or `--run` options (see Section 3.16 [Testing Filter Scripts], page 33).

  When used without argument it directs the `echo` output to the standard output stream. If an argument is supplied (second form above), the output goes to the named file. The file will be created if it doesn't exist. Notice, that the use of '`=`' is compulsory when specifying an argument. A single dash as a filename means standard output, so that `--echo=-` and `--echo` are equivalent.

`--foreground`
: Stay in foreground. When given this option, `mailfromd` will not disconnect itself from the controlling terminal and will run in the foreground.

`-g` *name*
`--group=`*name*
: Retain the group *name* when switching to user privileges. See Section 8.2 [Starting and Stopping], page 239.

  This option complements the `group` configuration statement (see Section 7.10 [conf-priv], page 229).

`--include-path=`*dir*
`-I` *dir*
: Add the directory *dir* to the list of directories to be searched for include files (see [include search path], page 53). This setting affects the `#include` statement (see Section 4.2 [include], page 53) and is used to locate preprocessor setup file (see Section 4.26 [Preprocessor], page 116). If enabled in the configuration (see Section 7.3 [conf-preprocessor], page 221), directories from the include search path will also be added to the preprocessor command line as additional `-I` options.

`--module-path=`*dir*
`-P` *dir*
: Add the directory *dir* to the list of directories searched for MFL module files (see [module search path], page 109). This setting affects the `require` and `import` MFL statements. See Section 4.24 [Modules], page 108.

`--mailer=`*url*
`-M` *url*
: Set the URL of the mailer to use. See Section 5.35 [Mail Sending Functions], page 191.

`--mtasim` This option is reserved for use by `mtasim` (see Chapter 12 [mtasim], page 261).

`-O[`*`level`*`]`
`--optimize[=`*`level`*`]`
: Set optimization level for code generator. Two levels are implemented: '`0`', meaning no optimization, and '`1`', meaning full optimization. This option is for debugging `mailfromd` code generation, please don't use it in production environment. In particular, notice that `-O0` disables, among others, tracing the use of Sendmail macros (see [macro tracing], page 58), so that you will have to use additional `#pragma miltermacros` to compensate for that.

`-p` *`string`*
`--port=`*`string`*
`--milter-socket=`*`string`*
: Set communication socket. Overrides the `listen` configuration statement, which you are advised to use instead (see Section 7.5 [conf-server], page 223).

`--pidfile=`*`file`*
: Set pidfile name. Overrides the `pidfile` configuration statement, which you are advised to use instead (see Section 7.2 [conf-base], page 220).

`--relayed-domain-file=`*`file`*
: Read relayed domains from *file*. Overrides the `relayed-domain-file` configuration statement (see Section 7.2 [conf-base], page 220), which you are advised to use instead. See Section 3.8 [Avoiding Verification Loops], page 20, and the description of `relayed` function (see [relayed], page 179) for more information.

`--resolv-conf-file=`*`file`*
: Read resolver settings from *file*, instead of the default `/etc/resolv.conf`.

`--state-directory=`*`dir`*
: Set new program state directory. See [statedir], page 11, for the description of this directory and its purposes. This option overrides the settings of `state-directory` configuration statement, described in Section 7.2 [conf-base], page 220.

`-S` *`ip`*
`--source-ip=`*`ip`*
: Set source address for TCP connections. Overrides the '`source-ip`' configuration statement, which you are advised to use instead (see Section 7.2 [conf-base], page 220).

`-u` *`name`*
`--user` *`name`*
: Switch to this user privileges after startup. Overrides the `user` configuration file statement, which you are advised to use instead (see Section 7.10 [conf-priv], page 229). Default user is '`mail`'.

`-v` *`var=value`*
`--variable` *`var=value`*
: Assign *value* to the global variable *var*. The variable must be declared in your startup script. See [overriding initial values], page 34, for a detailed discussion of this option.

8.1.3 Preprocessor Options

Following command line options control the preprocessor feature. See Section 4.26 [Preprocessor], page 116, for a detailed discussion of these.

`--no-preprocessor`
: Do not run the preprocessor.

`--preprocessor[=`*command*`]`
: If *command* is supplied, use it as the external preprocessor instead of the default. This overrides the `preprocessor.command` setting (see Section 7.3 [conf-preprocessor], page 221).

  If used without arguments, forces the use of the configured preprocessor. This form is intended as a way to manually enable the use of the preprocessor if disabled in the configuration.

`-D` *name*`[=`*value*`]`
`--define=`*name*`[=`*value*`]`
: Define a preprocessor symbol *name* to have a value *value*. This option is ignored if the `preprocessor.pass-defines` configuration setting is `false`.

`-U` *name*
`--undefine=`*name*
: Undefine the preprocessor symbol *name*. This option is ignored if the `preprocessor.pass-defines` configuration setting is `false`.

`-E`
: Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

8.1.4 Timeout Control

See [time interval specification], page 220, for information on *interval* format.

`--milter-timeout=`*interval*
: Set MTA connection timeout. Overrides `milter-timeout` statement in the `mailfromd` configuration file, which you are advised to use instead (see Section 7.6 [conf-milter], page 225).

8.1.5 Logging and Debugging Options

`--location-column`
`--no-location-column`
: Mention column number in error messages. See Section 3.16 [location-column], page 33. Use `--no-location-column` to disable

`-d` *string*
`--debug=`*string*
: Set debugging level. See Section 3.19 [Logging and Debugging], page 43.

`--dump-code`
: Parse and compile the script file and dump the disassembled listing of the produced code to the terminal. See Section 3.19 [Logging and Debugging], page 43.

`--dump-grammar-trace`
: Enable debugging the script file parser. While parsing the file, the detailed dump of the parser states and tokens seen will be output.

`--dump-lex-trace`
: Enable debugging the lexical analyzer. While parsing the script file, the detailed dump of the lexer states and matched rules will be output.

`--dump-macros`
: Show Sendmail macros used in the script file. The macro names are displayed as comma-separated lists, grouped by handler names. See Section 9.1 [Sendmail], page 243, for a detailed description of this option and its usage.

`--dump-tree`
: Parse and compile the script file and dump the parse tree in a printable form to the terminal.

`--dump-xref`
: Print a cross-reference of variables used in the filter script. See Section 3.16 [Testing Filter Scripts], page 33.

`-E`
: Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. See Section 4.26 [Preprocessor], page 116.

`--lint`
: Check script file syntax and exit. If the file is OK, return 0 to the shell, otherwise print appropriate messages to stderr and exit with code 78 (‘`configuration error`’).

`--single-process`
: Do not fork sub-processes to serve requests. This option is meant to assist in debugging `mailfromd`. Don’t use it for anything else but for debugging, as it terribly degrades performance!

`--stack-trace`
`--no-stack-trace`
: Add MFL stack trace information to runtime error output. Overrides `stack-trace` configuration statement. Use the `--no-stack-trace` to disable trace information.

  See [tracing runtime errors], page 50, for more information on this feature.

`--gacopyz-log=`*`level`*
: Set desired logging level for `gacopyz` library (see Appendix A [Gacopyz], page 281). There are five logging levels. The following table lists them in order of decreasing priority:

  fatal
  : Log fatal errors.

  err
  : Log error messages.

  warn
  : Log warning messages.

  info
  : Log informational messages. In particular, this enables printing messages on each subprocess startup and termination, which look like that:

```
Apr 28 09:00:11 host mailfromd[9411]: connect
    from 192.168.10.1:50398
Apr 28 09:00:11 host mailfromd[9411]: finishing
    connection
```

This level can be useful for debugging your scripts.

debug Log debugging information.

proto Log Milter protocol interactions. This level prints huge amounts of information, in particular it displays dumps of each Milter packet sent and received.

Although it is possible to set these levels independently of each other, it is seldom practical. Therefore, the option `--gacopyz-log=`*`level`* enables all logging levels from *level* up. For example, `--gacopyz-log=warn` enables log levels '`warn`', '`err`' and '`fatal`'. It is the default. If you need to trace each subprocess startup and shutdown, set `--gacopyz-log=info`. Setting the logging level to '`proto`' can be needed only for `Gacopyz` developers, to debug the protocol.

See Section 3.16 [Testing Filter Scripts], page 33.

`--logger=`*`mech`*
Set logger mechanism (*mech* is one of '`stderr`', '`syslog`', '`syslog:async`'). See Section 3.19 [Logging and Debugging], page 43.

`--log-facility=`*`facility`*
Output logs to syslog *facility*.

`--log-tag=`*`string`*
Tag syslog entries with the given *string*, instead of the program name.

`--source-info`
`--no-source-info`
Include C source information in debugging messages. This is similar to setting `line-info yes` in the `debug` configuration block (see Section "debug statement" in *GNU Mailutils Manual*).

The `--no-source-info` can be used to cancel the effect of the `line-info yes` configuration statement.

You do not need this option, unless you are developing or debugging `mailfromd`.

`--syntax-check`
Synonym for `--lint`.

`--trace`

`--no-trace`
Enable or disable action tracing. If `--trace` is given, `mailfromd` will log all executed actions. See Section 3.19 [Logging and Debugging], page 43.

`--trace-program[=`*`string`*`]`
Enable program instruction tracing. With this option `mailfromd` will log execution of every instruction in the compiled filter program. The optional arguments allows to specify a comma-separated list of source code modules for which the

tracing is to be enabled, for example `--trace-program=bi_io,bi_db` enables tracing for functions from modules `bi_io.c` and `bi_db.c` (notice, that you need not give file suffixes in *string*).

This option is useful for debugging `mailfromd`, but is not advised to use otherwise, since it is very time-costly.

`-X`
`--transcript`
`--no-transcript`
: Enable or disable transcript of the SMTP sessions to the log channel. See Section 3.19 [Logging and Debugging], page 43.

`--syslog` Selects default syslog mechanism for diagnostic output.

`--stderr` Directs all logging to standard output. Similar to `--logger=stderr`.

`--xref` Same as `--dump-xref`. See Section 3.19 [Logging and Debugging], page 43.

8.1.6 Informational Options

`-?`
`--help` Give a short help summary.

`--usage` Give a short usage message.

`-V`
`--version`
: Print program version.

`--show-defaults`
: Show compilation defaults. See Section 3.18 [Examining Defaults], page 40.

8.2 Starting and Stopping

Right after startup, when `mailfromd` has done the operations that require root privileges, it switches to the privileges of the user it is configured to run as (see [default user privileges], page 10) or the one given in its configuration file (see Section 7.10 [conf-priv], page 229). During this process it will drop all supplementary groups and switch to the principal group of that user.

Such limited privileges of the daemon can cause difficulties if your filter script needs to access some files (e.g. `Sendmail` databases) that are not accessible to that user and group. For example, the following fragment using `dbmap` function:

```
if dbmap("/etc/mail/aliases.db", $f, 1)
  ...
fi
```

will normally fail, because `/etc/mail/aliases.db` is readable only to the root and members of the group '`smmsp`'.

In such situations you need to instruct `mailfromd` to retain the privileges of one or several supplementary groups when switching to the user privileges. This is done using

`group` statement in the `mailfromd` configuration file (see Section 7.10 [conf-priv], page 229). In example above, you need to use the following settings:

```
group smmsp;
```

(The same effect can be achieved with `--group` command line option: *`mailfromd --group=smmsp`*).

To stop a running instance of `mailfromd` use one of the following signals: `SIGQUIT`, `SIGTERM`, `SIGINT`. All three signals have the same effect: the program cancels handling any pending requests, deinitializes the communication socket (if it is a UNIX socket, the program unlinks it) and exits.

To restart the running `mailfromd` instance, send it `SIGHUP`. For this to be possible, two conditions must be met: `mailfromd` must be invoked with the absolute file name, and the configuration file name must be abosulute as well. If either of them is not met, `mailfromd` displays a similar warning message:

```
warning: script file is given without full file name
warning: restart (SIGHUP) will not work
```

or:

```
warning: mailfromd started without full file name
warning: restart (SIGHUP) will not work
```

The reaction of `mailfromd` on `SIGHUP` in this case is the same as on the three signals described previously, i.e. cleanup and exit immediately.

The PID of the master instance of `mailfromd` is kept on the *pidfile*, which is named `mailfromd.pid` and is located in the program *state directory*. Assuming the default location of the latter, the following command will stop the running instance of the daemon:

```
kill -TERM `head -n1 /usr/local/var/mailfromd/mailfromd.pid`
```

The default pidfile location is shown in the output of `mailfromd --show-defaults` (see Section 3.18 [Examining Defaults], page 40), and can be changed at run time using `pidfile` statement (see Section 7.2 [conf-base], page 220).

To facilitate the use of `mailfromd`, it is shipped with a shell script that can be used to launch it on system startup and shut it down when the system goes down. The script, called `rc.mailfromd`, is located in the directory `/etc` of the distribution. It takes a single argument, specifying the action that should be taken:

start — Start the program.

stop — Shut down the program

reload — Reload the program, by sending it `SIGHUP` signal.

restart — Shut down the program and start it again.

status — Display program status. It displays the PID of the master process and its command line, for example:

```
$ /etc/rc.d/rc.mailfromd status
mailformd appears to be running at 26030
26030 /usr/local/sbin/mailfromd --group smmsp
```

If the second line is not displayed, this most probably mean that there is a '`stale`' pidfile, i.e. the one left though the program is not running.

An empty *`rc.mailfromd status`* output means that `mailfromd` is not running.

configtest [*file*]
: Check the script file syntax, report any errors found and exit. If *file* is given it is checked instead of the default one.

macros [-c] [*file*]
: Parse the script file (or *file*, if it is given, extract the names of Sendmail macros it uses and generate corresponding export statements usable in the Sendmail configuration file. By default, `mc` statements are generated. If `-c` (`--cf`) is given, the statements for `sendmail.cf` are output. See the next chapter for the detailed description of this mode.

You can pass any additional arguments to `mailfromd` by editing `ARGS` variable near line 22.

The script is not installed by default. You will have to copy it to the directory where your system start-up scripts reside and ensure it is called during the system startup and shut down. The exact instructions on how to do so depend on the operating system you use and are beyond the scope of this manual.

9 Using `mailfromd` with Various MTAs

The following sections describe how to configure various Milter-capable MTAs to work with `mailfromd`.

9.1 Using `mailfromd` with Sendmail.

This chapter assumes you are familiar with Sendmail configuration in general and with Milter configuration directives in particular. It concentrates only on issues, specific for `mailfromd`.

To prepare `Sendmail` to communicate with `mailfromd` you need first to set up the *milter port*. This is done with `INPUT_MAIL_FILTER` statement in your `Sendmail` file:

```
INPUT_MAIL_FILTER(`mailfrom', `S=unix:/usr/local/var/mailfromd/mailfrom')
```

Make sure that the value of '`S`' matches the value of `listen` statement in your `mailfromd.conf` file (see Section 7.6 [conf-milter], page 225). Notice, however, that they may not be literally the same, because `listen` allows to specify socket address in various formats, whereas Sendmail's '`S`' accepts only milter format.

If you prefer to fiddle directly with `sendmail.cf` file, use this statement instead:

```
Xmailfrom, S=unix:/usr/local/var/mailfromd/mailfrom
```

If you are using Sendmail version 8.14.0 or newer, you may skip to the end of this section. These versions implement newer Milter protocol that enables `mailfromd` to negotiate with the MTA the macros it needs for each state.

Older versions of Sendmail do not offer this feature. For Sendmail versions prior to 8.14.0, you need to manually configure `Sendmail` to export macros you need in your `mailfromd.mfl` file. The simplest way to do so is using `rc.mailfromd` script, introduced in the previous chapter. Run it with `macros` command line argument and copy its output to your `sendmail.mc` configuration file:

```
$ rc.mailfromd macros
```

If you prefer to work with `sendmail.cf` directly, use `-c` (`--cf`) command line option:

```
$ rc.mailfromd macros -c
```

Finally, if you use other mailfromd script file than that already installed (for example, you are preparing a new configuration while the old one is still being used in production environment), give its name in the command line:

```
$ rc.mailfromd macros newscript.mfl
# or:
$ rc.mailfromd macros -c newscript.mfl
```

If you use this method, you can skip the rest of this chapter. However, if you are a daring sort of person and prefer to do everything manually, follow the instructions below.

First of all you need to build a list of macros used by handlers in your `mailfromd.mfl` file. You can obtain it running `mailfromd --dump-macros`. This will display all macros used in your handlers, grouped by handler name, for example:

```
envfrom i, f, {client_addr}
envrcpt f, {client_addr}, {rcpt_addr}
```

Now, modify `confMILTER_MACROS_handler` macros in your `mc` file. Here, *handler* means the uppercase name of the `mailfromd` handler you want to export macros to, i.e. the first word on each line of the above `mailfromd --dump-macros` output. *Notice,* that in addition to these macros, you should also export the macro `i` for the very first handler (`rc.mailfromd macros` takes care of it automatically, but you preferred to do everything yourself...) It is necessary in order for `mailfromd` to include '`Message-ID`' in its log messages (see [Message-ID], page 44).

For example, given the above macros listing, which corresponds to our sample configuration (see Section 4.27 [Filter Script Example], page 120), the `sendmail.mc` snippet will contain:

```
define(`confMILTER_MACROS_ENVFROM',dnl
confMILTER_MACROS_ENVFROM `, i, f, {client_addr}')
define(`confMILTER_MACROS_ENVRCPT',dnl
confMILTER_MACROS_ENVRCPT `, f, {client_addr}, {rcpt_addr}')
```

Now, if you are a *really* daring person and prefer to do everything manually *and* to hack your `sendmail.cf` file directly, you certainly don't need any advices. Nonetheless, here's how the two statements above *could* look in this case:

```
O Milter.macros.envfrom=i, {auth_type}, {auth_authen}, \
   {auth_ssf}, {auth_author}, {mail_mailer}, {mail_host}, \
   {mail_addr} ,{mail_addr}, {client_addr}, f
O Milter.macros.envrcpt={rcpt_mailer}, {rcpt_host}, \
   {rcpt_addr} ,i, f, {client_addr}
```

9.2 Using `mailfromd` with MeTA1.

MeTA1 (`http://www.meta1.org`) is an MTA of next generation which is designed to provide the following main features:

- Security
- Reliability
- Efficiency
- Configurability
- Extendibility

Instead of using Sendmail-compatible Milter protocol, it implements a new protocol, called *policy milter*, therefore an additional program is required to communicate with `mailfromd`. This program is a *Pmilter–Milter multiplexer* `pmult`, which is part of the '`Mailfromd`' distribution. See Chapter 13 [pmult], page 271, for a detailed description of its configuration.

The configuration of '`Meta1--Mailfromf`' interaction can be subdivided into three tasks.

1. Configure `mailfromd`

   This was already covered in previous chapters. No special '`MeTA1`'-dependent configuration is needed.

2. Configure `pmult` to communicate with `mailfromd`

   This is described in detail in Chapter 13 [pmult], page 271.

3. Set up MeTA1 to communicate with pmult

   The MeTA1 configuration file is located in /etc/meta1/meta1.conf. Configure the smtps component, by adding the following section:

```
policy_milter {
    socket {
        type = type;
        address = addr;
        [path = path;]
        [port = port-no;]
    };
    [timeout = interval;]
    [flags = { flag };]
};
```

   Statements in square brackets are optional. The meaning of each instruction is as follows:

   type = *type*

   Set the type of the socket to communicate with pmult. Allowed values for *type* are:

   inet — Use INET socket. The socket address and port number are set using the address and port statements (see below).

   unix — Use UNIX socket. The socket path is given by the path statement (see below).

   Notice, that depending on the type setting you have to set up either address/port or path, but not both.

   address = *addr*

   Configure the socket address for type = inet. *Addr* is the IP address on which pmult is listening (see Section 13.1.1 [pmult-conf], page 272).

   port = *port-no*

   Port number pmult is listening on (see Section 13.1.1 [pmult-conf], page 272).

   path = *socket-file*

   Full pathname of the socket file, if type = unix.

   timeout = *interval*

   Sets the maximum amount of time to wait for a reply from pmult.

   The behavior of smtps in case of time out depends on the flags settings:

   flags = { *flag* }

   Flag is one of the following:

   abort — If pmult does not respond, abort the current SMTP session with a '421' error.

   accept_but_reconnect — If pmult does not respond, continue the current session but try to reconnect for the next session.

For example, if the `pmult` configuration has:

```
listen inet://127.0.0.1:3333;
```

then the corresponding part in `/etc/meta1/meta1.conf` will be

```
smtps {
    policy_milter {
        socket {
            type = inet;
            address = 127.0.0.1;
            port = 3333;
        };
        ...
    };
    ...
};
```

Similarly, if the `pmult` configuration has:

```
listen unix:///var/spool/meta1/pmult/socket;
```

then the `/etc/meta1/meta1.conf` should have:

```
smtps {
    policy_milter {
        socket {
            type = unix;
            path = /var/spool/meta1/pmult/socket;
        };
        ...
    };
    ...
};
```

9.3 Using `mailfromd` with Postfix

To configure `postfix` to work with your filter, you need to inform it about the socket your filter is listening on. The `smtpd_milters` (or `non_smtpd_milters`) statement in `/etc/postfix/main.cf` serves this purpose. If the filter is to handle mail that arrives via SMTP, use `smtpd_milters`. If it is to handle mail submitted locally to the queue, use `non_smtpd_milters`. In both cases, the value is a whitespace-separated list of socket addresses. Note, that Postfix syntax for socket addresses differs from that used by Sendmail and mailfromd. The differences are summarized in the following table:

Sendmail	**Mailfromd**	**Postfix**
inet:*port*@*host*	inet://*host*:*port*	inet:*host*:*port*
unix:*file*	unix://*file*	unix:*file*

Table 9.1: Socket addresses in various formats

For example, if your `mailfromd` listens on '`inet://127.0.0.1:4111`', add the following to `/etc/postfix/main.cf`:

```
smtpd_milters = inet:127.0.0.1:4111
```

Mailfromd uses Milter protocol version 6. Postfix, starting from version 2.6 uses the same version. Older versions of Postfix use Milter protocol 2 by default. Normally, it should not be a problem, as mailfromd tries to detect what version the server is speaking. If, however, it fails to select the proper version, you will have to instruct Postfix what version to use. To do so, add the following statement to /etc/postfix/main.cf:

```
milter_protocol = 6
```

The way Postfix handles macros differs from that of Sendmail. Postfix emulates a limited subset of Sendmail macros, and not all of them are are available when you would expect them to. In particular, the 'i' macro is not available before the 'DATA' stage, which brings two consequences. First, mailfromd log messages will not include message ID until the 'DATA' stage is reached. Secondly, you cannot use 'i' in handlers 'connect', 'helo', 'envfrom' and 'envrcpt',

If you wish to tailor Postfix defaults to export the actual macros used by your filter, run mailfromd --dump-macros and filter its output through the postfix-macros.sed filter, which is installed to the *prefix*/share/mailfromd directory, e.g.:

```
$ mailfromd --dump-macros | \
  sed -f /usr/share/mailfromd/postfix-macros.sed
milter_helo_macros = {s}
milter_mail_macros = {client_addr} {s} {f}
milter_rcpt_macros = {rcpt_addr} {f} {client_addr}
milter_end_of_data_macros = {i}
```

Cut and paste its output to your /etc/postfix/main.cf.

For more details regarding Postfix interaction with Milter and available Postfix configuration options, see Postfix before-queue Milter support (http://www.postfix.org/MILTER_README.html).

10 calloutd

The callout verification is usually performed by a special instance of `mailfromd` (see [callout server], page 19). However, it is also possible to set up a dedicated callout server on a separate machine. You can choose to do so, for instance, in order to reduce the load on the server running `mailfromd`.

This stand-alone callout facility is provided by the `calloutd` daemon.

10.1 Calloutd Configuration

Main configuration file `/etc/mailfromd.conf` is used (see Chapter 7 [Mailfromd Configuration], page 219). The configuration statements are basically the same as for `mailfromd`.

The address to listen on is defined in the `server` statement. Basically, it is the only statement the configuration file is required to have. The minimal configuration can look like:

```
program calloutd {
  server {
    listen inet://198.51.100.1:3535;
  }
}
```

To instruct the `mailfromd` daemon to use this server, the following statement should be added to the `/etc/mailfromd.conf` file:

```
program mailfromd {
  callout-url inet://198.51.100.1:3535;
}
```

The `server` statement differs a little from the similar statement for `mailfromd`. This and another `calloutd`-specific statements are described in detail in the subsections that follow. The rest of statements is shared with `mailfromd`. The following table lists all supported configuration statements along with cross-references to the correspondent descriptions:

Statement	**Reference**
`acl`	See Section "acl statement" in *GNU Mailutils Manual*.
`auth`	See Section "auth statement" in *GNU Mailutils Manual*.
`database`	See Section 7.11 [conf-database], page 229.
`database-mode`	See Section 7.11 [conf-database], page 229.
`database-type`	See Section 7.11 [conf-database], page 229.
`debug` (section)	See Section "debug statement" in *GNU Mailutils Manual*.
`debug`	See Section 10.1.3 [conf-calloutd-log], page 251.
`ehlo-domain`	See Section 7.9 [conf-callout], page 227.
`enable-vrfy`	See Section 7.9 [conf-callout], page 227.
`group`	See Section 7.10 [conf-priv], page 229.
`include`	See Section "include" in *GNU Mailutils Manual*.
`io-timeout`	See Section 7.8 [conf-timeout], page 226.

`locking`	See Section "locking statement" in *GNU Mailutils Manual*.
`lock-retry-count`	See Section 7.11 [conf-database], page 229.
`lock-retry-timeout`	See Section 7.11 [conf-database], page 229.
`logger`	See Section 10.1.3 [conf-calloutd-log], page 251.
`logging`	See Section "logging statement" in *GNU Mailutils Manual*.
`mailer`	See Section "mailer statement" in *GNU Mailutils Manual*.
`mail-from-address`	See Section 7.9 [conf-callout], page 227.
`pidfile`	See Section 10.1.1 [conf-calloutd-setup], page 250.
`server`	See Section 10.1.2 [conf-calloutd-server], page 250.
`source-ip`	See Section 10.1.1 [conf-calloutd-setup], page 250.
`smtp-timeout`	See Section 7.8 [conf-timeout], page 226.
`state-directory`	See Section 10.1.1 [conf-calloutd-setup], page 250.
`transcript`	See Section 10.1.3 [conf-calloutd-log], page 251,
`user`	See Section 7.10 [conf-priv], page 229.

10.1.1 `calloutd` General Setup

`source-ip` *IP* [Calloutd Conf]

Sets source IP address for TCP connections.

`pidfile` *filename* [Calloutd Conf]

Defines the name of the file to store PID value in.

`state-directory` *dir* [Calloutd Conf]

Sets the name of the program state directory. See [statedir], page 11.

10.1.2 The `server` statement

The `server` statement configures how `calloutd` will communicate with the client `mailfromd` server.

```
server {
  id name;
  listen url;
  backlog num;
  max-instances num;
  single-process bool;
  reuseaddr bool;
  default bool;
  callout url;
  acl { ... }
}
```

`server` [Calloutd Conf]

Define a server. Optional label may follow the `server` keyword. The label is ignored.

The substatements in the `server` block provide parameters for configuring this server.

`id` *name* [server]
: Assign an identifier to this server. This identifier is used as a suffix to syslog tag (see [syslog tag], page 44) in messages related to this server. For example, if a server block had the following statement in it:

```
id main;
```

then all messages related to this server will be marked with tag '`calloutd#main`'.

The part before the '`#`' is set using the `tag` statement in `logging` block (see Section "Logging Statement" in *GNU Mailutils Manual*).

`listen` *url* [server]
: Listen for connections on the given URL. See [milter port specification], page 220, for a description of allowed *url* formats.

Example:

```
listen inet://10.10.10.1:3331;
```

`backlog` *num* [server]
: Configures the size of the queue of pending connections. Default value is 8.

`max-instances` *number* [server]
: Sets the maximum number of instances allowed for this server.

`single-process` *bool* [server]
: When set to '`yes`', this server will run in *single-process* mode, i.e. it will not fork subprocesses to serve requests. This option is meant exclusively to assist in debugging `calloutd`. Don't use it for anything else but for debugging!

`reuseaddr` *bool* [server]
: When set to '`yes`', `calloutd` will attempt to reuse existing socket addresses. This is the default behavior.

`acl` *statements* [server]
: Defines access control list for this server. See Section "ACL Statement" in *GNU Mailutils Manual*, for a detailed discussion.

If the global ACL is defined as well, an incoming connection is checked against both lists: first the per-server ACL, then the global one. The connection will be permitted only if it passes both checks.

10.1.3 `calloutd` logging

`logger` *mech* [Calloutd Conf]
: Set default logger mechanism. Allowed values for *mech* are:

stderr
: Log everything to the standard error.

syslog
: Log to syslog.

syslog:async
: Log to syslog using the asynchronous syslog implementation.

See Section 3.19 [Logging and Debugging], page 43, for a detailed discussion. See also [syslog-async], page 11, for information on how to set default syslog implementation at compile time.

`debug` *spec* [Calloutd Conf]
: Set `mailfromd` debug verbosity level. The *spec* must be a valid debugging level specification (see [debugging level specification], page 45).

`transcript` *bool* [Calloutd Conf]
: If the boolean value *bool* is '`true`', enables the transcript of call-out SMTP sessions.

10.2 Calloutd Command-Line Options

The `calloutd` invocation syntax is:

```
calloutd [option...]
```

The following options are available:

Server configuration modifiers

`--foreground`
: Stay in foreground. When given this option, `calloutd` will not disconnect itself from the controlling terminal and will run in the foreground.

`-g` *name*
`--group=`*name*
: Retain the group *name* when switching to user privileges. See Section 8.2 [Starting and Stopping], page 239.

`--pidfile=`*file*
: Set pidfile name. Overrides the `pidfile` configuration statement, which you are advised to use instead (see Section 7.2 [conf-base], page 220).

`--resolv-conf-file=`*file*
: Read resolver settings from *file*, instead of the default `/etc/resolv.conf`.

`-S` *ip*
`--source-ip=`*ip*
: Set source address for TCP connections. Overrides the '`source-ip`' configuration statement, which you are advised to use instead (see Section 7.2 [conf-base], page 220).

`--single-process`
: Do not fork sub-processes to serve requests. This option is meant to assist in debugging `calloutd`. Don't use it for anything else but for debugging, as it terribly degrades performance!

`--state-directory=`*dir*
: Set new program state directory. See [statedir], page 11, for the description of this directory and its purposes.

`-u` *name*
`--user` *name*
: Switch to this user's privileges after startup. Overrides the `user` configuration file statement, which you are advised to use instead (see Section 7.10 [conf-priv], page 229). Default user is '`mail`'.

Logging and debugging options

`-d` *string*
`--debug=`*string*
: Set debugging level. See Section 3.19 [Logging and Debugging], page 43.

`--log-facility=`*facility*
: Output logs to syslog *facility*.

`--log-tag=`*string*
: Tag syslog entries with the given *string*, instead of the program name.

`--logger=`*mech*
: Set logger mechanism (*mech* is one of '`stderr`', '`syslog`', '`syslog:async`'). See Section 3.19 [Logging and Debugging], page 43.

`--syslog` Selects default syslog mechanism for diagnostic output.

`--stderr` Directs all logging to standard output. Similar to `--logger=stderr`.

`-S` *ip*
`--source-ip=`*ip*
: Set source address for TCP connections. Overrides the '`source-ip`' configuration statement, which you are advised to use instead (see Section 7.2 [conf-base], page 220).

`--debug-level=`*level*
: Set Mailutils debugging level. See `http://mailutils.org/wiki/Debug_level`, for a detailed discussion of *level* argument.

`--source-info`
`--no-source-info`
: Include C source information in debugging messages. This is similar to setting `line-info yes` in the `debug` configuration block (see Section "debug statement" in *GNU Mailutils Manual*).

  The `--no-source-info` can be used to cancel the effect of the `line-info yes` configuration statement.

  You do not need this option, unless you are developing or debugging `calloutd`.

`-X`
`--transcript`
`--no-transcript`
: Enable or disable transcript of the SMTP sessions to the log channel. See Section 3.19 [Logging and Debugging], page 43.

Configuration file control

`--config-file=`*file*
: Load this configuration file.

`--config-lint`
: Check syntax of configuration files and exit. Exit code is 0 if the file or files are OK, and 78 otherwise.

`--config-verbose`
: Verbosely log parsing of the configuration files.

`--no-site-config`
`--no-config`
: Don't load site-wide configuration file.

`--set=`*`param=value`*
: Set configuration parameter

Informational options

`--config-help`
: Show configuration file summary.

`--show-config-options`
: Show compilation options.

`-?`
`--help`
: Give a short help list.

`--usage`
: Give a short usage message.

`-V`
`--version`
: Print program version

10.3 The Callout Protocol

This section describes the protocol used to communicate with the `calloutd` server.

The protocol works over stream-oriented TCP/IP transport. Either UNIX or IPv4 socket can be used. Commands and responses are terminated by a single CR LF pair. Each command occupies exactly one line. If the server succeeded in executing the command, it replies with a line starting with the word '`OK`'. Depending on the command, this keyword may be followed by a single space character and additional information. More information can be returned in *unsolicited replies* before the '`OK`' line. Each unsolicited reply line starts with and asterisk followed by a single horizontal space character.

On error, the server replies with '`NO`' followed by a horizontal space character and human-readable description of the problem.

The valid commands are discussed below. In examples illustrating the commands, the lines sent by the client are prefixed with *`C:`*, and lines sent by the server are prefixed with *`S:`*.

`vrfy` *`email`* [*`parameter=value`*] [Command]
: Adds *email* to the queue of email addresses to be verified. Available *parameter*s are:

  `mode` [callout parameter]
  : Sets verification mode for this email address. Available modes are:

    `mxfirst`
    `default`
    : The default mode.

If the `host` option is also given, its argument is taken as the domain name. Otherwise, domain part of *email* is used.

The verification goes as follows. First, determine MX servers for that domain. Query each of them in order of increasing priority. First of them that replies determines the result of the test.

If no MX servers are defined for that domain, look for its '`A`' record. If available, run SMTP probe on that IP.

`mxonly` Query MX servers for the domain specified with the `host` option.

`hostonly` Query the server whose name or IP address is supplied with the `host` option.

`hostfirst`
The reverse of `mxfirst`: first query the host, then the MX servers. The domain must be specified using the `host` option.

`host` [callout parameter]
Supplies the domain name for `mode=mxonly`, and host name or IP address for `mode=hostfirst` and `mode=hostonly`. The use of this keyword with any of these modes is mandatory.

`ehlo` [callout parameter]
Supplies the value to use as the argument to the SMTP `EHLO` command.

`mailfrom` [callout parameter]
Supplies the value to use in the SMTP `MAIL FROM` command.

`starttls` [callout parameter]
Controls whether to use SMTP `STARTTLS` command if the server offers it. Allowed values are:

`never` Never use `STARTTLS`.

`always` Always use `STARTTLS`.

`ondemand` Use `STARTTLS` only if `MAIL FROM:` command fails with code 530 (authorization failed). This is the default.

On success, the server replies with '`OK`', followed by a non-negative session ID for that email:

```
C: VRFY gray@example.org
S: OK 0000000001
```

`get` *arg* [*arg*] [Command]
Query value of internal callout parameters. Valid values for *arg* are:

ehlo Return the string used as argument to the SMTP `EHLO` command.

mailfromd Return the email address that is used in the SMTP `MAIL FROM` command.

On success, the server returns the requested value (if found) in an unsolicited reply:

```
C: GET ehlo timeout
S: * ehlo=example.net
S: OK
```

`sid` *string* [Command]

Sets *string* as session identifier for that session. Example:

```
C: SID deadbeef
S: OK
```

`timeout` *connect initial helo mail rcpt rset quit* [Command]

Sets timeouts for various stages of SMTP session. On success, '`OK`' is returned.

```
C: timeout 300 300 300 600 300 300 120
S: OK timeouts set
```

`run` [Command]

Runs callout session for emails registered with the `vrfy` command. On success, results of the check are returned after the '`OK`' keyword in a whitespace-separated list of '*id*`=`*result*' pairs. In each pair, *id* is its identifier as returned in the reply to the `VRFY` command and *result* is one of the following result strings: '`success`', '`not_found`', '`failure`', '`temp_failure`', '`timeout`'.

Additional information about each callout session is returned in unsolicited replies. Each such reply is prefixed with the email identifier and callout stage name. Stage names are:

`INIT` *remote_name*
: The `calloutd` server is establishing communication with the remote SMTP server *remote_name*.

`GRTNG` *line*
: `calloutd` received initial response from the remote server. *line* is the first line of the reply.

`HELO` *line*
: `calloutd` received response to the `EHLO` (or `HELO`) command. In case of multiline response, *line* is the first line.

`SENT` *command*
: The SMTP command *command* has been sent to the remote server.

`RECV` *line*
: The remote server returned *line* in response. In case of multiline response, *line* is the first line.

Example of verification session:

```
C: RUN
S: * 0000000000 INIT mx.example.org
S: * 0000000000 GRTNG 220 mx.example.org ESMTP Ready
S: * 0000000000 HELO 250-mx.example.org Hello tester
S: * 0000000000 SENT RCPT TO:<gray@example.org>
S: * 0000000000 RECV 250 Accepted
S: * 0000000001 INIT foo.example.net
S: * 0000000001 GRTNG 220 foo ESMTP server ready
S: * 0000000001 HELO 250-foo.example.net Hello
S: * 0000000001 SENT RCPT TO:<gray@example.net>
S: * 0000000001 RECV 450 4.7.0 You are greylisted for 3600 seconds
OK 0000000000=success 0000000001=temp_failure
```

`drop` *`serial`* [Command]

Drop the email with the given *serial* number from the verification queue. Example:

```
C: DROP 0000000002
S: OK
```

`quit` [Command]

Finishes the current session and disconnects from the callout server.

```
C: QUIT
S: OK bye
```

11 mfdbtool

The `mfdbtool` utility manages `mailfromd` databases.

11.1 Invoking `mfdbtool`

The following options request the operation to be performed on the database. Exactly one of them must be specified. Each of them implied the `--stderr` option (see [–stderr], page 239).

`--list` List the database. By default, '`cache`' database is assumed. To list another database, use `--format` option (see [–format], page 260).

See Section 3.15.2 [Basic Database Operations], page 32.

`--delete` Delete given entries from the database (see [deleting from databases], page 33). By default, '`cache`' database is assumed. To specify another database, use `--format` option (see [–format], page 260).

See Section 3.15.2 [Basic Database Operations], page 32.

`--expire` Delete all expired entries from the database (see Section 3.15.3 [Database Maintenance], page 33). By default, '`cache`' database is assumed. To specify another database, use `--format` option (see [–format], page 260). Full database name can be given in the command line (see `--file` option below), if it differs from the one specified in the script file.

Use with the option `--all` (see [–all], page 259) to expire all databases.

See Section 3.15.3 [Database Maintenance], page 33.

`--compact`

Compact database (see [compaction], page 33). By default, '`cache`' database is compacted. To specify another database, use `--format` option (see [–format], page 260). Full database name can be given in the command line (see `--file` option below), if it differs from the one specified in the script file.

Use with the option `--all` (see [–all], page 259) to compact all databases.

See [compaction], page 33.

The following options modify the behavior of `mfdbtool`:

`--all` When used with `--compact` or `--expire` option, applies the action to all available databases. See [compact cronjob], page 33.

`-d` *string*
`--debug=`*string*

Sets debugging level. The *string* argument must be a valid mailfromd debug level specification, as described in [debugging level specification], page 45.

`-e` *interval*
`--expire-interval=`*interval*

Set expiration intervals for all databases to the specified interval. See [time interval specification], page 220, for a description of *interval* format. The option overrides the `expire-interval` configuration statement (see [expire-interval-conf], page 230), which you are advised to use instead.

`-f` *filename*
`--file=`*filename*
: Set the name of the database to operate upon (for `--compact`, `--delete`, `--expire`, and `--list` options). Useful if, for some reason, you need to operate on a database whose file name does not match the one `mfdbtool` is configured to use.

`-H` *dbformat*
`--format=`*dbformat*
: Use database of the given format, instead of the default '`cache`'. See Section 3.15.2 [Basic Database Operations], page 32.

`--ignore-failed-reads`
: Ignore records that cannot be retrieved while compacting the database. Without this option, `mfdbtool` will abort the compaction if any such error is encountered.

`--predict=`*rate-limit*
: Used with `--list` enables printing of the estimated times of sending along with the '`rate`' database dump. Implies `--list --format=rate`. See [estimated time of sending], page 32.

`--state-directory=`*dir*
: Sets program state directory. See [statedir], page 11, for the description of this directory and its purposes. This option overrides the `state-directory` configuration statement, described in Section 7.2 [conf-base], page 220.

`--time-format=`*format*
: Set format to be used for timestamps in listings, produced by `--list`. The *format* is any valid `strftime` format string, see Appendix B [Time and Date Formats], page 283, for a detailed description. The default *format* is '`%c`' (see [%c time format], page 283). To analyze *`mfdbtool --list`* output using text tools, such as `awk` or `grep`, the following format might be useful: '`%s`' (see [%s time format], page 284). Another format I find useful is '`%Y-%m-%d_%H:%M:%S`'.

11.2 Configuring `mfdbtool`

Configuration settings are read from the `/etc/mailfromd.conf` file (see Chapter 7 [Mailfromd Configuration], page 219). The following statements are understood:

Statement	**Reference**
`database`	See Section 7.11 [conf-database], page 229.
`database-mode`	See Section 7.11 [conf-database], page 229.
`database-type`	See Section 7.11 [conf-database], page 229.
`debug`	See Section 10.1.3 [conf-calloutd-log], page 251.
`lock-retry-count`	See Section 7.11 [conf-database], page 229.
`lock-retry-timeout`	See Section 7.11 [conf-database], page 229.
`state-directory`	See Section 10.1.1 [conf-calloutd-setup], page 250.

12 `mtasim` — a testing tool

The `mtasim` utility is a MTA simulator for testing `mailfromd` filter scripts. By default it operates in *stdio* mode, similar to that of `sendmail -bs`. In this mode it reads SMTP commands from standard input and sends its responses to the standard output. There is also another mode, called *daemon*, where `mtasim` opens a TCP socket and listens on it much like any MTA does. In both modes no actual delivery is performed, the tool only simulates the actions an MTA would do and responses it would give.

This tool is derived from the program `mta`, which I wrote for GNU Anubis test suite.

12.1 `mtasim` interactive mode mode

If you start `mtasim` without options, you will see the following:

```
220 mtasim (mailfromd 9.2) ready
(mtasim) _
```

The first line is an usual RFC 2821 reply. The second one is a prompt, indicating that `mtasim` is in interactive mode and ready for input. The prompt appears only if the package is compiled with GNU Readline and `mtasim` determines that its standard input is connected to the terminal. This is called *interactive mode* and is intended to save the human user some typing by offering line editing and history facilities (see Section "Command Line Editing" in *GNU Readline Library*). If the package is compiled without GNU Readline, you will see:

```
220 mtasim (mailfromd 9.2) ready
_
```

where '`_`' represents the cursor. Whatever the mode, `mtasim` will wait for further input.

The input is expected to consist of valid SMTP commands and special `mtasim` statements. The utility will act exactly like a RFC 2821-compliant MTA, except that it will not do actual message delivery or relaying. Try typing `HELP` to get the list of supported commands. You will see something similar to:

```
250-mtasim (mailfromd 9.2); supported SMTP commands:
250- EHLO
250- HELO
250- MAIL
250- RCPT
250- DATA
250- HELP
250- QUIT
250- HELP
250  RSET
```

You can try a simple SMTP session now:

```
220 mtasim (mailfromd 9.2) ready
(mtasim) ehlo localhost
250-pleased to meet you
250 HELP
(mtasim) mail from: <me@localhost>
250 Sender OK
```

```
(mtasim) rcpt to: <him@domain>
250 Recipient OK
(mtasim) data
354 Enter mail, end with `.' on a line by itself
(mtasim) .
250 Mail accepted for delivery
(mtasim) quit
221 Done
```

Notice, that `mtasim` does no domain checking, so such thing as '`rcpt to: <him@domain>`' was eaten without complaints.

So far so good, but what all this has to do with `mailfromd`? Well, that's what we are going to explain. To make `mtasim` consult any milter, use `--port` (`-X`) command line option. This option takes a single argument that specifies the milter port to use. The port can be given either in the usual Milter format (See [milter port specification], page 220, for a short description), or as a full `sendmail.cf` style `X` command, in which case it allows to set timeouts as well:

```
$ mtasim --port=inet:999@localhost
# This is also valid:
$ mtasim --port='mailfrom, S=inet:999@localhost, F=T, T=C:100m;R:180s'
```

If the milter is actually listening on this port, `mtasim` will connect to it and you will get the following initial prompt:

```
220-mtasim (mailfromd 9.2) ready
220 Connected to milter inet://localhost:999
(mtasim)
```

Notice, that it makes no difference what implementation is listening on that port, it may well be some other filter, not necessarily `mailfromd`.

However, let's return to `mailfromd`. If you do not want to connect to an existing `mailfromd` instance, but prefer instead to create a new one and run your tests with it (a preferred way, if you already have a stable filter running but wish to test a new script without disturbing it), use `--port=auto`. This option instructs `mtasim` to do the following:

1. Create a unique temporary directory in `/tmp`.
2. Start a new instance of `mailfromd`. This instance is configured to communicate over a UNIX socket in that temporary directory. Additional arguments and options for the new instance may be given in the command line after a double-dash marker ('`--`').
3. Connect to that filter.

When `mtasim` exits, it terminates the subsidiary `mailfromd` process and removes the temporary directory it has created. For example, the following command will start `mailfromd -I. -I../mflib test.rc`:

```
$ mtasim -Xauto -- -I. -I../mflib test.rc
220-mtasim (mailfromd 9.2) ready
220 Connected to milter unix:/tmp/mtasim-j6tRLC/socket
(mtasim)
```

The /tmp/mtasim-j6tRLC directory and any files within it will exist as long as mtasim is running and will be removed when you exit from it.[1] You can also instruct the subsidiary mailfromd to use this directory as its state directory (see [statedir], page 11). This is done by --statedir command line option:

```
$ mtasim -Xauto --statedir -- -I. -I../mflib test.rc
```

(notice that --statedir is the mtasim option, therefore it must appear *before* '--')

You can use an existing directory instead of creating a temporary one. To do so, use the -Xdir:*name* option, e.g.:

```
$ mtasim -Xdir:/var/lib/teststate --statedir -- -I. -I../mflib test.rc
```

Special care should be taken when using mtasim from root account, especially if used with -Xauto and --statedir. The mailfromd utility executed by it will switch to privileges of the user given in its configuration (see Section 8.2 [Starting and Stopping], page 239) and will not be able to create data in its state directory, because the latter was created using 'root' as owner. To help in this case, mtasim understands --user and --group command line options, that have the same meaning as for mailfromd.

Now, let's try HELP command again:

```
250-mtasim (mailfromd 9.2); supported SMTP commands:
250- EHLO
250- HELO
250- MAIL
250- RCPT
250- DATA
250- HELP
250- QUIT
250- HELP
250- RSET
250-Supported administrative commands:
250- \Dname=value [name=value...]       Define Sendmail macros
250- \Ecode [regex]                     Expect given SMTP reply code
250- \L[name] [name...]                 List macros
250- \Uname [name...]                   Undefine Sendmail macros
250  \Sfamily hostname address [port]   Define sender socket address
```

While the SMTP commands do not need any clarification, some words about the *administrative commands* are surely in place. These commands allow to define, undefine and list arbitrary Sendmail macros. Each administrative command consists of a backslash followed by a command letter. Just like SMTP ones, administrative commands are case-insensitive. If a command takes arguments, the first argument must follow the command letter without intervening whitespace. Subsequent arguments can be delimited by arbitrary amount of whitespace.

For example, the \D command defines Sendmail macros:

```
(mtasim) \Dclient_addr=192.168.10.1 f=sergiusz@localhost i=testmsg
(mtasim)
```

[1] However, this is true only if the program is exited the usual way (via QUIT or end-of-file). If it is aborted with a signal like SIGINTR, the temporary directory is not removed.

Notice that `mailfromd` does not send any response to the command, except if there was some syntactic error, in which case it will return a '`502`' response.

Now, you can list all available macros:

```
(mtasim) \L
220-client_addr=192.168.10.1
220-f=sergiusz@localhost
220 i=testmsg
(mtasim)
```

or just some of them:

```
(mtasim) \Lclient_addr
220 client_addr=192.168.10.1
(mtasim)
```

To undefine a macro, use `\U` command:

```
(mtasim) \Ui
(mtasim) \l
220-client_addr=192.168.10.1
220 f=sergiusz@localhost
(mtasim)
```

The `\S` command declare sender socket and host name. These parameters are passed to the `connect` handler, if one is declared (see [connect handler], page 70). To give you a chance to use this command, `mtasim` does not invoke `connect` handler right after connecting to the milter. Instead it waits until either the `\S` command or any SMTP command (except '`HELP`') is given. After calling `connect` handler the `\S` is disabled (to reflect it, it also disappears from the `HELP` output).

The `\S` command takes 1 to 4 arguments. The first argument supplies the socket family (see Table 4.3). Allowed values are: '`stdio`', '`unix`', '`inet`', '`inet6`' or numbers from '`0`' to '`3`'.

The `\S stdio` (or `\S 0`) command needs no additional arguments. It indicates that the SMTP connection is obtained from the standard input. It is the default if sender socket is not declared explicitly.

The command `\S unix` indicates that the connection is accepted from a UNIX socket. It requires two more argument. The first one supplies sender host name and the second one supplies full path name of the socket file. For example:

```
\S unix localhost /var/run/smtp.sock
```

The commands `\S inet` and `\S inet6` indicate that the connection came from an '`INET`' IPv4 or IPv6 socket, correspondingly[2]. They require all four arguments to be specified. The additional arguments are: host name, IP address, and port number, in that order. For example:

```
\S inet relay.gnu.org.ua 213.130.31.41 34567
```

or

```
\S inet6 relay.gnu.org.ua 2001:470:1f0a:1be1::2 34567
```

[2] Depending on how `mailfromd` is configured, '`inet6`' may be not available.

Sender socket address can also be configured from the command line (see Section 12.6 [option summary], page 268).

Now, let's try a real-life example. Suppose you wish to test the greylisting functionality of the filter script described in Section 4.27 [Filter Script Example], page 120. To do this, you start `mtasim`:

```
$ mtasim -Xauto -- -I. -I../mflib test.rc
220-mtasim (mailfromd 9.2) ready
220 Connected to milter unix:/tmp/mtasim-ak3DEc/socket
(mtasim)
```

The script in `test.rc` needs to know `client_addr` macro, so you supply it to `mtasim`:

```
(mtasim) \Dclient_addr=10.10.1.13
```

Now, you try an SMTP session:

```
(mtasim) ehlo yahoo.com
250-pleased to meet you
250 HELP
(mtasim) mail from: <gray@yahoo.com>
250 Sender OK
(mtasim) rcpt to: <gray@localhost>
450 4.7.0 You are greylisted for 300 seconds
```

OK, this shows that the greylisting works. Now quit the session:

```
(mtasim) quit
221 Done
```

12.2 `mtasim` expect commands

Until now we were using `mtasim` interactively. However, it is often useful in shell scripts, for example the `mailfromd` test suite is written in shell and `mtasim`. To avoid the necessity to use auxiliary programs like `expect` or `DejaGNU`, `mtasim` contains a built-in expect feature. The administrative command `\E` introduces the SMTP code that the next command is expected to yield. For example,

```
\E250
rcpt to: <foo@bar.org>
```

tells `mtasim` that the response to `RCPT TO` command must begin with '`250`' code. If it does, `mtasim` continues execution. Otherwise, it prints an error message and terminates with exit code 1. The error message it prints looks like:

```
Expected 250 but got 470
```

The expected code given with the `\E` command may have less than 3 digits. In this case it specifies the first digits of expected reply. For example, the command '`\E2`' matches replies '`200`', '`220`', etc.

If `\E` is passed two arguments, the second one is treated as an extended regular expression. The subsequent command will then succeed if its return code matches the one supplied as the first argument, and its extended SMTP code and textual message match the supplied regular expression. If the regular expression contains whitespace, enclose it in a pair of double quotes. Within double quotes, backslash can be used to escape double quote, and backslash character.

This feature can be used to automate your tests. For example, the following script tests the greylisting functionality (see the previous section):

```
# Test the greylisting functionality
#
\E220
\Dclient_addr=10.10.1.13
\E250
ehlo yahoo.com
\E250
mail from: <gray@yahoo.com>
\E450
rcpt to: <gray@localhost>
\E221
quit
```

This example also illustrates the fact that you can use '`#`'-style comments in the `mtasim` input.

`mtasim` scripts can be used in shell programs, for example:

```
mtasim -Xauto --statedir -- -P../mflib test.rc < scriptfile
if $? -ne 0; then
  echo "Greylisting test failed"
fi
```

12.3 Trace Files

It is possible to log an entire SMTP session to a file. This is called *session tracing*. Two options are provided for this purpose:

`--trace-file=`*file*
: Sets the name of the trace file, i.e. a file to which the session transcript will be written. Both the input commands, and the `mtasim` responses are logged. If the file *file* exists, it will be truncated before logging. This, however, can be changed using the following option:

`-a`
`--append` If the trace file exists, append new trace data to it.

12.4 Daemon Mode

To start `mtasim` in *daemon* mode, use the `--daemon` (or `-bd`) command line option. This mode is not quite the same as Sendmail `-bd` mode. When started in *daemon* mode, `mtasim` selects the first available TCP port to use from the range '`1024 -- 65535`'. It prints the selected port number on the standard output and starts listening on it. When a connection comes, it serves a *single* SMTP session and exits immediately when it is ended.

This mode is designed for use in shell scripts and automated test cases.

12.5 Summary of the `mtasim` Administrative Commands

This section provides a summary of administrative commands available in `mtasim`.

`\D` *name=value* [*name=value...*] [mtasim command]
Defines Sendmail macro *name* to the given *value*. Any number of *name=value* pairs can be given as arguments.

See [D command], page 263.

`\E` *code* [*regex*] [mtasim command]
Instructs `mtasim` to expect next SMTP command to return given *code* (a three-digit decimal number). Optional *regex* argument is an extended POSIX regular expression. If supplied, the program will also require that the extended SMTP code (if any) and textual message returned by the command match this expression.

See Section 12.2 [expect commands], page 265.

`\L` [*name...*] [mtasim command]
Lists defined macros. See [L command], page 264.

`\U` *name* [*name...*] [mtasim command]
Undefines macros given as its arguments.

`\S` *family* [*hostname address* [*port*]] [mtasim command]
Declares the sender socket parameters. See [S command], page 264, for a detailed description and examples.

This command is available only at the initial stage of a `mtasim` session, before the first SMTP command was given. It is disabled if the `--sender-socket` option was given in the command line (see Section 12.6 [option summary], page 268). The `help` output reflects whether or not this command is available.

If neither this command nor the `--sender-socket` option were given, `mtasim` behaves as if given the `\S stdio` command.

The *family* argument supplies the socket family, i.e. the first argument to the `connect` handler (see [connect handler], page 70). It can have either literal or numeric value, as described in the table below:

Literal	**Numeric**	**Meaning**
stdio	0	Standard input/output (the MTA is run with `-bs` option)
unix	1	UNIX socket
inet	2	IPv4 protocol
inet6	3	IPv6 protocol

Table 12.1: Socket families

See also Table 4.3.

Depending on the *family*, the rest of arguments supply additional parameters:

stdio
: The *hostname* argument can be specified. It defines the first argument of the `connect` handler (see [hostname in connect handler], page 70).

inet
inet6
: All arguments must be specified.

argument	connect argument	meaning
hostname	1	Sender host name
address	4	Sender IP address
port	3	Sender port number

unix *Hostname* and *address* must be supplied. The *address* argument must be a full pathname of the UNIX socket.

12.6 `mtasim` command line options

This section summarizes all available `mtasim` command line options.

`--append`
`-a` Append to the trace file. See Section 12.3 [traces], page 266.

`--body-chunk=`*`number`*
Set the body chunk length (bytes) for `xxfi_body` calls.

`--daemon`
`-bd` Run as daemon. See Section 12.4 [daemon mode], page 266.

`--define=`*`macro`*`=`*`value`*
`-D` *`macro`*`=`*`value`*
Define Sendmail macro *macro* to the given *value*. It is similar to the `\D` administrative command (see [D command], page 263)

`--gacopyz-log=`*`level`*
Set desired logging level for `gacopyz` library (see Appendix A [Gacopyz], page 281). See [gacopyz-log option], page 237, for a detailed description of *level*. Notice, that unless this option is used, the `--verbose` (`-v`) command line option implies `--gacopyz-log=debug`.

`--group=`*`name`*
`-g` *`name`* When switching to user's privileges as requested by the `--user` command line option, retain the additional group *name*. Any number of `--group` options may be given to supply a list of additional groups.

`--user=`*`name`*
`-u` *`name`* Run with this user privileges. This option and the `--group` option have effect only if `mtasim` was started with root privileges.

`--help`
`-?` Display a short help summary

`--milter-version=`*`version`*
Force using the given Milter protocol version number. The *version* argument is either a numeric version (e.g. '`2`'), or a version string in form '*`major.minor`*`[.patch]`', where square brackets indicate optional part. The default is '`1.0.0`'. If *version* is any of '`2`', '`3`' or '`1.0.0`', the default protocol capabilities and actions for that version are set automatically. This option is intended for development and testing of the Gacopyz library (see Appendix A [Gacopyz], page 281).

`--milter-proto=`*bitmask*
: Set Milter protocol capabilities. See `gacopyz/gacopyz.h` for the meaning of various bits in the *bitmask*. Look for the C macros with the prefix '`SMFIP_`'.

`--milter-timeout=`*values*
: Set timeouts for various Milter operations. *Values* is a comma-separated list of assignments '*T*`=`*V*', where *T* is a *timeout code*, indicating which timeout to set, and *V* is its new value. Valid timeout codes are:

  C
  : Timeout for connecting to a filter.

  W
  S
  : Timeout for sending information from the simulator to a filter.

  R
  : Timeout for reading reply from the filter.

  E
  : Overall timeout between sending end-of-message to filter and receiving final acknowledgment. Indirectly, it configures the upper limit on the execution time of the `eom` handler (see [eom handler], page 74).

`--milter-actions=`*bitmask*
: Set Milter actions. See `gacopyz/gacopyz.h` for the meaning of various bits in the *bitmask*. Look for the C macros with the prefix '`SMFIF_`'.

`--no-interactive`
: Not-interactive mode (disable readline). See Section "Command Line Editing" in *GNU Readline Library*.

`--port=`*port*
`-X` *port*
: Communicate with given Milter *port*. Valid values for *port* are:

  port specification
  : See [milter port specification], page 220, for a detailed discussion. Example:

    ```
    --port inet:999@localhost
    ```

  `sendmail.cf` `X` command format
  : For example:

    ```
    --port='mailfrom, S=inet:999@localhost, F=T, T=C:100m;R:180s'
    ```

  `auto`
  : Create a temporary directory and start an instance of `mailfromd` configured to communicate over a UNIX socket in that directory. If `--statedir` is also given, the created `mailfromd` instance will use that directory as its state directory (see [statedir], page 11). Additional arguments for `mailfromd` may be supplied after the `--` delimiter.

    Before termination, `mtasim` will stop the `mailfromd` instance it created and remove the temporary directory.

  `dir:`*dirname*
  : Same as `auto`, except that instead of the temporary directory the directory *dirname* is used. This directory is not removed when `mtasim` terminates. Example:

```
--port dir:/tmp/state
```

See [mtasim milter port], page 262, for a detailed discussion of the `--port` option and its use.

`--prompt=`*`string`*
: Set readline prompt. The default prompt string is '`(mtasim) `'.

`--sender-socket=`*`family`*`[,`*`hostname`*`,`*`address`*`[,`*`port`*`]]`
: Declare sender socket address. This option has the same effect as the `S command`. See [S command], page 264, for a detailed discussion and a description of its arguments.

`--statedir`
: When using `-Xauto`, use the temporary directory name as `mailfromd` state directory (see [statedir mtasim option], page 262).

`--stdio`
`-bs`
: Use the SMTP protocol on standard input and output. This is the default mode for `mtasim`. See Section 12.1 [interactive mode], page 261.

`--trace-file=`*`file`*
: Set name of the trace file. See Section 12.3 [traces], page 266.

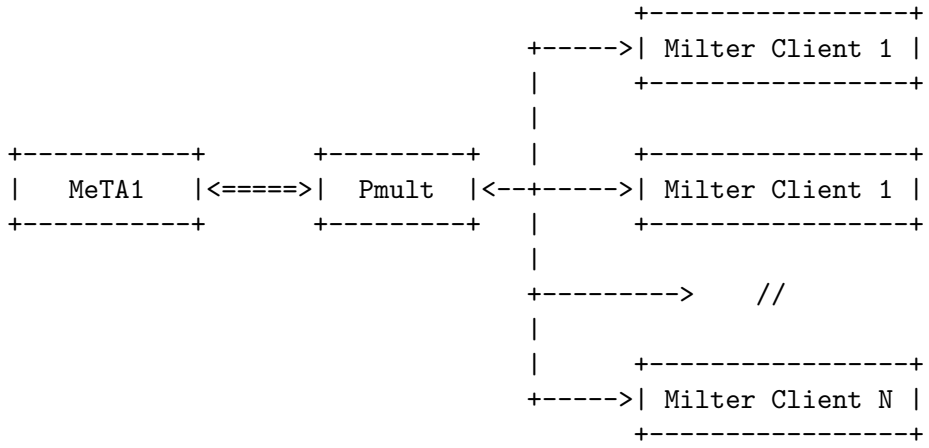
`--usage`
: Display option summary

`--verbose`
`-v`
: Increase verbosity level. Implies `--gacopyz-log=debug`, unless that option is used explicitly.

`--version`
`-V`
: Print program version

13 Pmilter multiplexer program.

`Pmult` is a *Pmilter–Milter multiplexer*, i.e. a program that acts as a mediator between the *Pmilter* server and one or several *Milter* clients. Usually, the former is an instance of `smtps` from MeTA1, and the latter are running `mailfromd` instances. `Pmult` receives Pmilter commands from the server, translates them into equivalent Milter commands and passes the translated requests to a preconfigured set of Milter filters. When the filters reply, the reverse operation is performed: Milter responses are translated into their Pmilter equivalents and are sent back to the server.



Due to the specifics nature of the threaded MeTA1 libraries, `pmult` does not detach from the controlling terminal (i.e. does not become a daemon). To run it as a background process, we recommend to use `pies` daemon. '`Pies`' is a powerful utility that allows you to launch several foreground-designed programs in the background and control their execution. See Section "Pies" in *Pies Manual*, for a detailed description of the program. For a practical advice on how to use it with `pmult`, see Section "Simple Pies" in *Pies Manual*. For a description on how to start both `pmult` and MeTA1 from the same `pies` configuration file, see Section "Hairy Pies" in *Pies Manual*.

13.1 Pmult Configuration

`Pmult` reads its configuration from the main configuration file `/etc/mailfromd.conf`. Unless it is the only component of the '`Mailfromd`' package that is being run, its configuration should be explicitly marked as such by using either `program` or `include` statement, as described in Chapter 7 [Mailfromd Configuration], page 219.

The following standard Mailutils statements are understood:

Statement	**Reference**
debug	See Section "debug statement" in *GNU Mailutils Manual*.
logging	See Section "logging statement" in *GNU Mailutils Manual*.
include	See Section "include" in *GNU Mailutils Manual*.

13.1.1 Multiplexer Configuration.

`Pmult` listens for Pmilter requests on a socket, configured using `listen` statement:

`listen` *`url`* [Pmult Conf]
: Listen on the given *url*. Argument is a valid Mailutils URL. See [milter port specification], page 220, for a description of *url*.

Since `pmult` runs as a foreground program, it does not write its PID number to a file by default. If this behavior is required, it can be enabled using the following statement:

`pidfile` *`file`* [Pmult Conf]
: Store PID of the `pmult` process in *file*.

The following three limits require MeTA1 version '`PreAlpha30.0`' or later.

`max-threads-soft` *`n`* [Pmult Conf]
: "Soft" limit on the number of '`pmilter`' threads. Default is 2.

`max-threads-hard` *`n`* [Pmult Conf]
: "Hard" limit on the number of '`pmilter`' threads. This is roughly equivalent to the number of emails `pmult` is able to handle simultaneously. The default value is 6. Raise this limit if you experience long delays when connecting to the SMTP port.

`max-pmilter-fd` *`n`* [Pmult Conf]
: Maximum number of file descriptors '`pmilter`' library is allowed to open simultaneously. Default is 10.

13.1.2 Translating MeTA1 macros.

MeTA1's notion of macros differs considerably from that of Sendmail. Macros in MeTA1 are identified by integer numbers and only a limited number of macros can be provided for each *Pmilter stage*. Pmilter stages mostly correspond to Milter states (see [handler names], page 13), except that there are no distinct header and body stages, instead these two are combined into a single '`data`' stage. This comes unnoticed to mailfromd scripts, because `pmult` takes care to invoke right Milter handlers within the single '`data`' Pmilter state. Therefore in the discussion that follows we will refer to Mailfromd handlers, rather than to Pmilter stages.

The most important standard Milter macros are always provided by `pmult` itself. These are:

client_addr
: The IP address of the SMTP client. As of version 9.2, only IPv4 addresses are supported. Defined in all handlers.

client_port
: The port number of the SMTP client. Defined in all handlers.

i
: MeTA1 session ID. Defined in all handlers.

f
: The envelope sender (from) address. Defined in `envfrom` and subsequent handlers.

nbadrcpts The number of bad recipients for a single message. Defined in `envfrom` and `envrcpt` handlers.

ntries The number of delivery attempts. As of version 9.2 it is always '`1`'. Defined in `envfrom` and subsequent handlers.

nrcpts The number of validated recipients for a single message. Defined in `envfrom` and `envrcpt` handlers.

r Protocol used to receive the message. The value of this macro is always '`SMTP`'. Defined in all handlers.

rcpt_host The host from the resolved triple of the address given for the SMTP RCPT command. Defined in `envrcpt` handler.

rcpt_addr The address part of the resolved triple of the address given for the SMTP RCPT command. Defined in `envrcpt` handler.

s Sender's helo domain (parameter to `EHLO` or `HELO` command).

Two additional macros are provided for all handlers that allow to identify whether the message is processed via `pmult`:

multiplexer
Canonical name of the multiplexer program, i.e. '`pmult`'.

mult_version
Version of `pmult`.

These macros can be used in mailfromd filters to provide alternative processing for messages coming from a MeTA1 server.

Macros defined in MeTA1 can be made available in Mailfromd handlers using the `define-macros` statement.

`define-macros` *handler macros* [Pmult Conf]
Define a set of Sendmail macros for the given Mailfromd handler. Allowed values for *handler* are: '`connect`', '`helo`', '`mail`' (or '`envfrom`'), '`rcpt`' (or '`envrcpt`'), '`data`' (or '`header`' or '`body`'), '`dot`' ('`eom`'). A list of these values is also accepted, in which case *macros* are defined for each handler from the list.

The second argument specifies a list of names of the macros that should be defined in this handler. Allowed macro names are:

hostname Hostname of SMTP server.

client_resolve
Result of client lookup.

tls_version TLS/SSL version used.

tls_cipher_suite
TLS cipher suite used.

tls_cipher_bits
Effective key length of the symmetric encryption algorithm.

tls_cert_subject
: The DN (distinguished name) of the presented certificate.

tls_cert_issuer
: The DN (distinguished name) of the CA (certificate authority) that signed the presented certificate (the cert issuer).

tls_alg_bits
: Maximum key length of the symmetric encryption algorithm. This may be less than the effective key length for export controlled algorithms.

tls_vrfy
: The result of the verification of the presented cert.

tls_cn_subject
cn_subject
: The CN (common name) of the presented certificate.

tls_cn_issuer
cn_issuer
: The CN (common name) of the CA that signed the presented certificate.

auth_type
: The mechanism used for SMTP authentication (only set if successful).

auth_authen
: The client's authentication credentials as determined by authentication (only set if successful). The actual format depends on the mechanism used, it might be just '`user`', or '`user@realm`', or something similar.

auth_author
: The authorization identity, i.e. the '`AUTH=`' parameter of the SMTP MAIL command if supplied.

taid
: MeTA1 transaction id.

msgid
: Message-Id of the message.

c
: The hop count. Basically, this is the number of '`Received:`' headers.

Notice the following limitations:

1. '`taid`' cannot be requested before '`mail`' stage.
2. '`msgid`' can be requested only in '`dot`' stage.
3. All '`tls_*`' macros are valid only after a `STARTTLS` command.
4. The number of MeTA1 macros per stage is limited by `PM_MAX_MACROS` define in `include/sm/pmfdef.h`. In MeTA1 versions up to and including 1.0.PreAlpha28.0, this number is 8. If you need more macros, increase this number and recompile MeTA1.

`auth-macros` *bool* [Pmult Conf]

If *bool* is `true` (see Section "Statements" in *GNU Mailutils Manual*), pass auth macros to mailfromd '`mail`' handler. It is equivalent to:

```
define-macros mail (auth_type, auth_authen, auth_author);
```

13.1.3 Pmult Client Configuration.

In `pmult` terminology, remote Milters are *clients*. The number of clients `pmult` is able to handle is not limited. Each client is declared using `client` statement:

```
client [ident] {
  # Set remote protocol type.
  type protocol-type;
  # Set remote client URL.
  url arg;
  # Set write timeout.
  write-timeout duration;
  # Set read timeout.
  read-timeout duration;
  # Set timeout for EOM.
  eom-timeout duration;
  # Set connect timeout.
  connect-timeout duration;
  # Set log verbosity level.
  log-level level;
};
```

`client` [*ident*] { *statements* } [Pmult Conf]

Declare a Milter client. Optional *ident* gives the identifier of this client, which will be used in diagnostics messages.

Statements are described below.

`type` *typestr* [Pmult Conf]

This statement is reserved for future use. In version 9.2 it is a no-op.

If given, the value of *typestr* must be '`milter`'.

In future versions this statement will declare the protocol to be used to interact with this client. The syntax for *typestr* is

```
type [version]
```

where *type* is either '`milter`' or '`pmilter`', and optional *version* is minimal protocol version.

`url` *arg* [Pmult Conf]

Set remote client URL. See [milter port specification], page 220, for a description of *url*.

`connect-timeout` *interval* [Pmult Conf]

Configure Milter initial connection timeout. Default is 300.

See [time interval specification], page 220, for information on *interval* format.

`write-timeout` *interval* [Pmult Conf]

Configure Milter write timeout. Default is 10.

See [time interval specification], page 220, for information on *interval* format.

`read-timeout` *interval* [Pmult Conf]

Configure Milter read timeout. Default is 10.

See [time interval specification], page 220, for information on *interval* format.

`eom-timeout` *interval* [Pmult Conf]

Configure Milter end of message timeout. Default is 300.

See [time interval specification], page 220, for information on *interval* format.

`log-level` *arg* [Pmult Conf]

Set Milter log verbosity level for this client. Argument is a list of items separated by commas or whitespace. Each item is a log level optionally prefixed with '`!`' to indicate "any level except this", '`<`', meaning "all levels up to and including this", or with '`>`', meaning "all levels starting from this".

Log levels in order of increasing priority are: '`proto`', '`debug`', '`info`', '`warn`', '`err`', '`fatal`'. The first two levels are needed for debugging `libgacopyz` and Milter protocol. See Appendix A [Gacopyz], page 281, for the description of the `libgacopyz` library. See also the following subsection.

13.1.4 Debugging Pmult

If needed, `pmult` can be instructed to provide additional debugging information. The amount of this information is configured by three configuration statements. First of all, the standard `debug` block statement controls debugging of the underlying GNU Mailutils libraries (see Section "Debug Statement" in *GNU Mailutils Manual*). Secondly, the `debug` statement controls debugging output of the `pmult` utility itself. The `pmilter-debug` statement controls debugging output of the underlying MeTA1 libraries, and, finally, the `log-level` statement, described in the previous subsection, defines debugging level for the Milter library (`libgacopyz`).

`debug` *spec* [Pmult Conf]

Set debugging level for the `pmult` code. See Section "Debug Statement" in *GNU Mailutils Manual*, for a description of *spec* syntax. Multiplexor-specific debugging is enabled by the '`pmult`' category. The following levels are used:

pmult.trace1
: Prints the following information:
  - opening and closing incoming connections;
  - entering particular Pmilter stage handlers;
  - received requests with unknown command code;
  - header modification requests that does not match any headers.

pmult.trace2
: Information about milter to Pmilter request translation.

pmult.trace7
: Detailed dump of message body chunks received during Pmilter '`DATA`' stage.

pmult.error
: Logs bad recipient addresses.

This information is printed using the output channel defined in the `logging` statement (see Section "Logging Statement" in *GNU Mailutils Manual*).

`pmilter-debug` *`level`* [Pmult Conf]

Set debug verbosity level for the Pmilter library. Argument is a positive integer between zero (no debugging, the default), and 100 (maximum debugging).

Pmilter debugging information is printed on the standard error. Use `pies stderr` statement to capture this stream and redirect it to the syslog or file (see Section "Output Redirectors" in *Pies Manual*).

13.2 Pmult Example

The following is an example of a working `pmult` configuration. The multiplexer listens on localhost, port '`3333`'. It prints its diagnostics using syslog facility `local2`. A single Mailfromd client is declared, which listens on UNIX socket `/usr/local/var/mailfromd/mailfrom`. The log verbosity level for this client is set to '`info`' and higher, i.e.: '`info`', '`warn`', '`err`' and '`fatal`'.

```
listen inet://127.0.0.1:3333;

logging {
   facility local2;
};

debug {
   level "pmult.trace7";
}

define-macros envmail (auth_type, auth_authen, auth_author, tls_vrfy);
define-macros envrcpt (auth_type, auth_authen, auth_author);

client {
   type milter;
   url /usr/local/var/mailfromd/mailfrom;
   log-level ">info";
   # Set write timeout.
   write-timeout 30 seconds;
   # Set read timeout.
   read-timeout 5 minutes;
   # Set timeout for EOM.
   eom-timeout 5 minutes;
}
```

13.3 Pmult Invocation

Normally, `pmult` is invoked without command line arguments. However, it does support several command line options. First of all, the common GNU Mailutils options are understood, which are useful for checking `pmult` configuration file for syntax errors. See Section "Common Options" in *GNU Mailutils Manual*, for a detailed description of these.

The rest of command line options supported by `pmult` is useful mostly for debugging. These options are summarized in the table below:

`--log-tag=`*`string`*
: Set the identifier used in syslog messages to *string*. This option mostly is for debugging purposes. We advise to use `logging` configuration statement for this purpose (see Section "Logging Statement" in *GNU Mailutils Manual*).

`--no-signal-handler`
: Disable signal handling in the main thread. This is for debugging purposes.

`--syslog`
: Log to the syslog. This is the default. See Section "Logging Statement" in *GNU Mailutils Manual*, for information on how to configure syslog logging.

`-s`
`--stderr`
: Log to the standard error stream.

`--url=`*`url`*
: Listen on the given *url*. This overrides the `url` configuration statement (see Section 13.1.3 [pmult-client], page 275).

`-x`
`--debug=`*`level`*
: Set debug verbosity level. This overrides the `debug` configuration statement. See Section 13.1.4 [pmult-debug], page 276, for more information.

14 How to Report a Bug

> *Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.*
> Dick Brandon

Although the author has tried to make this documentation as detailed as is possible and practical, he is well aware that the result is rather "better than nothing", than "very good". So, if you find that some piece of explanation is lousy or if you find anything that should have been mentioned here, but is not, please report it to `bug-mailfromd@gnu.org.ua`.

Similarly, if the program itself fails to meet your expectations, or does not do what is described in this document; if you have found a bug or happen to have any suggestion... or have written a useful function you wish to share with the rest of `mailfromd` users, or wish to express your thanks, email it to the same address, `bug-mailfromd@gnu.org.ua`.

If you think you've found a bug, please be sure to include maximum information needed to reliably reproduce it, or at least to analyze it. The information needed is:

- Version of the package you are using.
- Compilation options used when configuring the package.
- Run-time configuration (`mailfromd.mfl` file and the command line options used).
- Conditions under which the bug appears.

Appendix A Gacopyz

> *Gacopyz, panie, to mówią ze to mysa... Ze to tako mysa co świeckę w kościele zjadła i wniebowstąpienia dostąpiła. A to nie je mysa, ino gacopyz! To nadprzyrodzłune, to głową na dół śpi!*
>
> Kazimierz Grześkowiak

'`Gacopyz`' is the client library implementing Milter protocol. It differs considerably from the Sendmail implementation and offers a new and more flexible API. The old API is supported for compatibility with `libmilter`.

The library name comes from the song '`Rozprawa o robokach`' by Kazimierz Grzeskowiak (`http://grzeskowiak.art.pl`). The phrase '`A to nie je mysa, ino gacopyz`' exactly describes what the library is: '`That is no libmilter, but gacopyz`'.

Future versions of this documentation will include a detailed description of the library.

Appendix B Time and Date Formats

This appendix documents the time format specifications understood by the command line option `--time-format` (see [–time-format], page 260). Essentially, it is a reproduction of the man page for GNU `strftime` function.

Ordinary characters placed in the format string are reproduced without conversion. Conversion specifiers are introduced by a '`%`' character, and are replaced as follows:

%a — The abbreviated weekday name according to the current locale.

%A — The full weekday name according to the current locale.

%b — The abbreviated month name according to the current locale.

%B — The full month name according to the current locale.

%c — The preferred date and time representation for the current locale.

%C — The century number (year/100) as a 2-digit integer.

%d — The day of the month as a decimal number (range 01 to 31).

%D — Equivalent to '`%m/%d/%y`'.

%e — Like '`%d`', the day of the month as a decimal number, but a leading zero is replaced by a space.

%E — Modifier: use alternative format, see below (see [conversion specs], page 285).

%F — Equivalent to '`%Y-%m-%d`' (the ISO 8601 date format).

%G — The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see '`%V`'). This has the same format and value as '`%y`', except that if the ISO week number belongs to the previous or next year, that year is used instead.

%g — Like '`%G`', but without century, i.e., with a 2-digit year (00-99).

%h — Equivalent to '`%b`'.

%H The hour as a decimal number using a 24-hour clock (range 00 to 23).

%I The hour as a decimal number using a 12-hour clock (range 01 to 12).

%j The day of the year as a decimal number (range 001 to 366).

%k The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also '`%H`'.)

%l The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also '`%I`'.)

%m The month as a decimal number (range 01 to 12).

%M The minute as a decimal number (range 00 to 59).

%n A newline character.

%O Modifier: use alternative format, see below (see [conversion specs], page 285).

%p Either '`AM`' or '`PM`' according to the given time value, or the corresponding strings for the current locale. Noon is treated as '`pm`' and midnight as '`am`'.

%P Like '`%p`' but in lowercase: '`am`' or '`pm`' or a corresponding string for the current locale.

%r The time in '`a.m.`' or '`p.m.`' notation. In the POSIX locale this is equivalent to '`%I:%M:%S %p`'.

%R The time in 24-hour notation ('`%H:%M`'). For a version including the seconds, see '`%T`' below.

%s The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

%S The second as a decimal number (range 00 to 61).

%t A tab character.

%T The time in 24-hour notation ('`%H:%M:%S`').

%u	The day of the week as a decimal, range 1 to 7, Monday being 1. See also '`%w`'.
%U	The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also '`%V`' and '`%W`'.
%V	The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also '`%U`' and '`%W`'.
%w	The day of the week as a decimal, range 0 to 6, Sunday being 0. See also '`%u`'.
%W	The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
%x	The preferred date representation for the current locale without the time.
%X	The preferred time representation for the current locale without the date.
%y	The year as a decimal number without a century (range 00 to 99).
%Y	The year as a decimal number including the century.
%z	The time-zone as hour offset from GMT. Required to emit RFC822-conformant dates (using '`%a, %d %b %Y %H:%M:%S %z`')
%Z	The time zone or name or abbreviation.
%+	The date and time in *date(1)* format.
%%	A literal '`%`' character.

Some conversion specifiers can be modified by preceding them by the '`E`' or '`O`' modifier to indicate that an alternative format should be used. If the alternative format or specification does not exist for the current locale, the behaviour will be as if the unmodified conversion specification were used. The Single Unix Specification mentions '`%Ec`', '`%EC`', '`%Ex`', '`%EX`', '`%Ry`', '`%EY`', '`%Od`', '`%Oe`', '`%OH`', '`%OI`', '`%Om`', '`%OM`', '`%OS`', '`%Ou`', '`%OU`', '`%OV`', '`%Ow`', '`%OW`', '`%Oy`', where the effect of the '`O`' modifier is to use alternative numeric symbols

(say, roman numerals), and that of the '`E`' modifier is to use a locale-dependent alternative representation.

Appendix C Upgrading

The following sections describe procedures for upgrading between the consecutive Mailfromd releases. The absence of a section for a pair of versions *x*-*y* numbers means that no specific actions are required for upgrading from *x* to *y*.

C.1 Upgrading from 9.1 to 9.2

No special actions are required. The following points are worth noticing:

This version introduces sendmail macro tracing that ensures that all macros used in a given milter stage (`prog` handler) will be requested from Sendmail. This includes macros used in functions called from the handler, those given as arguments to `getmacro` or `macro_defined` functions (see Section 5.1 [Macro access], page 125), as well as those used in `prog action` callback (see Section 4.15.3 [action hook], page 77). This eliminates the need for `#pragma miltermacros` in almost all cases, so it can safely be removed. See [macro tracing], page 58, for details.

It was noted that macros used in `prog eom` were often missing. This was due to a bug in Sendmail that made it ignore any macros requested for that stage, unless some were also requested for `prog eoh`. This version installs a workaround that ensures the macros are properly registered.

C.2 Upgrading from 8.17 to 9.0

Old '`.mf`' suffix gone

The '`.mf`' suffix, that has long been used for MFL files and has been deprecated since version 8.15 (see Section C.4 [8140-8150], page 288) is no longer supported. MFL files have suffix '`.mfl`'.

Begin and end handlers must be defined using `prog`

The old syntax was:

```
begin
do
  ...
end
```

It is no longer supported. The correct way to declare `begin` and `end` handlers is

```
prog begin
do
  ...
end
```

Module and include search paths

Since version 8.15 if a module was not found in module search path, the search was retried using include search path. This is no longer the case: the two paths serve different purposes and don't interact in any way. MFL modules are searched in module search path only.

IPv6

Version 9.0 features full IPv6 support.

The following modules used in previous releases are now deprecated. They are retained for backward compatibility. Any existing code that uses any of these will compile and work as in previous releases, except that a warning will be printed to draw your attention to the fact. You are advised to remove uses of these modules, as they will be removed in future versions:

`match_cidr`

The function `match_cidr` is reimplemented as a built-in function. Remove any instances of `require 'match_cidr'` or any equivalent `import` statements from your code.

See [match_cidr], page 165.

`is_ip` This module defines the `is_ip` function:

`boolean is_ip (string str)` [Library Function]
Returns 1 if *str* is a string representation of an IPv4 address and 0 otherwise.

This function has been obsoleted by `is_ip4str` function. You are advised to use it, or a more general `is_ipstr`, function instead. See Section 5.23 [Internet address manipulation functions], page 164, for a detailed discussion of these functions,

`revip` This module defines the following function:

`string revip (string ip)` [Library Function]
Reverses octets in *ip*, which must be a valid string representation of an IPv4 address.

This function has been obsoleted by `reverse_ipstr`, discussed in Section 5.23 [Internet address manipulation functions], page 164.

C.3 Upgrading from 8.16 to 8.17

No special actions are required after upgrading. However, one point is worth noticing. Until now, it was an error to specify two `prog` definitions with the same handler name. This rule however, applied only to generic milter handlers, special handlers (`begin`, `end`, `startup`, `shutdown`) being exempt from it. Starting with version 8.17, multiple `prog` declarations with the same handler name are allowed for all handlers. See Section 4.14.1 [Multiple handler definitions], page 75, for details.

C.4 Upgrading from 8.14 to 8.15

This release introduces certain changes that require the installer's attention.

MFL file suffix changed

MFL source files now have the suffix '`.mfl`' instead of '`.mf`' which was used in previous versions. This is to avoid confusion with Metafont files, which also have suffix '`.mf`'.

All installed MFL module files have been renamed.

As of version 8.15, this change is of advisory nature, and the legacy '`.mf`' suffix is still supported. Thus, for example if module '`X`' was required and the file '`X.mfl`' was not found, `mailfromd` will look for file '`X.mf`'. Nevertheless, module authors are urged to rename their module files accordingly.

In particular, the default script file is now named `mailfromd.mfl`. If it does not exist in the system configuration directory, then the file `mailfromd.mf` is looked up. Installers are advised to rename `mailfromd.mf` to `mailfromd.mfl`.

MFL module search path

Until now, `mailfromd` searched for module files in *include search path*, which was mostly due to historical reasons. Version 8.15 introduces a separate *module search path* for this purpose. See [module search path], page 109, for details.

All modules shipped with `mailfromd` are installed to the new location: *prefix*`/share/mailfromd/8.15`.

For backward compatibility, if unable to find module file in the module search path, `mailfromd` retries the search using include search path, so that existing installation will still work. A warning message is issued for each module file found in include search path, to help you locate files that need moving to the new place.

This additional search phase will remain active during a transitional period of couple of new releases to help users accommodate to the change.

Special handlers

Special handlers (see Section 4.15.1 [begin/end], page 76) are defined using the same syntax as standard milter stage handlers, i.e.:

```
prog begin
do
  ...
done
```

Old syntax (without the `prog` keyword) is still supported, but causes a deprecation warning. Module writers are advised to update their sources accordingly.

Two new special handlers are provided. See Section 4.15.2 [startup/shutdown], page 77, for details.

Deprecated features removed

Deprecated configuration statements `lock-retry-count` and `lock-retry-timeout` were removed in this version. Please use the `locking` statement instead. See Section "locking statement" in *GNU Mailutils Manual*, for a detailed decription. To make the long story short:

Deprecated keyword	**Use this instead**
`lock-retry-count`	`retry-count`
`lock-retry-timeout`	`retry-sleep`

So that, instead of

```
lock-retry-count 10;
lock-retry-timeout 1;
```

you would write:

```
locking {
  retry-count 10;
  retry-sleep 1;
}
```

C.5 Upgrading from 8.13 to 8.14

No special actions are required. There are several important changes in version 8.14, most notable ones being improved `dkim_sign` function, DNS resolver tolerance for CNAME chains, and buffered I/O.

dkim_sign

This change is important for those who use `dkim_sign` with Sendmail. See [dkim_sign and sendmail], page 204, for a detailed discussion.

CNAME chains

Mailfromd dropped support for CNAME chains in version 8.3 (see Section C.8 [820-830], page 291). This change was forced by the behavior of the `adns` resolver library, which had apparently been grounded on somewhat rigorous reading of RFCs 1034[1] and 2181[2] . Reportedly, many existing DNS servers still employ CNAME chains leading to TXT records (in particular, to DKIM TXT records), so this change made certain DKIM signatures unverifiable for `mailfromd`.

To fix this, starting from version 8.14 `mailfromd` implements a work-around that allows it to follow CNAME chains of limited length. The default length is 2 (which means CNAME pointing to a CNAME, pointing to a valid RR). It can be changed using the `dns.max-cname-chain` configuration statement (see Section 7.4 [conf-resolver], page 222).

I/O buffering

Buffered I/O can tremendously improve performance of `getdelim` and `getline`.

The global variables `io_buffering` and `io_buffer_size` (see [io_buffering], page 137) define buffering mode and associated buffer size for file descriptors returned by the subsequent calls to `open` or `spawn`. Buffering mode of an already open file descriptor can be changed using the `setbuf` function.

The `io_buffering` variable defines the buffering mode. By default it is 0 (`BUFFER_NONE`), which disables buffering for backward compatibility with the previous versions. Another possible values are: 1 (`BUFFER_FULL`) and 2 (`BUFFER_LINE`).

When set to `BUFFER_FULL`, all I/O operations become fully buffered. The buffer size is defined by the `io_buffer_size` global variable.

`BUFFER_LINE` is similar to `BUFFER_FILE` when used for input. When used for the output, the data are accumulated in buffer and actually sent to the underlying transport stream when the newline character is seen. The `io_buffer_size` global variable sets the initial value for the buffer size in this mode. The actual size can grow as needed during the I/O.

[1] `https://www.rfc-editor.org/rfc/rfc1034#section-3.6.2`

[2] `https://www.rfc-editor.org/rfc/rfc2181#section-10.1`

The default value for `io_buffer_size` is the size of the system page.

The symbolic constants `BUFFER_NONE`, `BUFFER_FULL` and `BUFFER_LINE` are defined in the `status.mf` module. E.g.:

```
require status

begin
do
  io_buffering = BUFFER_FULL
done
```

will set up all the subsequent buffer-aware Section 5.7 [I/O functions], page 137, for buffered reads and writes using the maximum buffer capacity.

The `setbuf` function (see [setbuf], page 142) changes the buffering mode and/or buffer size for an already opened stream, e.g.:

```
setbuf(fd, BUFFER_FULL, 4096)
```

For detailed discussion of various buffering modes and their effect on the I/O, see Section 5.7 [I/O functions], page 137.

C.6 Upgrading from 8.7 to 8.8

DKIM support (see Section 5.38 [DKIM], page 199) introduced in this version requires the Nettle cryptographic library[3]. It you need DKIM, make sure Nettle is installed prior to compiling `mailfromd`. Otherwise, no special actions are required.

C.7 Upgrading from 8.5 to 8.6

New configure option `--with-dbm` allows you to select any DBM flavor supported by GNU mailutils as the default DBM implementation for `mailfromd`.

C.8 Upgrading from 8.2 to 8.3 (or 8.4)

Versions 8.3 and 8.4 differ only in required minimal version of mailutils (3.3 and 3.4, correspondingly). Apart from that, the following instructions apply to both versions.

In version 8.3 I abandoned the legacy DNS resolver and switched to *GNU adns*. GNU ands is a standalone resolver library, which provides a number of advanced features. It is included in most distributions. The source code of the recent release is available from `http://www.chiark.greenend.org.uk/~ian/adns/adns.tar.gz`.

This change brought a number of user-visible changes. In particular, arbitrary limits on the sizes of the RRs are removed. Consequently, the following configurations statements are withdrawn:

[3] `http://www.gnu.org/software/nettle`

```
runtime.max-dns-reply-a
runtime.max-dns-reply-ptr
runtime.max-dns-reply-mx
max-match-mx
max-callout-mx
```

Secondly, the new resolver is less tolerant to deviations from the standard. This means that remote DNS misconfigurations that would have slipped unnoticed in previous versions, will be noticed by mailfromd 8.3. For example, a CNAME record pointing to another CNAME is treated as an error.

A new command line option was added to `mailfromd` and `calloutd`: `--resolv-conf-file`. This option instructs the programs to read resolver settings from the supplied file, instead of the default `/etc/resolv.conf`.

Another user-visible change is in handling of SPF checks. Previously, results if SPF checks were cached in a database. This proved to cause more problems than solutions and was removed in this version. As a result, the following MFL global variables have been withdrawn:

```
spf_ttl
```

```
spf_cached
spf_database
spf_negative_ttl
```

C.9 Upgrading from 7.0 to 8.0

Version 8.0 is a major rewrite, that introduces a lot of new concepts and features. Nevertheless, it is still able to run the MFL scripts from version 7.0 without modifications.

Note the following important points:

- The `listen` configuration statement withdrawn

  Use the `server milter` statement instead. See Section 7.5 [conf-server], page 223.
- The `--remove` option withdrawn

  This option was a noop since version 7.0.91.
- The use of '`%`' before variable names is no longer supported

  The '`%`' characters is used as modulo operator. See Section 4.17.4 [Arithmetic operations], page 85.
- The `debug_spec` built-in function changed signature.

  See [debug_spec], page 210.
- `listens` and `portprobe`

  The `listens` function was moved to the `portprobe` module. It is actually an alias to the `portprobe` function. If your filter uses `listens`, make sure to `require` the `portprobe` module.

  See Section 5.34 [Special test functions], page 190.
- `_pollhost`, `_pollmx`, `stdpoll`, `strictpoll`

  These functions have been moved to the `poll` module, which must be required prior to using any of them.

- The `message_header_count` function.

  This function takes an optional string argument, supplying the header name. See [message_header_count], page 157.

C.10 Upgrading from 6.0 to 7.0

The release 7.0 removes the features which were declared as obsolete in 6.0 and introduces important new features, both syntactical, at the MFL level, and operational.

Unless your filter used any deprecated features, it should work correctly after upgrade to this version. It will, however, issue warning messages regarding the deprecated features (e.g. the use of '`%`' in front of identifiers, as described below). To fix these, follow the upgrade procedure described in [upgrade procedure], page 295.

The removed features are:

- Old-style functional notation
- The use of functional operators
- Implicit concatenations
- #pragma option
- #pragma database

The MFL syntax has changed: it is no longer necessary to use '`%`' in front of a variable to get its value. To reference a variable, simply use its name, e.g.:

```
set x var + z
```

The old syntax is still supported, so the following statement will also work:

```
set x %var + %z
```

It will, however, generate a warning message.

Of course, the use of '`%`' to reference variables within a string literal remains mandatory.

Another important changes to MFL are user-defined exceptions (see Section 4.22.2 [User-defined Exceptions], page 101) and the *try–catch* construct (see Section 4.22.3 [Catch and Throw], page 101).

Several existing MFL functions have been improved. In particular, it is worth noticing that the `open` function, when opening a pipe to or from a command, provides a way to control where the command's standard error would go (see [open], page 137).

The `accept` function (or action) issues a warning if its use would cancel any modifications to the message applied by, e.g., `header_add` and similar functions. See Section 5.13 [Message modification queue], page 151, for a detailed discussion of this feature.

The most important change in `mailfromd` operation is that the version 7.0 is able to run several servers (see Section 7.5 [conf-server], page 223). Dedicated *callout servers* make it possible to run sender verifications in background, using a set of long timeouts, as prescribed by RFC 2822 (see Section 3.7 [SMTP Timeouts], page 19). This diminishes the number of false positives, allows for coping with servers showing large delays and also reduces the number of callouts performed for such servers.

This release no longer includes the `smap` utility. It was moved into a self-standing project, which by now provides much more functionality and is way more flexible than this utility was. If you are interested in using `smap`, visit `http://www.gnu.org.ua/software/smap`, for a detailed information, including pointers to file downloads.

C.11 Upgrading from 5.x to 6.0

The 6.0 release is aimed to fix several logical inconsistencies that affected the previous versions. The most important one is that until version 5.2, the filter script file contained both the actual filter script, and the run-time configuration for `mailfromd` (in form of '`#pragma option`' and '`#pragma database`' statements). The new version separates run-time configuration from the filter script by introducing a special configuration file `mailfromd.conf` (see Chapter 7 [Mailfromd Configuration], page 219).

Consequently, the '`#pragma option`' and '`#pragma database`' statements become deprecated. Furthermore, the following deprecated pragmas are removed: '`#pragma option ehlo`', '`#pragma option mailfrom`'. These pragmas became deprecated in version 4.0 (see Section C.18 [31x-400], page 298).

The second problem was that the default filter script file had '`.rc`' suffix, which usually marks a configuration file, not the source. In version 6.0 the script file is renamed to `mailfromd.mf`. In the absence of this file, the legacy file `mailfromd.rc` is recognized and parsed. This ensures backward compatibility.

This release also fixes various inconsistencies and dubious features in the MFL language.

The support for unquoted literals is discontinued. This feature was marked as deprecated in version 3.0.

The following features are deprecated: '`#pragma option`' (pragma-option (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/pragma_002doption.html`) and '`#pragma database`' (pragma-database (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/pragma_002ddatabase.html`)) directives, the legacy style of function declarations (old-style function declarations (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/old_002dstyle-function-declarations.html`)), calls to functions of one argument without parentheses (operational notation (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/implicit-concatenation.html`)), the '`#require`' statement (See Section 4.24.3 [import], page 109, for the new syntax) and implicit concatenation (implicit concatenation (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/implicit-concatenation.html`)). See Deprecated Features (`http://mailfromd.man.gnu.org.ua/historic/6/html_node/Deprecated-Features.html`), for more information about these.

This release also introduces important new features, which are summarized in the table below:

Feature	**Reference**
Configuration	See Chapter 7 [Mailfromd Configuration], page 219.
Module system	See Section 4.24 [Modules], page 108.
Explicit type casts	See [explicit type casts], page 89.
Concatenation operator	See Section 4.17.3 [Concatenation], page 85.
Scope of visibility	See Section 4.24.2 [scope of visibility], page 109.
Precious variables	See Section 3.10 [rset], page 22.

`Mailfromd` version '`6.0`' will work with unchanged scripts from '`5.x`'. When started, it will verbosely warn you about any deprecated constructs that are used in your filter sources and will create a script for upgrading them.

To upgrade your filter scripts, follow the steps below:

1. Run '`mailfromd --lint`'. You will see a list of warnings similar to this:

```
mailfromd: Warning: using legacy script file
 /usr/local/etc/mailfromd.rc
mailfromd: Warning: rename it to /usr/local/etc/mailfromd.mf
 or use script-file statement in /usr/local/etc/mailfromd.conf
 to disable this warning
mailfromd: /usr/local/etc/mailfromd.rc:19: warning: this pragma is
 deprecated: use relayed-domain-file configuration statement instead
mailfromd: /usr/local/etc/mailfromd.rc:23: warning: this pragma is
 deprecated: use io-timeout configuration statement instead
mailfromd: Info: run script /tmp/mailfromd-newconf.sh
 to fix the above warnings
...
```

2. At the end of the run `mailfromd` will create a shell script `/tmp/mailfromd-newconf.sh` for fixing these warnings. Run it:

```
$ sh /tmp/mailfromd-newconf.sh
```

3. When the script finishes, run `mailfromd --lint` again. If it shows no more deprecation warnings, the conversion went correctly. Now you can remove the upgrade script:

```
$ rm /tmp/mailfromd-newconf.sh
```

Notice, that the conversion script attempts to fix only deprecation warnings. It will not try to correct any other type of warnings or errors. For example, you may get warning messages similar to:

```
mailfromd: /etc/mailfromd.mf:7: warning: including a module file is unreliable and may
mailfromd: /etc/mailfromd.mf:7: warning: use `require dns' instead
```

This means that you use '`#include`' where you should have used '`require`'. You will have to fix such warnings manually, as suggested in the warning message.

If, for some reason, you cannot upgrade your scripts right now, you may suppress deprecation warnings by setting the environment variable `MAILFROMD_DEPRECATION` to '`no`' before starting `mailfromd`. Nonetheless, I recommend to upgrade as soon as possible, because the deprecated features will be removed in version '`6.1`'.

C.12 Upgrading from 5.0 to 5.1

Upgrading from 5.0 to 5.1 does not require any changes in your filter scripts. Notice, however, the following important points:

- Starting from this release `mailfromd` supports Milter protocol version 6, which is compatible with Sendmail 8.14.0 and newer. While being backward compatible with earlier Sendmail releases, it allows you to use the new '`prog data`' handler (see Section 4.14 [Handlers], page 70). It also supports macro negotiation, a feature that enables Mailfromd to ask MTA to export the macros it needs for each particular handler. This

means that if you are using Sendmail 8.14.0 or higher (or Postfix 2.5 or higher), you no longer need to worry about exporting macro names in `sendmail.cf` file.

The same feature is also implemented on the server side, in `mtasim` and `pmult`. Consequently, using `define-macros` in `pmult` configuration file is not strictly necessary. However, keep in mind that due to the specifics of MeTA1, the number of symbols that may be exported for each stage is limited (see Section 13.1.2 [pmult-macros], page 272).

- The semantics of `__preproc__` and `__statedir__` built-in constant is slightly different from what it used to be in 5.0. These constants now refer to the *current* values of the preprocessor command line and program state directory, correspondingly. This should not affect your script, unless you redefine the default values on run time. If your script needs to access default values, use `__defpreproc__` and `__defstatedir__`, correspondingly (see Section 4.11.1 [Built-in constants], page 63). The following example explains the difference between these:

```
$ cat pval.mf
prog envfrom
do
  echo "Default value: " __defstatedir__
  echo "Current value: " __statedir__
done
$ mailfromd --state-directory=/var/mfd --test pval.mf
Default value: /usr/local/var/mailfromd
Current value: /var/mfd
```

- If your filter uses the `rate` function, you might consider using the new function `rateok` or `tbf_rate` instead. For a detailed discussion of these functions, see Section 3.12 [Sending Rate], page 24.
- If your script extensively uses database access functions, you might be interested in the new `#pragma dbprop` (see Section 4.5.4 [dbprop], page 57).

C.13 Upgrading from 4.4 to 5.0

This version of Mailfromd requires GNU mailutils (`http://www.gnu.org/software/mailutils`) version 2.0 or later.

Upgrading from version 4.4 to 5.0 requires no additional changes. The major differences between these two versions are summarized below:

1. Support for '`MeTA1`'.
2. New '`Mailutils`' configuration file.
3. New MFL functions.
   a. Message functions. See Section 5.19 [Message functions], page 156.
   b. Mailbox functions. See Section 5.18 [Mailbox functions], page 154.
   c. Mail body functions. See Section 5.15 [Mail body functions], page 153.
   d. Header modification functions. See Section 5.11 [Header modification functions], page 148.
   e. Envelope modification functions. See Section 5.10 [Envelope modification functions], page 147.

f. Quarantine functions. See Section 5.20 [Quarantine functions], page 161.
g. `getopt` and `varptr`. See Section 3.17.2 [getopt], page 37.
h. Macro access functions. See Section 5.1 [Macro access], page 125.
i. Character type functions. See Section 5.6 [Character Type], page 136.
j. New string functions (see Section 5.4 [String manipulation], page 130): `verp_extract_user`, `sa_format_report_header`, `sa_format_score`.
k. Sequential access to DBM files. See [dbm-seq], page 177.

4. Changes in MFL
   1. See [variadic functions], page 80.
   2. See [function alias], page 82.
5. New operation mode: See Section 3.17 [Run Mode], page 35.
6. Improved stack growth technique.

   The stack can be grown either by fixed size blocks, or exponentially. Upper limit can be specified. See Section 4.5.2 [stacksize], page 55.
7. Milter ports can be specified using URL notation.
8. Removed deprecated features.

   Support for some deprecated features has been withdrawn. These are:
   a. Command line options `--ehlo`, `--postmaster-email`, and `--mailfrom`. These became deprecated in version 4.0. See Section C.18 [31x-400], page 298.

C.14 Upgrading from 4.3.x to 4.4

The deprecated `--domain` command line option has been withdrawn. The short option `-D` now defines a preprocessor symbol (see Section 8.1.3 [Preprocessor Options], page 236).

This version correctly handles name clashes between constants and variables, which remained unnoticed in previous releases. See [variable–constant shadowing], page 91, for a detailed description of it.

To minimize chances of name clashes, all symbolic exception codes has been renamed by prefixing them with the '`e_`', thus, e.g. `divzero` became `e_divzero`, etc. The `ioerr` exception code is renamed to `e_io`. See [status.mf], page 99, for a full list of the new exception codes.

For consistency, the following most often used codes are available without the '`e_`' prefix: success, not_found, failure, temp_failure. This makes most existing user scripts suitable for use with version 4.4 without any modification. If your script refers to any exception codes other than these four, you can still use it by defining a preprocessor symbol `OLD_EXCEPTION_CODES`, for example:

```
$ mailfromd -DOLD_EXCEPTION_CODES
```

C.15 Upgrading from 4.2 to 4.3.x

Upgrading from 4.2 to 4.3 or 4.3.1 does not require any changes to your configuration and scripts. The only notable change in these versions is the following:

The asynchronous syslog implementation was reported to malfunction on some systems (notably on Solaris), so this release does not enable it by default. The previous meaning

of the `--enable-syslog-async` configuration option is also restored. Use this option in order to enable asynchronous syslog feature. To set default syslog implementation, use `DEFAULT_SYSLOG_ASYNC` configuration variable (see [syslog-async], page 11).

The following deprecated features are removed:

1. `#pragma option ehlo` statement.

   It became deprecated in version 4.0. See [pragma-option-ehlo], page 299.

2. `#pragma option mailfrom` statement.

   It became deprecated in version 4.0. See [pragma-option-ehlo], page 299.

3. The `--config-file` command line option.

   It became deprecated in version 3.1. See Section C.19 [30x-31x], page 299.

4. Built-in exception codes in catch statements.

   They are deprecated since version 4.0. See Section C.18 [31x-400], page 298.

C.16 Upgrading from 4.1 to 4.2

Upgrading to this version does not require any special efforts. You can use your configuration files and filter scripts without any changes. The only difference worth noticing is that starting from this version `mailfromd` is always compiled with asynchronous syslog implementation. The `--enable-syslog-async` configuration file option is still available, but its meaning has changed: it sets the *default* syslog implementation to use (see [syslog-async], page 11). Thus, it can be used the same way it was in previous versions. You can also select the syslog implementation at run time, see Section 3.19 [Logging and Debugging], page 43, for more detailed information.

C.17 Upgrading from 4.0 to 4.1

Upgrading to this version does not require any special efforts. You can use your configuration files and filter scripts without any changes. Notice only the following major differences between 4.1 and 4.0:

- Input files are preprocessed before compilation. See Section 4.26 [Preprocessor], page 116, for more information.
- There is a way to discern between a not-supplied optional parameter, and a supplied one, having null value (see [defined], page 118).
- The version 4.1 implements `sprintf` function (see Section 5.5 [String formatting], page 133) and `printf` macro (see Section 4.26 [Preprocessor], page 116).
- Support for some obsolete features is withdrawn. These include:
  1. Using '`&code`' to specify exception codes
  2. Pragma options: `retry`, `io-retry`, and `connect-retry`.

C.18 Upgrading from 3.1.x to 4.0

Before building this version, please re-read the chapter See Chapter 2 [Building], page 9, especially the section [syslog-async], page 11.

Starting from the version 4.0, MFL no longer uses the predefined symbolic names for exception codes (previous versions used the '`&`' prefix to dereference them). Instead, it relies on constants defined in the include file `status.mfh` (see [status.mf], page 99).

However, the script files from 3.1 series will still work, but the following warning messages will be displayed:

```
Warning: obsolete constant form used: &failure
Warning: remove leading '&' and include <status.mfh>

Warning: Using built-in exception codes is deprecated
Warning: Please include <status.mfh>
```

Another important difference is that pragmatic options '`ehlo`' and '`mailfromd`' are now deprecated, as well as their command line equivalents `--ehlo` and `--domain`. These options became superfluous after the introduction of `mailfrom_address` and `ehlo_domain` built-in variables. For compatibility with the previous versions, they are still supported by `mailfromd` 4.0, but a warning message is issued if they are used:

```
warning: `#pragma option ehlo' is deprecated,
 consider using `set ehlo_domain "domain.name"' instead
```

To update your startup scripts for the new version follow these steps:

1. Change `#pragma option mailfrom` *`value`* to `set mailfrom_address` *`value`*. Refer to [mailfrom_address], page 69, for a detailed discussion of this variable.
2. Change `#pragma option ehlo` *`value`* to `set ehlo_domain` *`value`*. Refer to [ehlo_domain], page 68, for a detailed discussion of this variable.
3. Include `status.mfh`. Add the following line to the top of your startup file:

   ```
   #include_once <status.mfh>
   ```
4. Remove all instances of '`&`' in front of the constants. You can use the following `sed` expression: '`s/&\([a-z]\)/\1/g`'.
5. If your code uses any of the following functions: `hostname`, `resolve`, `hasmx` or `ismx`, add the following line to the top of your script:

   ```
   #require dns
   ```

   See Section 4.24 [Modules], page 108, for a detailed description of the module system.
6. Replace all occurrences of `next` with `pass`.
7. If your code uses function `match_cidr`, add the following line to the top of your script:

   ```
   #require match_cidr
   ```

   See Section 4.24 [Modules], page 108, for a description of MFL module system.

C.19 Upgrading from 3.0.x to 3.1

1. The `mailfromd` binary no longer supports `--config-file` (`-c`) option. To use an alternative script file, give it as an argument, i.e. instead of:

   ```
   $ mailfromd --config-file file.rc
   ```

   write:

   ```
   $ mailfromd file.rc
   ```

For backward compatibility, the old style invocation still works but produces a warning message. However, if `mailfromd` encounters the `-c` option it will print a diagnostic message and exit immediately. This is because the semantics of this option will change in the future releases.

2. If a variable is declared implicitly within a function, it is created as automatic. This differs from the previous versions, where all variables were global. It is a common practice to use global variables to pass additional information between handlers (See Section 3.9 [Using Variables], page 22). If your filter uses it, make sure the variable is declared as global. For example, this code:

```
prog helo
do
  # Save the host name for further use
  set helohost $s
done
```

Figure C.1: Implicit declaration, old style

has to be rewritten as follows:

```
set helohost ""

prog helo
do
  # Save the host name for further use
  set helohost $s
done
```

Figure C.2: Implicit declaration, new style

3. Starting from version 3.1 the function `dbmap` takes an optional third argument indicating whether or not to count the terminating null character in key (see [dbmap], page 176). If your startup script contained any calls to `dbmap`, change them as follows:

in 3.0.x	**in 3.1**
dbmap(*db*, *key*)	dbmap(*db*, *key*, 1)

C.20 Upgrading from 2.x to 3.0.x

Update your startup scripts and/or crontab entries. The `mailfromd` binary is now installed in `${prefix}/sbin`.

We also encourage you to update the startup script (run *`cp etc/rc.mailfromd /wherever-your-startup-lives`*), since the new version contains lots of enhancements.

C.21 Upgrading from 1.x to 2.x

If you are upgrading from version 1.x to 2.0, you will have to do the following:

1. Edit your script file and enclose the entire code section into:

```
prog envfrom
do
  ...
done
```

See Section 4.14 [Handlers], page 70, for more information about the `prog` statement.

2. If your code contained any `rate` statements, convert them to function calls (see Section 5.32 [Rate limiting functions], page 189), using the following scheme:

   ```
   Old statement: if rate key limit / expr
   New statement: if rate(key, interval("expr")) > limit
   ```

   For example,

   ```
   rate $f 180 / 1 hour 25 minutes
   ```

   should become

   ```
   rate($f, interval("1 hour 25 minutes")) > 180
   ```

3. Rebuild your databases using the following command:

   ```
   mailfromd --compact --all
   ```

   This is necessary since the format of `mailfromd` databases has changed in version 2.0: the key field now includes the trailing '`NUL`' character, which is also reflected in its length. This allows for empty (zero-length) keys. See Section 3.15.3 [Database Maintenance], page 33, for more information about the database compaction.

Appendix D GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

D.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual

!

! (exclamation point), != operator 86

#

'#! ... !#' initial comment 37
'#!' shell magic sequence 36
#error 54
#include 53
#include statement 53
#line 54
#pragma 55
#pragma dbprop 175
#pragma statement 55
#warning 54

$

$# 79
$(*n*) 80
$@ 81
$*n* 80

(

(string 202

_

--prefix, configure option 10
--sysconfdir, configure option 11
--with-dbm, configure option 9

/

/etc/postfix/main.cf 246
/etc/resolv.conf 223

<

< (left angle bracket), < operator 86
< (left angle bracket), <= operator 86

=

= (equals sign), = operator 86

>

> (right angle bracket), > operator 86
> (right angle bracket), >= operator 86

_

_ 119
__defpreproc__ 64
__defstatedir__ 64
__file__ 63
__function__ 63
__git__ 63
__line__ 63
__major__ 64
__minor__ 64
__module__ 64
__package__ 64
__patch__ 64
__preproc__ 64
__statedir__ 64
__version__ 64
_expand_dataseg 212
_heap_reserve 212
_pollhost 163
_pollmx 163
_reg 212
_register 212
_stack_free 212
_wd 212

@

@*var*, special construct 79

\

\D 267
\E 267
\L 267
\S 267
\U 267

~

~/.emacs 215

7

7bit 143

8

8bit 143

A

`a, -a, mtasim option, summary` ... 268
`accept` ... 92
accept action, defined ... 92
accept action, introduced ... 13
`accept` in '`end`' ... 76
`ACCEPT_ACTION` ... 77
`access` ... 180
accessing variables from `catch` ... 104
account probing ... 6
`acl` ... 225, 251
action hook ... 77
actions ... 92
actions, header manipulation ... 93
actions, introduced ... 15
actions, using in `connect` handler ... 71
`add` ... 93
add action, defined ... 93
`add` in '`begin`' ... 76
`add` in '`end`' ... 76
adns ... 9
Alan Dobkin ... 3
`alias` ... 82
aliases ... 82
aliases, looking up ... 30
`all, --all` mfdbtool option, introduced ... 33
`all, --all` mfdbtool option, summary ... 259
`always` ... 228, 255
and ... 87
`append, --append, mtasim` option, described ... 266
`append, --append, mtasim` option, summary ... 268
argument number in the list of arguments ... 79
arguments, `catch` ... 103
arguments, optional ... 79
as ... 107
`ASSERT_ARGCOUNT` ... 113
`ASSERT_ARGTYPE` ... 113
assignment to undeclared variable ... 22
assignment to variable ... 22
assignment, defined ... 94
associativity, operators ... 87
asynchronous syslog ... 43
`auth-macros` ... 274
automatic variables ... 82

B

back reference interpretation ... 60
back references, in program text ... 69
`backlog` ... 224, 251
backslash interpretation ... 59
`base64` ... 144
`begin` ... 76
'`begin`' and `add` ... 76
'`begin`' and `delete` ... 76
'`begin`' and `replace` ... 76
'`begin`', handler restrictions ... 76
begin, special handler ... 13, 76
Ben McKeegan ... 3
Berkeley DB ... 9
`binary` ... 143
`bindtextdomain` ... 208
`body` ... 73
body, handler ... 13
`body-chunk, --body-chunk, mtasim` option, summary ... 268
`body_has_nulls` ... 153
`body_string` ... 153
`break` ... 97
break statement ... 97
Brent Spencer ... 3
`bs, -bd, mtasim option, summary` ... 268
`bs, -bs, mtasim option, summary` ... 270
building `mailfromd` ... 9
built-in and library functions, introduced ... 17
built-in constants ... 63
`burst_eb_min_length` ... 161
`BURST_DECODE` ... 160
`BURST_ERR_BODY` ... 160
`BURST_ERR_FAIL` ... 160
`BURST_ERR_IGNORE` ... 160
`bye` ... 108
`B` ... 144

C

cache database ... 31
cache, disabling ... 179
cache, getting status ... 179
`cache_used` ... 67
`cache_used` variable, usage example ... 67
cache_used, global variable, introduced ... 164
caching DNS requests ... 7
`callout` ... 162, 224
callout server ... 19
callout, described ... 6
`callout-socket, --callout-socket` mailfromd option, summary ... 234
`callout-url` ... 224
`callout_close` ... 161
`callout_do` ... 161
`callout_open` ... 161
`callout_transcript` ... 210
`calloutd` ... 249
`cancel_program_trace` ... 211
`case` ... 95
`case`, `switch` statement ... 95
`catch` ... 101
`catch` arguments ... 103
`catch` scope ... 102
`catch` statement ... 101
`catch`, accessing variables from ... 104
`catch`, returning from ... 103
catch, standalone ... 102
`cdb.mfl` ... 179
`cdb_check` ... 179

`cdb_greylist_interval` ... 179
`cdb_name` ... 179
`charset` ... 144
`check_host` ... 198
`check_host` function, introduced ... 195
checking SPF host records ... 195
`clamav` ... 188
`clamav_stream_max_length` ... 68
clamav_stream_max_length, global vaiable ... 188
`clamav_virus_name` ... 68
clamav_virus_name, global variable ... 188
ClamAV ... 188
clamd.conf ... 68, 188
`client` ... 275
client_addr, `Sendmail` macro ... 243
`close` ... 139
CNAME chains ... 223
`command` ... 117, 222
command line arguments, parsing in MFL ... 37
command line, `mailfromd` invocation syntax ... 233
comments ... 53
`compact, --compact` mfdbtool option, introduced ... 33
`compact, --compact` mfdbtool option, summary ... 259
compaction, database ... 33
Con Tassios ... 3
Con Tassios greylisting type ... 28
concatenation ... 85
conditional statements ... 95
`config` ... 223
`config-file, --config-file` calloutd option, summary ... 253
`config-help, --config-help` calloutd option, summary ... 254
`config-lint, --config-lint` calloutd option, summary ... 253
`config-verbose, --config-verbose` calloutd option, summary ... 253
configuring the preprocessor ... 117
confMAPDEF, `Sendmail` macro ... 9
confMILTER_MACROS_ENVFROM, mc file directive ... 243
`connect` ... 70
connect, handler ... 13
`connect-timeout` ... 275
`connection` ... 226
const ... 62
constants, built-in ... 63
constants, defining ... 62
constants, using in literals ... 63
constants, using in program text ... 62
`continue` ... 92
continue action, defined ... 92
continue action, introduced ... 13
`continue` in '`end`' ... 76
`CONTINUE_ACTION` ... 77
controlling argument, getopt ... 39
`copy` ... 140
`copy-octal` ... 145
`copy-pass` ... 145
`create_dsn` ... 194
`crlf` ... 144
`crlfdot` ... 144
cross-reference ... 34
ctype_mismatch, global variable ... 136
`current_header` ... 154
`current_header_count` ... 154
`current_header_nth_name` ... 154
`current_header_nth_value` ... 154
`current_message` ... 154
customization, Emacs ... 216
customization, MFL mode ... 216

D

D, -D option, described ... 118
D, -D option, summary ... 236
`D, -D, mtasim option, summary` ... 268
D, `\D`, a `mtasim` command ... 263
`daemon, --daemon` mailfromd option, summary ... 233
`daemon, --daemon, mtasim` option, described ... 266
`daemon, --daemon, mtasim` option, summary ... 268
`data` ... 73
data, handler ... 13
`database` ... 229
database compaction ... 33
database formats ... 31
database maintenance ... 33
database, listing ... 32
`database-mode` ... 230
`database-type` ... 230
databases used by `mailfromd` ... 31
`db_expire_interval` ... 178
`db_get_active` ... 179
`db_name` ... 178
`db_set_active` ... 179
`dbbreak` ... 178
`dbdel` ... 177
`dbfirst` ... 178
`dbget` ... 176
`dbinsert` ... 177
`dbkey` ... 178
DBM implementation, default ... 230
DBM scheme ... 229
`dbmap` ... 176
DBM ... 9
`dbnext` ... 178
`dbprop` ... 57
`dbprop,` ... 175
dbprop, pragma ... 175
`dbput` ... 176
`dbvalue` ... 178
`dc` ... 128
`dclex` ... 101

`debug` ... 209, 225, 252, 276
`debug, --debug` calloutd option, summary ... 253
`debug, --debug` mailfromd option, introduced ... 45
`debug, --debug` mailfromd option, summary ... 236
`debug, --debug` mfdbtool option, summary ... 259
`debug-level, --debug-level` calloutd option, summary ... 253
`debug_level` ... 210
`debug_spec` ... 210
debugging ... 95
debugging level ... 45
debugging the filter script ... 34
debugging, pmult ... 272
declaring milter state handler ... 15
decode MIME ... 158, 159
`default` ... 224, 254
default communication port ... 11
default communication socket ... 11
default database type ... 230
default exception handling ... 101
default expiration interval ... 11
default syslog facility ... 44
default user privileges ... 10
`default_callout_server_url` ... 162
DEFAULT_EXPIRE_INTERVAL, `configure` variable ... 11
DEFAULT_EXPIRE_RATES_INTERVAL, `configure` variable ... 11
DEFAULT_SOCKET, `configure` variable ... 11
DEFAULT_STATE_DIR, `configure` variable ... 11
`DEFAULT_SYSLOG_ASYNC`, `configure` variable ... 12, 297
`DEFAULT_USER`, `configure` variable ... 10
`define, --define` mailfromd option, described ... 118
`define, --define` mailfromd option, summary ... 236
`define, --define, mtasim` option, summary ... 268
`define-macros` ... 273
`defined` ... 118
`delete` ... 94
delete action, defined ... 94
`delete` in '`begin`' ... 76
`delete` in '`end`' ... 76
`delete, --delete` mfdbtool option, introduced ... 33
`delete, --delete` mfdbtool option, summary ... 259
`dequote` ... 131
`dgettext` ... 208
diagnostics channel ... 43
digest, message ... 160
disabling cache ... 179
`discard` ... 92
discard action, defined ... 92
discard action, introduced ... 13
`discard` in '`end`' ... 76
`DISCARD_ACTION` ... 77
dkim, module ... 202
`dkim_explanation` ... 200
`dkim_explanation_code` ... 200
`dkim_sendmail_commaize` ... 205
`dkim_sign` ... 202
`dkim_signing_algorithm` ... 200
`dkim_verified_signature` ... 201
`dkim_verified_signature_tag` ... 202
`dkim_verify` ... 199
DKIM ... 8
DKIM, defined ... 199
DKIM, setting up ... 205
`DKIM_EXPL_BAD_ALGORITHM` ... 200
`DKIM_EXPL_BAD_BASE64` ... 201
`DKIM_EXPL_BAD_BODY` ... 201
`DKIM_EXPL_BAD_KEY_TYPE` ... 201
`DKIM_EXPL_BAD_QUERY` ... 200
`DKIM_EXPL_BAD_SIG` ... 201
`DKIM_EXPL_BAD_VERSION` ... 200
`DKIM_EXPL_DNS_NOTFOUND` ... 201
`DKIM_EXPL_DNS_UNAVAIL` ... 201
`DKIM_EXPL_DOMAIN_MISMATCH` ... 200
`DKIM_EXPL_EXPIRED` ... 201
`DKIM_EXPL_FROM` ... 201
`DKIM_EXPL_INTERNAL_ERROR` ... 200
`DKIM_EXPL_KEY_REVOKED` ... 201
`DKIM_EXPL_KEY_SYNTAX` ... 201
`DKIM_EXPL_NO_SIG` ... 200
`DKIM_EXPL_OK` ... 200
`DKIM_EXPL_SIG_MISS` ... 200
`DKIM_EXPL_SIG_SYNTAX` ... 200
`DKIM_VERIFY_OK` ... 199
`DKIM_VERIFY_PERMFAIL` ... 199
`DKIM_VERIFY_TEMPFAIL` ... 199
`dlcall` ... 213
`dlopen` ... 213
`dngettext` ... 208
`dns.mfl` ... 166, 168
`dns_getaddr` ... 168
`dns_getname` ... 168
`dns_query` ... 166
`dns_reply_count` ... 168
`dns_reply_ip` ... 168
`dns_reply_release` ... 167
`dns_reply_string` ... 168
`DNS_TYPE_A` ... 166
`DNS_TYPE_MX` ... 166
`DNS_TYPE_NS` ... 166
`DNS_TYPE_PTR` ... 166
`DNS_TYPE_TXT` ... 167
do loop ... 98
DomainKeys Identified Mail ... 199
`domainpart` ... 130
`dot` ... 144
`drop` ... 257
`dump-code, --dump-code` mailfromd option, summary ... 236
`dump-grammar-trace, --dump-grammar-trace` mailfromd option, summary ... 236
`dump-lex-trace, --dump-lex-trace` mailfromd option, summary ... 237

`dump-macros, --dump-macros` mailfromd option, described ... 243
`dump-macros, --dump-macros` mailfromd option, summary ... 237
`dump-tree, --dump-tree` mailfromd option, summary ... 237
`dump-xref, --dump-xref` mailfromd option, summary ... 237

E

`e_badmmq` ... 99
`e_dbfailure` ... 99
`e_divzero` ... 99
`e_eof` ... 99
`e_exists` ... 99
`e_failure` ... 99
`e_format` ... 99
`e_ilseq` ... 100
`e_inval` ... 100
`e_invcidr` ... 100
`e_invip` ... 100
`e_invtime` ... 100
`e_io` ... 100
`e_macroundef` ... 100
`e_not_found` ... 100
`e_range` ... 100
`e_regcomp` ... 100
`e_ston_conv` ... 100
`e_success` ... 101
`e_temp_failure` ... 101
`e_too_many` ... 101
`e_url` ... 101
E, -E option, described ... 117
E, -E option, summary ... 236, 237
E, `\E`, a `mtasim` command ... 265
`echo` ... 95
`echo, --echo` mailfromd option, echo to stdout or file ... 35
`echo, --echo` mailfromd option, summary ... 234
`ehlo` ... 255
`ehlo-domain` ... 227
`ehlo_domain` ... 68
`elif` ... 95
`else` ... 95
Emacs, MFL mode ... 215
`email.mfl` ... 147
`email_map` ... 147
`email_valid` ... 147
`EMAIL_COMMENTS` ... 147
`EMAIL_DOMAIN` ... 147
`EMAIL_LOCAL` ... 147
`EMAIL_MULTIPLE` ... 147
`EMAIL_PERSONAL` ... 147
`EMAIL_ROUTE` ... 147
`enable` ... 222, 230
enable-syslog-async, `--enable-syslog-async`, `configure` option ... 11
`enable-vrfy` ... 228
Enabling MFL mode ... 215
encapsulation boundaries, RFC 934 ... 160
`end` ... 76
'`end`' and `accept` ... 76
'`end`' and `add` ... 76
'`end`' and `continue` ... 76
'`end`' and `delete` ... 76
'`end`' and `discard` ... 76
'`end`' and `reject` ... 76
'`end`' and `replace` ... 76
'`end`' and `tempfail` ... 76
'`end`', handler restrictions ... 76
end, special handler ... 13, 76
enumeration ... 62
`envfrom` ... 72
envfrom, handler ... 13
`envrcpt` ... 72
envrcpt, handler ... 13
`eoh` ... 73
eoh, handler ... 13
`eom` ... 74
eom, handler ... 13
`eom-timeout` ... 276
equals sign (`=`), `=` operator ... 86
`escape` ... 130
estimated time of sending, prediction of ... 32
exception handler scope ... 102
exception handler, returning from ... 103
exception handlers ... 101
exception types ... 99
exception-handling routines ... 101
exceptions, default handling ... 101
exceptions, defined ... 99
exceptions, raising from code ... 104
exceptions, symbolic names ... 99
exclamation point (`!`), `!=` operator ... 86
expect mode, `mtasim` ... 265
`expire, --expire` mfdbtool option, introduced ... 33
`expire, --expire` mfdbtool option, summary ... 259
`expire-interval` ... 230
`expire-interval, --expire-interval` mfdbtool option, summary ... 259
explicit type casts ... 89
expressions ... 85

F

f, `Sendmail` macro ... 243
`F_OK` ... 180
Fail, SPF result code ... 195
FAMILY_INET ... 71
FAMILY_INET6 ... 71
FAMILY_STDIO ... 71
FAMILY_UNIX ... 71
fatal runtime errors ... 48
`fd_delimiter` ... 141
`fd_set_delimiter` ... 141

FDL, GNU Free Documentation License 303
fi 95
file 229
file, --file mfdbtool option, summary 259
filter pipe 143
filter script, debugging 34
filter script, described 13
filter script, running in test mode 34
filter_fd 143
filter_string 143
Finding function definition 215
fnmatches 86
for loop 97
foreground, --foreground calloutd option, summary 252
foreground, --foreground mailfromd option, summary 234
format, --format mfdbtool option, introduced . 32
format, --format mfdbtool option, summary . 260
format, --format mfdbtool option, using with --list 32
from 107, 110, 145
from ... import 110
fromrd 145
func statement, function definition 79
function arguments, counting 79
function arguments, getting the number of 79
function calls 17
function definition, syntax of 78
function returning void 81
function, defined 17

G

g, -g, mtasim option, summary 268
g, transform flag 128
gacopyz-log, --gacopyz-log mailfromd option, summary 237
gacopyz-log, --gacopyz-log, mtasim option, summary 268
GDBM 9
geoip2_dbname 173
geoip2_get 175
geoip2_get_json 175
geoip2_open 173
GeoIP2 173
geolocation 173
get 255
getbufsize(number 142
getbuftype(number 142
getdelim 141
getdomainname 181
getenv 181
gethostname 181
getline 141
getmacro 125
getmx 168
getns 172
getopt 37
getpwnam 182
getpwuid 182
gettext 209
getting cache status 179
global cleanup handler 77
global startup handler 77
globbing patterns 86
GNU Emacs, MFL mode 215
GNU Readline 261
greylist 57, 190
greylist database 32
greylist_seconds_left 68
greylist_seconds_left, global variable 190
greylist_seconds_left, global variable, introduced . 27
greylisting types 28
greylisting, Con Tassios type 28
greylisting, traditional 28
group 229
group, --group calloutd option, summary 252
group, --group mailfromd option, summary .. 234
group, --group, mtasim option, described 263
group, --group, mtasim option, summary 268
groups 239
growth policy, stack 55

H

handler arguments 75
handler declaration 15
handler, defined 70
handler, described 13
handler, global cleanup 77
handler, global startup 77
handler, session cleanup 76
handler, session initialization 76
handler, session startup 76
handshake-timeout 228
hard STMP timeout 19
hasmx 169
hasmx, definition of the function 103
hasns 172
header 73, 145
header manipulation actions 93
header modification 148
header, handler 13
header_add 148
header_delete 149
header_insert 148
header_prefix_all 149
header_prefix_pattern 150
header_rename 149
header_rename.mfl 149, 150
header_replace 149
'Heap overrun; increase #pragma stacksize', runtime error 48
helo 72, 226
helo, handler 13

heloarg_test ... 191
heloarg_test.mfl ... 191
help, --help calloutd option, summary ... 254
HELP, mtasim statement ... 261
here document ... 60
host ... 255
hostfirst ... 255
hostname ... 169
hostonly ... 255
htonl ... 164
htons ... 164

I

i, Sendmail macro ... 243
i, Sendmail macro in MeTA1 ... 272
i, Sendmail macro in Postfix ... 247
i, transform flag ... 128
i18n ... 207
iconv ... 145
id ... 223, 251
if ... 95
ignore-failed-reads, --ignore-failed-reads mfdbtool option, summary ... 260
implicit type casts ... 88
implicit variable declaration ... 22
import ... 110
importing from modules ... 109
include search path, introduced ... 53
include-path ... 221
include-path, --include-path mailfromd option, summary ... 234
include_once ... 54
including files ... 53
indentation, MFL, default ... 215
index ... 130
inet_aton ... 164
inet_ntoa ... 164
infinite loop ... 97
initial-response ... 226
inline-comment ... 145
INPUT_MAIL_FILTER, mc file directive ... 243
internationalization ... 207
interval ... 130
'Invalid back-reference number', runtime error ... 49
'Invalid exception number', runtime error ... 49
invocation ... 233
io-timeout ... 227
io_buffer_size ... 137
io_buffering ... 137
is_greylisted ... 190
is_ip ... 288
is_ip.mfl ... 288
is_ip4str(string ... 165
is_ip6str(string ... 165
is_ipstr(string ... 165
isalnum ... 136
isalpha ... 136
isascii ... 136
isblank ... 136
iscntrl ... 136
isdigit ... 136
isgraph ... 136
islower ... 136
ismx ... 170
isprint ... 136
ispunct ... 136
isspace ... 136
isupper ... 136
isxdigit ... 137

J

Jan Rafaj ... 3
Jeff Ballard ... 3
John McEleney ... 3

K

keywords ... 122

L

l10n ... 207
L, \L, a mtasim command ... 264
last_poll_greeting ... 68
last_poll_helo ... 68
last_poll_host ... 68
last_poll_host, global variable, introduced ... 164
last_poll_recv ... 69
last_poll_recv, global variable, introduced ... 164
last_poll_sent ... 69
last_poll_sent, global variable, introduced ... 164
left angle bracket (<), < operator ... 86
left angle bracket (<), <= operator ... 86
len_to_netmask ... 165
length ... 131
libmaxminddb ... 173
library and built-in functions, introduced ... 17
line, #line statement ... 54
linecon ... 146
linelen ... 146
lint mode ... 33
lint, --lint mailfromd option, introduced ... 33
lint, --lint mailfromd option, summary ... 237
list, --list mfdbtool option, described ... 32
list, --list mfdbtool option, summary ... 259
listen ... 223, 251, 272
listens ... 190
listing a database contents ... 32
literal concatenation ... 85
literals ... 59
local state directory ... 11
local variables ... 82
localdomain ... 181

`localdomain.mfl` ... 181
localization ... 207
`localpart` ... 131
`location-column, --location-column` mailfromd option, described ... 33
`location-column, --location-column` mailfromd option, summary ... 236
`log-facility, --log-facility` calloutd option, summary ... 253
`log-facility, --log-facility` mailfromd option, introduced ... 44
`log-facility, --log-facility` mailfromd option, summary ... 238
`log-level` ... 276
`log-tag, --log-tag` calloutd option, summary ... 253
`log-tag, --log-tag` mailfromd option, introduced ... 44
`log-tag, --log-tag` mailfromd option, summary ... 238
`logger` ... 225, 251
`logger, --logger` calloutd option, summary ... 253
`logger, --logger` mailfromd option, introduced ... 43
`logger, --logger` mailfromd option, summary ... 238
`loop` ... 96
loop body ... 97
loop statement ... 96
loop, do-style ... 98
loop, for-style ... 97
loop, infinite ... 97
loop, while-style ... 97
`ltrim` ... 132

M

m4 ... 116
macro expansion ... 59
`macro_defined` ... 125
macros, MeTA1 ... 272
macros, referencing ... 61
`mail` ... 227
mail filtering language ... 53
mail sending rate, explained ... 8
Mail Transfer Agent (MTA) ... 13
`mail-from-address` ... 227
mailbox functions ... 154
`mailbox_append_message` ... 155
`mailbox_close` ... 155
`mailbox_get_message` ... 155
`mailbox_messages_count` ... 155
`mailbox_open` ... 154
mailer URL ... 191
`mailer, --mailer` mailfromd option, summary ... 234
`mailfrom` ... 255
`mailfrom_address` ... 69
`mailfromd`, building ... 9
mailutils ... 9
`mailutils_set_debug_level` ... 209
main, MFL function ... 35
maintenance, database ... 33
`mappwnam` ... 183
`mappwuid` ... 183
`match_cidr` ... 165
`match_cidr.mfl` ... 288
`match_dnsbl` ... 194
match_dnsbl, definition ... 84
`match_dnsbl.mfl` ... 194
`match_rhsbl` ... 194
`match_rhsbl.mfl` ... 194
`matches` ... 86
`max-cname-chain` ... 223
`max-instances` ... 224, 251
`max-mfmods` ... 231
`max-open-mailboxes` ... 231
`max-open-messages` ... 231
`max-pmilter-fd` ... 272
`max-streams` ... 231
`max-threads-hard` ... 272
`max-threads-soft` ... 272
`MaxRecipientsPerMessage`, `sendmail` option ... 23
'`memory chunk too big to fit into heap`', runtime error ... 48
message digest ... 160
message functions ... 156
message modification queue ... 151
Message-ID, exporting ... 44
Message-ID, exporting in `mc` file ... 243
Message-ID, using in `mailfromd` logs ... 44
`message_body_decode` ... 159
`message_body_is_empty` ... 156
`message_body_lines` ... 158
`message_body_rewind` ... 158
`message_body_size` ... 157
`message_body_to_stream` ... 158
`message_burst` ... 160
`message_close` ... 156
`message_content_type` ... 158
`message_count_parts` ... 158
`message_find_header` ... 157
`message_from_stream` ... 156
`message_get_part` ... 158
`message_has_header` ... 157
`message_header_count` ... 157
`message_header_decode` ... 152
`message_header_encode` ... 152
`message_header_lines` ... 157
`message_header_size` ... 157
`message_is_multipart` ... 158
`message_lines` ... 156
`message_nth_header_name` ... 157
`message_nth_header_value` ... 157
`message_part_decode(number` ... 160
`message_read_body_line` ... 158

message_read_line 156
message_rewind 156
message_size 156
message_to_stream 156
meta1 244
meta1 macros 272
MF_SIEVE_DEBUG_INSTR 183
MF_SIEVE_DEBUG_TRACE 183
MF_SIEVE_FILE 183
MF_SIEVE_LOG 183
MF_SIEVE_TEXT 183
mfdbtool 32, 259
mfl-basic-offset 216
mfl-case-line-offset 216
mfl-comment-offset 217
mfl-include-path 216
mfl-loop-continuation-offset 217
mfl-loop-statement-offset 217
mfl-mailfromd-command 216
mfl-mode.el 215
mfl-returns-offset 217
MFL 53
MFL mode, ... 215
MFL mode, enabling 215
MFL mode, GNU Emacs 215
mfmod 111
mfmod, interface module 114
mfmod, loadable library 111
mfmod-path 231
mfmod.h 111
mfmod.m4 116
mfmod_data_type_str(mfmod_data_type 113
mfmod_error 112
mfmod_error_argtype 112
mfmod_message 112
mfmod_number 111
mfmod_string 111
MFMOD_PARAM 111
mfmodnew 115
milter abort 22
milter stage handler arguments 75
milter stage handler, defined 70
milter state handler, declaring 15
milter state handler, described 13
milter-actions, --milter-actions, mtasim option, summary 269
milter-proto, --milter-proto, mtasim option, summary 268
milter-socket, --milter-socket mailfromd option, summary 235
milter-timeout 225
milter-timeout, --milter-timeout mailfromd option, summary 236
milter-timeout, --milter-timeout, mtasim option, summary 269
milter-version, --milter-version, mtasim option, summary 268
milter.mfl 66
milter_action 67
milter_action_name 213
milter_client_address 67
milter_client_family 67
milter_server_address 67
milter_server_family 67
milter_server_id 67
milter_state 66
milter_state_begin 67
milter_state_body 67
milter_state_code 213
milter_state_connect 67
milter_state_data 67
milter_state_end 67
milter_state_envfrom 67
milter_state_envrcpt 67
milter_state_eoh 67
milter_state_eom 67
milter_state_header 67
milter_state_helo 67
milter_state_name 212
milter_state_none 67
milter_state_shutdown 67
milter_state_startup 67
miltermacros 58
mime.mfl 159
MIME, decoding 158, 159
mimedecode 146, 158, 159
mmq_purge 152
mode 254
module 108
module declaration 108
module search path 109
module, defined 17, 108
module-path 221
module-path, --module-path mailfromd option, summary 234
mtasim 261
mtasim administrative commands 263
mtasim auto mode 262
mtasim daemon mode 266
mtasim expect mode 265
mtasim, --mtasim mailfromd option, summary 234
mtasim, declare sender socket 264
mtasim, defining Sendmail macros 263
mtasim, introduced 35
mtasim, listing Sendmail macros 264
mtasim, undefining Sendmail macros 264
mtasim, using in shell scripts 266
MTA 13
multiline strings 60
multiple sender addresses 69
multiple sender addresses, using with polling commands 163
mx fnmatches 87
mx matches 86
mxfirst 254

`mxonly` ... 255

N

`N_` ... 119
Nacho González López ... 3
name clashes ... 89
National Language Support ... 207
Navigating through function definitions ... 215
negative expiration period, defined ... 31
`negative-expire-interval` ... 230
`netmask_to_len` ... 165
Neutral, SPF result code ... 195
`never` ... 228, 255
`next` ... 97
next statement ... 97
`ngettext` ... 209
`nls.mfl` ... 207
NLS ... 207
'`No previous regular expression`', runtime error ... 49
`no-interactive, --no-interactive, mtasim` option, summary ... 269
`no-preprocessor, --no-preprocessor` mailfromd option, summary ... 236
`no-preprocessor, --no-preprocessor` mailfromd option, usage ... 118
`no-site-config, --no-site-config` calloutd option, summary ... 254
non-blocking syslog ... 43
non_smtpd_milters, postfix configuration ... 246
`none` ... 145
None, SPF result code ... 195
not ... 87
'`Not enough memory`', runtime error ... 48
`ntohl` ... 164
`ntohs` ... 164
number of actual arguments ... 79

O

`OLD_EXCEPTION_CODES`, preprocessor symbol ... 297
`on` statement ... 106
`ondemand` ... 228, 255
`open` ... 137
operator associativity ... 87
operator precedence, defined ... 87
`optarg` ... 39
`opterr` ... 39
`optimize, --optimize` mailfromd option, summary ... 235
`optind` ... 39
`option` ... 224
optional arguments to a function ... 79
optional arguments, checking if supplied ... 79
`optopt` ... 39
or ... 87
'`Out of stack space; increase #pragma stacksize`', runtime error ... 48
overriding initial variable values ... 34

P

parsing command line arguments ... 37
`pass` ... 94
Pass, SPF result code ... 195
`pass-defines` ... 117, 222
`pass-includes` ... 117, 222
passing variable arguments on to another function ... 81
'`pc out of range`', runtime error ... 49
Peter Markeloff ... 3
Phil Miller ... 3
`pidfile` ... 221, 240, 250, 272
`pidfile, --pidfile` calloutd option, summary ... 252
`pidfile, --pidfile` mailfromd option, summary ... 235
`pies` ... 271
pipe ... 143
`pmilter-debug` ... 277
`pmult` ... 271
pmult debugging ... 272
`pmult`, described ... 271
`poll` command, standard verification ... 107
`poll` command, strict verification ... 107
`poll` keyword ... 107
`poll` statement, defined ... 107
`poll.mfl` ... 162
`port, --port` mailfromd option, summary ... 235
`port, --port, mtasim` option, described ... 262
`port, --port, mtasim` option, summary ... 269
`portprobe` ... 190
`portprobe.mfl` ... 190
positive expiration period, defined ... 31
`positive-expire-interval` ... 230
Postfix ... 246
`postfix-macros.sed` ... 247
`pp-setup` ... 116
pragmatic comments ... 55
precedence, operators ... 87
`precious` ... 23, 65
precious variables ... 23
predefined variables ... 66
`predict, --predict` mfdbtool option, introduced ... 32
`predict, --predict` mfdbtool option, summary ... 260
preprocessor ... 116, 221
preprocessor configuration ... 117
preprocessor setup file ... 116
`preprocessor, --preprocessor` mailfromd option, summary ... 236
`preprocessor, --preprocessor` mailfromd option, usage ... 118

preprocessor.command 117
preprocessor.pass-defines 117
preprocessor.pass-includes 117
preprocessor.setup-file 117
primitive_hasmx 169
primitive_hasns 172
primitive_hostname 169
primitive_ismx 170
primitive_resolve 170
printf 119
probe message 6
procedures 81
prog 70
program_trace 211
progress 153
prompt, --prompt, mtasim option, summary .. 270
ptr_validate 171
public 65, 79

Q

qr 130
qualifier, function declaration 79
qualifiers, variable declaration 65
quarantine 161
quit 227, 257
quoted-printable 146
Q 146

R

R_OK 180
raising exceptions 104
rate 189
rate database 31
rateok 189
rateok.mfl 189
rc.mailfromd 240
rcpt 227
rcpt_add 148
rcpt_count 69
rcpt_delete 148
read 140
read-timeout 276
readline 261
regex 56
regular expression matching 86
reject 92
reject action, defined 92
reject action, introduced 13
reject in 'end' 76
reject messages, marking cached rejects 67
REJECT_ACTION 77
relayed 179
relayed-domain-file 221
relayed-domain-file, --relayed-domain-file mailfromd option, summary 235
replace 94
replace action, defined 94
replace in 'begin' 76
replace in 'end' 76
replbody 150
replbody_fd 150
replstr 131
require 17, 109
requiring modules 109
reserved words 122
resolv-conf-file, --resolv-conf-file calloutd option, summary 252
resolv-conf-file, --resolv-conf-file mailfromd option, summary 235
resolve 171
RESOLVE_DFL 167, 170
RESOLVE_IP4 ..... 167, 170
RESOLVE_IP6 ..... 167, 170
RESOLVE_NONE 167
resolver 222
return statement, defined 81
returning from a catch 103
returning from an exception handler 103
returns statement, function definition 79
reuseaddr 224, 251
reverse_ipstr(string 165
revip 288
revip, definition of 83
revip.mfl 288
revstr 131
rewind 140
rfc822 144
right angle bracket (>), > operator 86
right angle bracket (>), >= operator 86
rindex 131
rset 227
RSET 22
rtrim 132
run 256
run mode 35
run, --run mailfromd option, described 35
run, --run mailfromd option, summary 233
runtime 231
runtime error 48
runtime errors, fatal 48
runtime errors, tracing 50

S

s-expression 128
S, \S, a mtasim command 264
sa 188
sa.mfl 133
sa_code 69
sa_format_report_header 133
sa_format_score 133
sa_keywords 69
sa_keywords, global variable 187
sa_score, global variable 186

`sa_threshold` ... 69
sa_threshold, global variable ... 186
`SA_FORGET` ... 186
`SA_LEARN_HAM` ... 186
`SA_LEARN_SPAM` ... 186
`SA_REPORT` ... 186
`SA_SYMBOLS` ... 186
`safedb.mfl` ... 177
`safedb_verbose` ... 69, 177
`safedbdel` ... 177
`safedbget` ... 177
`safedbmap` ... 177
`safedbput` ... 177
scope of a `catch` ... 102
scope of exception handlers ... 102
scope of visibility ... 109
scope of visibility, functions ... 79
scope of visibility, variables ... 65
script file checking ... 33
`script-file` ... 220
scripting, parsing command line arguments ... 37
`sed` ... 128
selecting syslog facility ... 44
`send_dsn` ... 193
`send_mail` ... 192
`send_message` ... 193
`send_text` ... 192
sender address verification, described ... 6
sender address verification, limitations ... 7
Sender Policy Framework ... 8
Sender Policy Framework, defined ... 195
sender verification, writing tests ... 104
`sender-socket, --sender-socket, mtasim` option, summary ... 270
sending rate, explained ... 8
Sendmail macros in '`end`' ... 76
`Sendmail` macros, exporting ... 243
Sendmail macros, referencing ... 61
sendmail macros, setting from the command line ... 34
Sergey Afonin ... 3
`server` ... 223, 250
server, callout ... 19
session cleanup handler ... 76
session startup handler ... 76
`set` ... 66, 94
`set, --set` calloutd option, summary ... 254
`set_from` ... 147
`setbuf(number` ... 142
`setup-file` ... 117, 222
`setvar` ... 221
shadowing, constant–constant ... 92
shadowing, defined ... 89
shadowing, variable ... 90
shadowing, variable–constant ... 91
`show-config-options, --show-config-options` calloutd option, summary ... 254
`show-defaults, --show-defaults` mailfromd option, introduced ... 30
`show-defaults, --show-defaults` mailfromd option, summary ... 239
`SHUT_RD` ... 139
`SHUT_RDWR` ... 140
`SHUT_WR` ... 139
`shutdown` ... 77, 139
shutdown, special handler ... 77
`sid` ... 256
Sieve ... 183
`sieve` ... 183
`sieve.mfl` ... 183
SIGHUP ... 240
SIGINT ... 240
signals ... 240
SIGQUIT ... 240
SIGTERM ... 240
Simon Christian ... 3
Simon Kelley ... 11
`single-process` ... 224, 251
`single-process, --single-process` calloutd option, summary ... 252
`single-process, --single-process` mailfromd option, summary ... 237
`site-start.el` ... 215
`sleep` ... 182
`smtp-starttls` ... 228
`smtp-timeout` ... 226
smtpd_milters, postfix configuration ... 246
socket map ... 206
`sockmap.mfl` ... 206
`sockmap_lookup` ... 207
`sockmap_single_lookup` ... 207
soft SMTP timeout ... 19
SoftFail, SPF result code ... 196
`source-info, --source-info` calloutd option, summary ... 253
`source-info, --source-info` mailfromd option, summary ... 238
`source-ip` ... 221, 250
`source-ip, --source-ip` calloutd option, summary ... 252, 253
`source-ip, --source-ip` mailfromd option, summary ... 235
SpamAssassin ... 186
`spamc` ... 186
spamd ... 186
`spawn` ... 139
`spf.mfl` ... 197
`spf_check_host` ... 197
`spf_explanation` ... 198
`spf_explanation_prefix` ... 199
`spf_mechanism` ... 199
`spf_status_string` ... 198
`spf_test_record` ... 198
SPF ... 8
SPF, checking host record ... 195
SPF, defined ... 195
`sprintf` ... 133

`sq` ... 128
`ssl-ca-file` ... 229
`ssl-certificate-file` ... 229
`ssl-key-file` ... 229
`ssl-priorities` ... 228
stack growth policy ... 55
stack traces, reading ... 51
'`Stack underflow`', runtime error ... 48
`stack-trace` ... 225
`stack-trace, --stack-trace` mailfromd option, explained ... 51
`stack-trace, --stack-trace` mailfromd option, summary ... 237
`stack_trace` ... 211
`stack_trace` function, introduced ... 51
`stack_trace_format` ... 211
`stack_trace_top` ... 212
`stacksize` ... 55
stage handler arguments ... 75
stage handler, defined ... 70
standalone catch ... 102
standard address verification ... 6
standard error, using for diagnostics output ... 43
standard verification with `poll` ... 107
`starttls` ... 255
STARTTLS in SMTP callout sessions ... 228
`startup` ... 77, 239
startup, special handler ... 77
state handler, declaring ... 15
`state-directory` ... 221, 250
`state-directory, --state-directory` calloutd option, summary ... 252
`state-directory, --state-directory` mailfromd option, summary ... 235
`state-directory, --state-directory` mfdbtool option, summary ... 260
`statedir, --statedir, mtasim` option, described ... 262
`statedir, --statedir, mtasim` option, summary ... 270
statements ... 92
statements, conditional ... 95
`static` ... 65, 79
`status.mfl` ... 99, 167
`status.mfl`, module ... 99
`stderr, --stderr` calloutd option, summary ... 253
`stderr, --stderr` mailfromd option, summary ... 239
`stdio, --stdio, mtasim` option, summary ... 270
`stdpoll` ... 163
`StreamMaxLength`, `clamd.conf` parameter ... 68, 188
`strftime` ... 181
strict address verification ... 7
strict verification with `poll` ... 107
`strictpoll` ... 163
`string_list_iterate` ... 119
`strip_domain_part` ... 133
strip_domain_part, definition of ... 84
`strip_domain_part.mfl` ... 133
`substr` ... 131
`substring` ... 132
supplementary groups ... 239
`switch` ... 95
`switch` statement ... 95
syntax check ... 33
`syntax-check, --syntax-check` mailfromd option, introduced ... 33
`syntax-check, --syntax-check` mailfromd option, summary ... 238
`syslog` ... 209
syslog facility, default ... 44
syslog facility, selecting ... 44
syslog tag ... 44
`syslog, --syslog` calloutd option, summary ... 253
`syslog, --syslog` mailfromd option, summary ... 239
syslog, asynchronous ... 43
syslog, default implementation ... 43
syslog, non-blocking ... 11, 43
syslog, using for diagnostics output ... 43
`syslog.mfl` ... 209
`system` ... 182
system-wide startup script ... 240

T

tbf database ... 32
`tbf_rate` ... 189
`tempfail` ... 92
tempfail action, defined ... 92
tempfail action, introduced ... 13
`tempfail` in '`end`' ... 76
`TEMPFAIL_ACTION` ... 77
`tempfile` ... 140
`test, --test` mailfromd option, introduced ... 34
`test, --test` mailfromd option, specifying handler name ... 35
`test, --test` mailfromd option, summary ... 233
Texinfo ... 5
`textdomain` ... 208
Thomas Lynch ... 3
`throw` ... 104
`time` ... 181
time formats, for `--time-format` option ... 283
Time Interval Specification ... 220
`time-format, --time-format` mfdbtool option, summary ... 260
`timeout` ... 256
timeout escalation ... 19
`tls` ... 228
TLS support ... 228
`tolower` ... 132
`toupper` ... 132
`tr` ... 126
trace file, `mtasim` ... 266
`trace, --trace` mailfromd option, introduced ... 44

`trace, --trace` mailfromd option, summary .. 238
`trace-actions` 225
`trace-file, --trace-file, mtasim` option, described 266
`trace-file, --trace-file, mtasim` option, summary 270
`trace-program` 225
`trace-program, --trace-program` mailfromd option, summary 238
`transcript` 226, 252
`transcript, --transcript` calloutd option, summary 253
`transcript, --transcript` mailfromd option, introduced 48
`transcript, --transcript` mailfromd option, output example 48
`transcript, --transcript` mailfromd option, summary 239
`try` statement 101
try–catch construct 101
trying several sender addresses 163
`type` 275
type casts, explicit 89
type casts, implicit 88

U

`u, -u, mtasim option, summary` 268
U, -U option, described 118
U, -U option, summary 236
U, `\U`, a `mtasim` command 264
`umask` 182
`uname` 182
`undefine, --undefine` mailfromd option, described 118
`undefine, --undefine` mailfromd option, summary 236
`unescape` 130
`unfold` 153
`unlink` 182
upgrading from 1.x to 2.x 300
upgrading from 2.x to 3.0.x 300
upgrading from 3.0.x to 3.1 299
upgrading from 3.1.x to 4.0 298
Upgrading from 4.0 to 4.1 298
Upgrading from 4.1 to 4.2 298
Upgrading from 4.2 to 4.3.x 297
Upgrading from 4.3.x to 4.4 297
Upgrading from 4.4 to 5.0 296
Upgrading from 5.0 to 5.1 295
Upgrading from 5.x to 6.0 294
Upgrading from 6.0 to 7.0 293
Upgrading from 7.0 to 8.0 292
Upgrading from 8.13 to 8.14 290
Upgrading from 8.14 to 8.15 288
Upgrading from 8.17 to 9.0 287
Upgrading from 8.2 to 8.3 291
Upgrading from 8.2 to 8.4 291
Upgrading from 8.5 to 8.6 291
Upgrading from 8.7 to 8.8 291
Upgrading from 9.1 to 9.2 287
`url` 275
URL, mailer 191
`usage, --usage` calloutd option, summary 254
`user` 229
user privileges 239
`user, --user` calloutd option, summary 252
`user, --user` mailfromd option, summary 235
`user, --user, mtasim` option, described 263
`user, --user, mtasim` option, summary 268

V

`v, -v, mtasim option, summary` 270
`valid_domain` 190
valid_domain, definition 84
`valid_domain.mfl` 190
`validuser` 190
vaptr 40
variable assignment 22, 66, 94
variable declaration 22
variable declaration, implicit 22
variable declarations 65
variable interpretation 59
variable lexical scope 65
variable number of arguments 80
variable shadowing 90
variable values, setting from the command line . 34
`variable, --variable` mailfromd option, introduced 34
`variable, --variable` mailfromd option, summary 235
variable, assigning a value 66
variable, precious 23
variables, accessing from `catch` 104
variables, automatic 82
variables, declaring 65
variables, defined 65
variables, introduced 22
variables, local 82
variables, precious 65
variables, predefined 66
variables, referencing 66
variadic function 80
`verbose, --verbose, mtasim` option, summary 270
verbosity level 45
`vercmp` 132
Verifying script syntax 215
`verp_extract_user` 133
`version, --version` calloutd option, summary 254
`void` 89, 115
void functions 81
`vrfy` 254
VRFY, SMTP statement 228

W

`W_OK` ... 181
`when` keyword ... 106
`while` ... 96
while loop ... 97
whitelisting ... 28
`WITH_GEOIP2` ... 173, 175
`write` ... 140
`write-timeout` ... 275
`write_body` ... 140

X

x, `transform` flag ... 129
`X, -X, mtasim option, summary` ... 269
`X_OK` ... 181
`XML` ... 147
`xref, --xref` mailfromd option, summary ... 239

Z

Zeus Panchenko ... 3