The Maclean IMAP Server Engine

Developer’s Guide

April 2013
Version 3.11
[Manual Version 12]

Pete Maclean
PO Box 520
Nellysford, Virginia 22958, USA

Voice: +1 434/361-1678

Internet: pete@maclean.com
# Table of Contents

Introduction to MISE .................................................................5
What you need to know about IMAP ........................................7
Cryptographic software ..........................................................7
Upgrading from previous versions ............................................8
   **Breaking Changes in 2012 Releases** .....................................8
   **Breaking Changes in 2013 Releases** .....................................8
MISE Implementation .............................................................9
   IMAP Implementation ........................................................9
RFC .........................................................................................9
   Architecture .....................................................................9
   M5 .................................................................................10
Extras: ...................................................................................10
   Listener ...........................................................................10
   Logging ..........................................................................10
   M5 .................................................................................11
   POP3 .............................................................................11
   SMTP ............................................................................11
Fitting MISE to an existing mailstore .......................................12
Store structure ........................................................................12
Folder names ..........................................................................12
Choosing a hierarchy character ...............................................12
Messages .............................................................................13
Designing a Mailstore ............................................................14
Delivery Manifest ...................................................................15
Open Source ...........................................................................15
MISE Source Modules ...........................................................16
MISE Source Modules ...........................................................16
Customization ........................................................................18
   custmise.h .................................................................18
   CUSTOM.C .....................................................................20
   MEM.C .........................................................................21
Building Your Server ............................................................23
   Organization ...................................................................23
   Compilation Options .......................................................23
   External Libraries ..........................................................23
Initiating IMAP Sessions .........................................................24
Subscription ...........................................................................27
Caching ...................................................................................28
   Basic Caching ..................................................................28
   Message Caching ...........................................................29
   To cache or not to cache? ..................................................30
SSL/TLS Support using OpenSSL ............................................31
Exptunged Messages .............................................................32
IMAP Extensions ...................................................................33
IDLE Command ....................................................................33
UIDPLUS ..............................................................................33
Login Referrals .....................................................................33
LOGINDISABLED ...............................................................34
ID Extension .........................................................................34
BINARY Extension ...............................................................34
UNSELECT command ...........................................................34
Namespaces ..........................................................................35
LIST-STATUS .......................................................................35
Extended SEARCH (ESEARCH) ...............................................36
**Introduction to MISE**

Welcome. This manual is intended to reveal most of what a developer needs to know about the Maclean IMAP Server Engine ("MISE"). It is supplemented by two other manuals, the MISE Internationalization Guide and the MBM Developer Guide (which documents the MBM API).

This document is somewhat lengthy because of the considerable wealth of options, extensions and customizations available with MISE. The core however is simple. Here is a picture of how MISE fits into a working IMAP server:

Integrating MISE into a server involves the following steps:

1). Assessing how your mailstore fits with the requirements of IMAP. Do you have to convert messages to RFC form? Does its foldering system fit the IMAP model? What IMAP message flags can be stored? What should you use as an IMAP hierarchy character? Does the store have something that will serve for message UIDs or can you add something for this?

2). Writing what we call an "MBM" (for Mailbase Manager). An MBM is the glue layer between MISE and a mailstore. Building an MBM means implementing a set of functions to fit MISE’s MBM API. These functions perform such actions as opening the mailstore, listing its folders, creating and deleting folders, enumerating messages, retrieving message data and manipulating message flags. There is a core set of about 30 functions that must be provided and several more that may be required to support optional capabilities. During initial development, these functions can be unit-tested off-line with utility programs that we provide.

3). Writing a few functions for customizing the IMAP server. These functions manage various elements of server operation that are not mailstore-related. The most important ones are for authenticating users. Others serve such purposes as feeding data about what is happening in the server to an operator’s display.

4). Hooking MISE to a listener, the component that listens for and accepts IMAP client connections. If you do not already have a listener, we can provide a simple one for you.

5). Choosing what IMAP extensions to support. In general, the more the better. Some extensions have essentially no cost while a few require significant resources to be added.

6). Selecting what MISE options to exploit. There is caching to consider and thread pooling. The use of OpenSSL, while nominally optional, is really a requirement in most environments for security.
The time required to do all this varies depending on a large number of factors. For someone experienced with the process, it can be as little as a week. More typically it takes about one month of work. And please bear in mind that you do not have to do this yourself; you can contract with us to do the whole job for you.
What you need to know about IMAP

MISE makes implementing IMAP4 easy because you don’t have to write any protocol code or concern yourself with sockets programming. In addition, it offers you several goodies that are not strictly IMAP-related such as a fine logging system. Using MISE does not, however, absolve you from knowing and understanding a certain few IMAP requirements. You should be passingly familiar with the IMAP4 specification (RFC 3501) and we can recommend other, less formal, texts on the protocol. Most importantly, you should be very careful to ensure that you understand and follow IMAP requirements concerning:

- The assignment of unique identifiers to messages and mailboxes (UIDs and UIDVALIDITYs).
- Rules pertaining to the expunging of deleted messages.
- Interpretation of the reference and mailbox arguments to the LIST command.

Cryptographic software

MISE is normally built with support for CRAM-MD5 and DIGEST-MD5 authentication and can optionally be built with support for SSL and NTLM (which involves DES and MD4 encryption). Please remember that export/import and/or use of strong cryptography software, providing cryptography hooks or even just communicating technical details about cryptography software is illegal in some parts of the world.

The only cryptographic software provided with MISE is comparatively weak stuff, 56-bit DES and MD5. Weak though it may be, we are not completely sure that even this is legally acceptable everywhere. If you choose to use stronger cryptographic software with MISE, such as OpenSSL, we leave you to obtain that independently.

The use of certain cryptographic algorithms is restricted by patent in some parts of the world. We are as sure as we can be that this does not apply to any used by MISE but, even here, we advise you to confirm that for yourself.

When you import the MISE package to your country, re-distribute it from there or even just email technical suggestions or even source patches to us or other people, you are strongly advised to pay close attention to any export/import and/or use laws that apply to you. The authors of MISE are not liable for any violations you make here. So be careful; it is your responsibility.
Upgrading from previous versions

If you are starting out with MISE then you can skip this section which is provided for existing customers who are updating their MISE software.

**Breaking Changes in 2012 Releases**

We make changes that will break existing customers’ code that works with MISE only when preparing major new releases and only for very compelling reasons. We prefer to avoid such changes but believe that occasionally making one or two is the right thing to do.

Due to the need by some customers to select at run time which authentications methods are to be allowed, the calling sequence and effects of the `CustomAuthMethods` function have been changed. If you are upgrading to version 3.1 or higher from an older version, this is very likely to affect you. A parallel change is that the `MISE_LOGIN_DISABLED` compilation option has been removed. Please review the documentation in the `custom.tc` module for full details of these changes.

In order to support IPv6, the MISE function that you call to start a server session (IMAP4Init) now requires that you pass an IPv6-compatible `sockaddr_in` structure. If your listener accepts an IPv4 connection, it should reform it to look like an IPv6 connection.

Other changes that will affect users upgrading from earlier versions are highlighted in this manual.

In addition there are breaking changes described in the MBM API manual. We have added some new MBM functions to support new options and extensions and have renamed some MISE functions that MBM code calls for consistency.

**Breaking Changes in 2013 Releases**

In release 4, we have made a change to one MBM function (MBMSetMessageFlags).
MISE Implementation
This section discusses how MISE is implemented.

IMAP Implementation
MISE implements IMAP4 revision 1 as documented in RFC 3501 and related documents (RFC 2683, etc.). Since MISE was written only when this version of IMAP was current, it has never supported any earlier versions. It also supports these IMAP extensions:

<table>
<thead>
<tr>
<th>RFC</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3501</td>
<td>IMAP Version 4rev1</td>
<td>Base spec for the current version of IMAP.</td>
</tr>
<tr>
<td>2177</td>
<td>IDLE command</td>
<td>Extension that facilitates notifications of new mail arriving and other changes to the current mailbox.</td>
</tr>
<tr>
<td>2221</td>
<td>Login Referrals</td>
<td>Advises client when a user’s account has moved to a different server</td>
</tr>
<tr>
<td>2342</td>
<td>NAMESPACE</td>
<td>Support for multiple message stores.</td>
</tr>
<tr>
<td>2359</td>
<td>UIDPLUS extension</td>
<td>Enhancements for better synchronization.</td>
</tr>
<tr>
<td>2595</td>
<td>LOGINDISABLE</td>
<td>Option to disallow LOGIN command (in favour of secure authentication methods).</td>
</tr>
<tr>
<td>2791</td>
<td>ID extension</td>
<td>Allows client and server to identify each other.</td>
</tr>
<tr>
<td>3516</td>
<td>BINARY extension</td>
<td>Allows client to send and receive MIME parts in binary.</td>
</tr>
<tr>
<td>3691</td>
<td>UNSELECT command</td>
<td>Extension for better control over mailboxes.</td>
</tr>
<tr>
<td>4551</td>
<td>Conditional STORE</td>
<td>Extension for coordinating updates.</td>
</tr>
<tr>
<td>4731</td>
<td>ESEARCH</td>
<td>Extended SEARCH responses.</td>
</tr>
<tr>
<td>5161</td>
<td>ENABLE</td>
<td>Client enabling of relevant extensions.</td>
</tr>
<tr>
<td>5162</td>
<td>QRESYNC</td>
<td>Quick Mailbox Resynchronization.</td>
</tr>
<tr>
<td>5255</td>
<td>LANGUAGE</td>
<td>LANGUAGE command.</td>
</tr>
<tr>
<td>5819</td>
<td>LIST-STATUS</td>
<td>STATUS responses in extended LIST command.</td>
</tr>
</tbody>
</table>

Architecture
MISE operates as a layer between a listener that accepts connections on a suitable port, normally the standard IMAP ports (143 and/or 993), and what we call a MailBase Manager ("MBM") that serves as an interface to a particular mail store. As noted above, the MBM API is documented separately.

MISE is written in pure C. In some areas the code is very generic, designed for portability; in other areas, we have sections optimized for Windows and Linux. We build MISE using Microsoft Visual Studio on Windows and with gcc on Linux. If you use a different compiler, we will verify its compatibility with that compiler for you.

We supply MISE in the form of C source code with the intention that you compile and link it into your own software. MISE could alternatively be built as an independent DLL or shared object library. Similarly, the MBM API is designed with the intention that your MBM implementation be statically linked in as well.
M5
MISE makes heavy use of a library called M5 for interpreting Internet messages. M5 is used to parse RFC 2822 headers and MIME directives. M5 is supplied as C++ source code but is essentially C-like in using no features of C++ such as STL or exceptions.

Extras
We can offer MISE licensees code for certain other purposes related to IMAP and email serving that may be useful in some cases. Typically we are happy to provide such code to MISE licensees, when requested, at little or no extra charge. This generosity reflects the fact that most of these extra bits of code are not of quite the same quality and completeness as MISE itself. Examples include:

Listener
Many MISE users, typically those adding IMAP capability to an existing email server, already have their own code for listening for incoming connections and can extend that for IMAP. For those who are building a server from scratch, we include an optional barebones listener module (server.c). Currently this is for Windows only.

Logging
MISE incorporates a logging system that can be included or excluded based on a build parameter (MISE_LOGGING). When you build with logging included, you can use the logging system as is, hook it into your own logging system if you have one, or use some combination of the two.

Our logging system, which you are most welcome to use, writes a journal file chronicling significant events and errors. Here is a fragment from such a file:

```
10-22-04 08:46:20 (I) 0AA8/Maclean IMAP Server 1.65 starting
10-22-04 08:46:20 (I) 0874/IMAP4 listener thread starting on port 143
10-22-04 08:46:37 (I) 0874/IPListen: IMAP4 connection received from 127.0.0.1
10-22-04 08:46:37 (I) 0544/Maclean IMAP4 Session starting
10-22-04 08:46:42 (I) 0544/User "pete" authenticated
10-22-04 08:46:43 (I) 0544-Mailbox directory: "D:\Mail\MES\pete.mbx"
10-22-04 08:46:48 (I) 0544/Selected: Inbox
10-22-04 08:47:34 (I) 0544/IMAP4 listener thread terminating normally
10-22-04 08:47:38 (I) 0AA8/Maclean IMAP Server terminating
```

Each entry consists of one line of text starting with a timestamp. The time is followed by a tag, a single letter in parentheses, that indicates the type of entry. Standard tags are:

```
I    informational
E    error
W    warning
D    debugging
```

and you can add your own. Next comes the thread id of the thread that issued the log entry. Since entries for all sessions running in the server are written to the same log, this helps you to see which entries pertain to which IMAP sessions. Finally comes the text that explains the event or error being recorded.

The logger includes an option to prune the log file every time it reaches a certain size. Other, more exotic options, are documented in the logger's source, logger.c and Logger.h.

A later section in this guide describes how to programmatically hook into this logging system.
M5
M5, as mentioned above, is our message-parsing library. If you would like to use this for other applications we can supply documentation on its API.

POP3
We have fairly complete code for building POP3 servers and clients.

SMTP
We have some functional but very limited SMTP server code.
Fitting MISE to an existing mailstore

This section presents some ideas for those who are using MISE to build an IMAP server onto an existing mailstore. This is what most of our customers have done. The following section focuses on designing a new mailstore for use with MISE.

Most of the material here is dictated by IMAP, not by MISE.

Store structure

The usual structure for a mailstore is a tree of named folders. In formal IMAP terminology, folders are called mailboxes but we stick with ‘folders’ as that is the more commonly used term these days.

IMAP does not consider the root of the tree to be a folder. It imposes no limits on folder depth, on the number of folders or on the length of folder names or paths. The only limit on the number of messages that a folder may contain is that it must fit in a 32-bit unsigned integer. Most mailstores allow folders to contain both messages and subfolders but IMAP and MISE can handle a store in which each node must be either a message container or a folder container (meaning that messages can be stored only in leaf nodes).

IMAP is flexible enough to also work with “flat” mailstores, that is stores that contain folders only at a single level.

Folder names

IMAP fully supports Unicode folder names. It even allows names to contain non-graphic characters (meaning essentially control characters). However many IMAP clients will have serious problems with non-graphic characters in names so we recommend that they be disallowed.

Choosing a hierarchy character

IMAP requires that one character be reserved as the “hierarchy character” for separating elements in folder paths. This is used like the backslash in Windows pathnames or the forward slash in UNIX pathnames except that IMAP folder paths do not start with the character. For example, in “folder/subfolder”, slash is the hierarchy character. Most IMAP clients display a folder tree but some may display a list of paths with the hierarchy separator included.

The hierarchy character is chosen by the IMAP server. In principle, it can be different for different namespaces but, in MISE, it must be fixed at build time and the same for all. IMAP specs stipulate that the hierarchy character must be a 7-bit ASCII code. If your mailstore has a hierarchical folder structure then you should, if possible, choose a character that is not valid in folder names. Experience shows that it is best to use slash, backslash or dot/period. If your mailstore has a “flat” folder structure then define this character as 0.

Choosing a hierarchy character gets tricky if you cannot use one of the usual choices (‘/’, ‘\’ or ‘.’). Actually, MISE will work okay if you choose a character that can appear in folder names – but folders with names that contain the character will then be invisible, that is not exposed to the IMAP client. (In some cases, this can be handy to hide folders from IMAP access but generally it is undesirable.) If your folder names can contain arbitrary characters (as is the case for example with PSTs and Exchange Mailboxes) then we suggest using a control character as the hierarchy separator. This assumes of course that users will typically not put control characters in folder names. Since most IMAP clients never expose the hierarchy character to a user, this may be satisfactory. ASCII code 127 (Del/Rubout) can be a tempting choice but proves to be a bad one because it causes problems for Thunderbird. The best control character that we have found for the purpose is ASCII code 31.
**Messages**

IMAP supports access only to email messages and not, for example, contacts or appointments (unless these are disguised as emails – which can be done). Furthermore messages must be immutable in their content; only flags associated with messages may be changed. This means that if users have the ability to alter any part of the content of a message that would be exposed via IMAP then, whenever a message is so altered, it must be represented to MISE and IMAP as a new message. For example, if a user has a “Drafts” folder in which messages are composed incrementally then, on each update, the earlier version must be effectively deleted and the new version made to look like a new message in the folder. Effectively this means giving the message a new unique ID (UID). If users are permitted to change the content of messages after they have been sent or received then the same treatment must be adhered to.
Designing a Mailstore

This section presents some advice for those who are developing a mailstore, or expanding an existing one, for use with MISE. This would apply to many people who already have a POP/SMTP email server and are adding IMAP capability to it. If you are retrofitting MISE to an existing mailstore then you should skip this section and read the next one instead.

IMAP itself does not demand this, but we strongly recommend that you implement a mailstore consisting of hierarchical folders like that provided by most popular mail clients. That is what the market expects and what IMAP clients are oriented towards; taking any other approach would be foolhardy unless you are targeting a very specific niche where a different organization (such as a “flat” store) is appropriate.

Things you should consider carefully are:

- What characters to allow in folder names. Almost certainly only printable characters should be accepted. Decide which character you will use as the IMAP hierarchy character; you will probably want to prohibit the use of that in folder names. A more difficult question may be whether or not to allow non-ASCII characters. IMAP is fully capable of handling Unicode folder names, as is MISE; some IMAP clients however do a poor job of this.

- How long should a folder name be. It is hard to give guidance on this. You will probably find implementation easier if you do impose a fixed limit. A good minimum would the always popular 32 characters and up to 256 may prove popular.

- How long a folder path should you allow. This limit may be imposed either by placing a ceiling on the number of characters in a path (say 256) or by setting a maximum depth to the folder tree (say 6 or 8).

- Should message containers be distinct from folder containers. We strongly recommend making folders homogeneous, that is capable of containing both messages and subfolders. IMAP is perfectly capable of handling folders either way but some clients are less than perfect in dealing with non-homogeneous folders.

- In what form should messages be stored. For IMAP purposes, it is best if messages are stored just as they come off the wire, i.e. in what is often called SMTP or EML format. If they are stored in another form then your MBM will need to convert them at a heavy rate. If your requirements dictate that messages be stored in another form, you might consider storing them in off-the-wire format as well. (The latest versions of Microsoft Exchange do this.)

If you have other troublesome questions about designing your mailstore, please discuss them with us.
Delivery Manifest

When you license MISE you will be supplied with four packages of source files. These comprise:

- The MISE sources which are further described in the next section. These are the sources for the core of the IMAP Server Engine.

- The M5 sources. These can be used to build our RFC 5322 message-parsing engine called M5.

- A collection of various logging and utility functions that are used by both of the above. These constitute the “Maclean Utility Library” and are supplied as a mix of C and C++ sources.

- Sample MBM sources. These are intended to serve as templates for building your own MBM. The primary sample works with a proprietary mailstore of our own called MACS. We can provide additional information on MACS and its own API if that would help you.

We normally do not supply project or make files but will gladly assist you in creating them.

Open Source

The code contains 3 open-source modules:

- acpmd5.c for calculating MD5 hashes
- des.c an implementation of the DES algorithm
- digestmd5.c for calculating MD5 digests

We believe that none of these carries any encumbrances but you should of course satisfy yourself of that. The MD5 module is public-domain code. The Digest MD5 code is lifted straight from RFC 2617. The status of the DES code is not quite so clear but it carries no copyright notice.

The MD5 modules are used in both the Linux and Windows versions. The DES module is delivered with the Linux package but is not actually used in the Linux build (since it serves NTLM authentication which normally is used only in Windows).
MISE Source Modules

This section describes the core source modules that comprise MISE. The .C files that comprise the IMAP server proper and are required are:

- `append.c` - APPEND command handling
- `auth.c` - authentication functions (except NTLM)
- `backdoor.c` - secret pseudo-IMAP commands for technical support and debugging
- `body.c` - FETCH BODY commands
- `command.c` - command pre-processor
- `copy.c` - COPY command handling
- `custom.tc` - template customization module
- `des.c` - encryption module
- `expunge.c` - EXPUNGE command handler
- `fetch.c` - basic FETCH command handling
- `imaps.c` - core IMAP server
- `imaputil.c` - miscellaneous utility functions
- `init.c` - initialization function
- `ipsupp.c` - general (system-independent) IP support
- `language.c` - language support
- `list.c` - LIST command handling
- `mbmutil.c` - utility functions for MBMs
- `mem.c` - memory-allocation functions (replaceable)
- `miserr.c` - error-handling code
- `receive.c` - functions for receiving IMAP data
- `search.c` - SEARCH command handling
- `select.c` - SELECT command handling
- `send.c` - functions for sending IMAP responses
- `sp.c` - core IMAP command processor
- `status.c` - STATUS command handling
- `store.c` - STORE command handling
- `subs.c` - SUBS command handling
- `transcribe.c` - functions for transcribing sessions
- `update.c` - handles updates to the selected folder
- `utf7.c` - modified-UTF7 functions

As you may notice, there are several modules that handle specific IMAP commands (FETCH, STORE, SUBS, etc.). Handlers for other IMAP commands, those that require little code, are included within the module `sp.c`. An additional `server.c` module is provided for those Windows users who are not integrating MISE into an existing server but need code to listen for connections. Modules highlighted above are new in 2012 MISE releases.

These C files are required for Linux builds only:

- `iplinux.c` - Linux-specific IP support
- `miselinux.c` - Linux-specific functions

And these are used for Windows builds only:

- `ipwin.c` - Windows-specific IP support
- `misewin.c` - Windows-specific functions
- `timefix.c` - time-related utility functions

The following C files are core parts of MISE but are required only when certain build options are selected:

- `async.c` - supports asynchronous sessions (used when thread pooling is enabled)
cache.c  supports caching (used when caching is enabled)
idle.c   supports the IDLE command (used when IDLE enabled)
ipasync.c supports asynchronous i/o (used when thread pooling is enabled)
ntlm.c  code for NTLM authentication (used when such is enabled)
pool.c  manages a thread pool (used when thread pooling is enabled)
tls.c    interface functions to the OpenSSL library

Header files include:

- MISE.h prototypes for externally callable functions
- MISE_Resources.h header for internationalization resources
- imap.h basic IMAP definitions
- ipsupp.h header file for socket i/o support (goes with IPWIN.C)
- Ascii.h ASCII character definitions
- custmise.h customization definitions (see next section)
- Filer.h header for a file-i/o library
- Logger.h definitions for the built-in logging system
- Maclean.h header for MACLEAN.LIB a utility library
- mbmfunc.h prototypes for MBM functions (see MBM document)
- mbmstruc.h definitions of MBM structures (see MBM document)

plus header files associated with each .c file.
Customization

This section discusses how to customize MISE. You can customize MISE in two ways: to configure it for your needs and to extend it. Here we discuss only configuration.

Before building your server, you must copy and edit three template header files and edit one C source file. One template header is distributed as custmise.th. Please copy this custmise.h and then edit the first one according to the directions given below.

Copy custom.tc to custom.c and edit it as described below.

You can also add customized “backdoor” commands to your server as described in a later section titled “Backdoor Commands”.

custmise.h

MISE compilation is affected by the following symbols which are #defined in the MISE customization header, (the cleverly named) custmise.h. This is the one header file that you have complete control over.

MISE_AUTH_NTLM #define this and include ntlm.c in your build to include support for NTLM authentication. Please read the section on NTLM authentication in this manual before doing so.

MISE_AUTH_NTLM_LM_ONLY (see the section on NTLM authentication).

MISE_BACKDOOR_ACCOUNT #define this to include support for a set of backdoor commands. For full details about this, see the section below titled “Backdoor Commands”.

MISE_CACHING #define this to include support for caching. MISE will cache IMAP BODYSTRUCTURE and ENVELOPE information. Generally we recommend caching. The main reason to disable it would be to avoid the cost in disk space that it takes. MISE’s caching is discussed in more detail below.

MISE_FOLDER_PATH_SIZE this must be #defined as the size of the longest folder path supported by your mailstore. By “folder path” we mean an IMAP path such as “folder/subfolder/subsubfolder”.

MISE_HIERARCHY_CHAR this establishes the character that is used to separate folder names in folder paths. Please refer to the section titled “Choosing a hierarchy character” below for guidelines on selecting a suitable character. If you leave this symbol undefined then MISE will use the forward slash (‘/’) by default. If your mailstore has a “flat” folder structure then #define this symbol as 0. Here are some examples:

```
#undef MISE_HIERARCHY_CHAR  // (use default)
#define MISE_HIERARCHY_CHAR 0x2E  // ASCII dot
#define MISE_HIERARCHY_CHAR 0  // (no hierarchy)
```

MISE_ID_RESPONSE can be defined if you wish the IMAP server to send a substantive response to an IMAP ID command. If you omit such a definition then a NIL response is sent.

MISE_IDLE #define this to include support for the IDLE command. For more information on what this involves see the section titled IDLE Command below.

MISE_LANGUAGE_COMMAND Defining this symbol gets MISE built with support for the LANGUAGE command, an extension for internationalization described in RFC 5255. See the later section
titled “Internationalization” for more. To comply with the RFC, when the LANGUAGE extension is enabled, the NAMESPACE extension must also be enabled.

MISE_LOGGING #define this as 0 if you wish to suppress logging and 1 to include logging. (We may add more options later.) For more information on what this involves see the section titled Logging below.

MISE_LOGIN_DISABLED Support for this option was removed in release 3.1. Please read the updated documentation on the CustomAuthMethods function.

MISE_LOGIN_REFERRALS This symbol should be #defined only if your server needs to provide Login Referrals (as per RFC 2221). Such referrals may be given either on a blanket basis at connection time or on a per-user basis at login time.

MISE_MAX_IN_MEMORY_BINARY_PART_SIZE When an IMAP FETCH BINARY command is used to retrieve a specified body part of a message in binary form, the encoded form of the body part is normally decoded to a temporary file, the contents of which are then sent to the client. You can have MISE decode such data directly to memory for body parts of up to a certain maximum size by setting this symbol. If you do not #define it, a default maximum value will be used. If you #define it as equal to zero then decoding to memory will not be done at all. For example:

// Use memory for holding body parts up to 65,536 bytes in size
#define MISE_MAX_IN_MEMORY_BINARY_PART_SIZE 65536

MISE_MESSAGE_CACHING This is a sub-option to MISE_CACHING. It tells MISE to cache whole messages in addition to summary information. This option is described in more detail below in the section on caching.

MISE_NAMESPACE Define this symbol to include support for the IMAP NAMESPACE extension (RFC 2342).

MISE_NOOP_DUPLICATE_SUBSCRIBES IMAP servers are of two minds about how to handle duplicate SUBSCRIBES, that is how to respond to a SUBSCRIBE command for a folder that is already on the user’s list of subscribed folders. The IMAP spec indicates that a server should send a NO response and that is what MISE does by default. (This holds as of November 2002; prior versions did the opposite.) Most servers send an OK response, treating the redundant subscription as a no-op. To select this latter behavior you must now #define this symbol.

MISE_PASSWORD_SIZE this must be #defined as the size of the longest password that your server supports. MISE needs this for implementing the IMAP AUTHENTICATE command.

MISE_THREAD_PRIORITY by default, MISE runs server threads (each session is a thread) at default priority. Most of our customers have found this to be the best choice. Should you wish to have IMAP threads run at a higher (or lower?) priority then you can #define this symbol to equal the priority level desired. For its effect, see the code in the module IMAPS.C. (Currently supported for Windows only.)

MISE_SSL This symbol is relevant only if you are using the MISE server module (server.c). #define it to enable support for SSL (Secure Sockets Layer) connections to IMAP via port 993. By default, the server accepts only non-secure connections via port 143. See also MISE_TLS below which must also be #defined when MISE_SSL is #defined.

MISE_SUBSCRIBE_NOOP This controls how MISE manages IMAP subscription. By default, MISE handles subscriptions internally as described in a later section. If this symbol is #defined, then MISE treats subscription as a null operation. In other words, SUBSCRIBE, UNSUBSCRIBE and LSUB commands become noops that draw a positive (OK) response but have no effect.
This provides a prefix used for naming temporary files used internally by MISE. If you do not provide a prefix then the default of "MISE" is used.

Tells MISE where to put temporary files. If defined, this should be equated to either 0 or 1. If undefined or set to 0, MISE uses the system’s temporary directory. If set to 1, it uses the “working directory” (i.e. the one whose path is passed into IMAP4Init).

#define this to build support for UTF-16 filenames in Windows. By default, only multibyte filenames are supported but we recommend supporting Unicode filenames. The use of UTF-16 for file names and paths applies throughout MISE including arguments to MBM functions.

#define this only if your server should support Unicode folder names. This should, of course, be selected only if your MBM and mailstore also support Unicode folder names.

This must be #defined as the size of the longest username that your server supports.

In the MISE sources, you may come across preprocessor symbols of the form MISE_xxx that are not included in the list above. Any such symbols are used for experimental, diagnostic or proprietary purposes and should not be #defined for your build.

There are two symbols that, when needed, should be #defined in your project settings or makefile (and NOT in custmise.h). These are:

#define this to enable support for SSL/TLS. Details of SSL support are given in a section devoted to that topic. When building with this option, in order to comply with RFC 2595, LOGIN should be disabled as well. MISE does not force this compliance in order to facilitate debugging but, for security, it is crucial.

controls the thread-pooling option discussed in a section below. This is available for Windows builds only.

This module contains all functions (other than backdoor functions) that need to be customized. Most of them relate to either account configuration or status display. Full instructions for customizing are included in the template CUSTOM.TC that is distributed with MISE and are not repeated here. Here is a list of the functions:

Called when a new IMAP session has just started. This gives the server the opportunity to display status information about the session.

Called when a new IMAP session has been created and before authentication is done. This allows you to specify which authentication methods are to be allowed for the session. Typically this would be done based on the IP address of the client (which is passed in). Users in the local subnet might, for example, be allowed to use weaker authentication methods than those on the outside.

This need be included only when MISE_BACKDOOR_ACCOUNT is defined. It validates a password for the backdoor account.

Determines whether or not a connection transcript should be created and optionally supplies a filename for it.
CustomGetAccountInfo: Retrieves configuration information for an account given a username. In particular, this information includes the password. IMAP requires the password in order to implement certain authentication schemes. The information is passed back in a structure, MISE_AccountInfo, described later in this document.

CustomGetServerParameters: Uses a structure to retrieve various values used as global parameters for the IMAP server. This custom function was added in version 2 of MISE; in some later revision, we plan to have it subsume CustomInactivityTimeout.

This function is passed a structure defined in IMAP.H and called MISE_ServerParameters. Currently this contains only a single parameter, the maximum number of concurrent sessions that the server will allow. If this is set to zero, no maximum session count is enforced and new sessions will be allowed until some resource lack dictates otherwise. When a non-zero maximum is set and that maximum is reached, new clients connecting will be refused with the message “Server cannot accept any more sessions at this time. Please try later.”

CustomThreadPoolSize: (To be documented.)

CustomInactivityTimeout: Gives you the chance to set the IMAP server’s inactivity timeout. By default, the server times out and disconnects a session after it has been inactive for 30 minutes. To set a different timeout, return the number of minutes to be used. If you return zero then the default is used. This function is called once upon server initialization; it is not possible to set the timeout on a per-session basis.

Please note that this inactivity timeout affects only sessions that are idle in the general sense of the word – that is, sessions in which the server is waiting for a command to arrive. It does not affect operation of the IDLE command, the timeout for which is always 30 minutes. If you consider 30 minutes to be an unacceptably long idle period then the thing to do is to not support the IDLE command which is, after all, an optional extension (see MISE_IDLE).

CustomLoginFailed: Gives you an opportunity to deal with a login that had an authentication failure. You may like to count successive failures and disable an account after a certain number of attempts.

CustomPreAuthorize: Gives you an opportunity to preauthorize a user. This capability is provided to support IMAP’s PREQUEST option by which a user may be automatically recognized (e.g. by virtue of the IP address that he connects from) and logged in without going through an explicit login procedure. This is rarely used but included in order to be 100% compliant with the IMAP spec.

CustomRemoveSession: Called when an IMAP session has just ended. This gives the server the opportunity to remove status information it has displayed about the session.

CustomServerGreeting: Indicates if the IMAP server is enabled or not and allows you to provide a greeting that is sent to the client.

CustomUpdateFolder: Called whenever the user opens a new folder with the name of that folder. This function is provided so that servers can display or log the folder name.

CustomUpdateUsername: Called when a session is authenticated. This allows the server to display or log the user’s identity

CustomValidateAccount: Called to validate an account during a login or authentication.

MEM.C
All long-term memory allocation in MISE is done using functions provided in the mem.c module. If you have reason to want to control memory allocation in some special way, you can replace our MEM.C with your
own. Our implementation consists of very simple wrappers around the C RTL functions malloc(), free() and realloc().

In a few cases, MISE functions use malloc() and free() directly for very small, short-term allocation needs.

**APPEND**

IMAP’s APPEND is the command that clients use to import messages into the server’s mailstore. When MISE receives an APPEND command, it saves the message data to a temporary file and calls the MBM’s MBMImportMessage function to store it away. Developers should pay some attention to what they should accept and what they should reject from an APPEND. While the payload of an APPEND is expected to be a message in RFC format, there is no guarantee that that is what actually arrives and MISE itself makes no check on the content. It would be a rare case and probably one that would occur only as the result of some rogue client but it could be garbage. It could also be a completely empty file, a document of some kind or even an executable. We recommend that developers take advantage of MISE’s message parser, M5 (documented separately), to validate incoming messages and reject any that do not pass muster.

When a valid message is received, it may also be appropriate to scan it for malware. In some environments this may happen automatically by the anti-virus software installed on the system. In others, developers may need to consider licensing a scanner solely and specifically for checking imported messages.
Building Your Server
This section discusses various aspects of compiling MISE and linking it into your server.

Organization
We recommend dividing the MISE-related source modules among three directories: MISE, M5 and Maclean. Each of these can then be compiled and agglomerated into a static library. These three libraries can then be linked with your MBM and listener to create your server application. Dependencies are as follows:

- Maclean is dependent only on C and system libraries
- M5 is dependent on Maclean
- MISE is dependent on both M5 and Maclean (and optionally OpenSSL)

Alternatively, you could place all source modules into one directory devoted to them and build a single static library that you then link into your server.

For Linux, we provide a makefile for building each library using gcc. You will need to edit these makefiles at least to change directory paths for your environment. We do not normally provide project files for Windows builds but will help you create them if you wish.

Compilation Options
The code makes considerable use of the C run-time library and must be compiled for and linked with a version of that library that supports multiple threads. We recommend specifying the option to force all chars to be treated as unsigned.

None of the code is designed for building in alternative multibyte and Unicode forms supported by Microsoft Visual C. In other words the _UNICODE symbol is ignored. We recommend building with Character Set = Not Set.

In Linux, the --pthread option must be included for the link phase.

External Libraries
In Windows, the MISE code must be linked with the following Windows import libraries:

    Advapi32.lib, Kernel32.lib, Rpcrt4.lib, User32.lib, Version.lib, WS2_32.lib

Note that versions earlier than 1.7 required WSOCK32.LIB rather than WS2_32.LIB. This means that, somewhere in the initialization code of your server, you should call WSASStartup with a first argument of MAKEWORD(2,2). Your server is also responsible for calling WSACleanup.

For Linux builds, if you get a link diagnostic about an undefined reference for clock_gettime, you will need either to explicitly link with librt.a or use the -lrt option. MISE also depends on the iconv library for character-set conversions. This is usually installed and linked by default.
**Initiating IMAP Sessions**

When your listener starts, it should call the function `IMAP4Init`. This function is located in the module `init.c` and prototyped in `mise.h`). It serves to perform a one-time initialization of the IMAP Server Engine. In Windows builds, the following parameters are required:

```c
HINSTANCE hAppInstance; // the instance handle of the application
const char *pszWorkingDir; // the full path of a working directory
MacleanLogger *pLogger; // the Server's logger object (optional)
```

In Linux builds the first of these is omitted.

The function returns TRUE if the initialization completes successfully. Currently the only reason for failure (and its returning FALSE) is if you pass an invalid working directory.

The working directory is critically important. The IMAP server uses it as a place to keep files listing users’ subscribed folders so it should be a fixed, dedicated directory.

If you use our logging system then the third parameter may carry a pointer to a `MacleanLogger` object. If you do not use our logging system then pass NULL.

Having called `IMAP4Init`, your server should end up calling `IMAP4Term` prior to terminating. This serves to release any and all resources held by the engine.

When your listener accepts an incoming connection for IMAP, it should launch a session by calling the MISE function `IMAP4StartServerThread`. This is located in the module `imaps.c` and prototyped in `mise.h`. The function takes the following parameters:

```c
SOCKET sockSession; // [in] the socket opened for this server session
struct sockaddr_in6 sinSession; // [in] the IP address of client
int bSSL, // [in] ? this is an SSL session
MacleanLogger *pLogger; // [in] the Server's log object (optional)
void *pvRef; // [in] listener’s reference value (optional)
pid_t pdwThreadId; // [out] id of server thread (or NULL)
```

The function launches a thread to service the session and then immediately returns. Its return value is a BOOL which is TRUE if the session was started successfully and FALSE otherwise. A FALSE return means that insufficient resources were available to launch the session: either there was a lack of allocatable memory or a new thread could not be created. When the function is successful, it stores the id of the new thread into the variable pointed to by the last parameter (unless that parameter is NULL).

The `pvRef` parameter can be used to pass a value that is stored in the IMAP session descriptor (ISD, described below) and which thus is available to Custom functions.
Backdoor Commands

MISE includes support for a special “backdoor” account. When a user logs into the IMAP server using this secret account, he or she can use a set of special pseudo-commands intended for troubleshooting. The intention is for this account to be used by technical support people only to help out customers when problems arise. The commands allow a technician to see what is going on “behind the scenes” in the server.

This backdoor feature is both optional and entirely modular. You can choose not to include it and you can replace our backdoor commands with your own.

The backdoor commands currently supported by our own BACKDOOR.C module are:

- **ACCOUNT**: retrieves information (excluding the password) about a server account
- **ACCOUNTS**: retrieves a list of account names
- **ALERT**: sends an IMAP alert message to a selected client
- **CX**: lists active IMAP sessions
- **DISCO**: forcibly terminates a given IMAP session
- **DNS**: retrieves information about Dynamic DNS state
- **FILE**: spills the contents of a given file in the server’s directory
- **LOG**: spills the contents of the server’s log file
- **ISD**: dumps the state of a given IMAP session
- **SETPW**: sets the password for a given server account
- **VERSION**: supplied version information
To include a backdoor account in your server, you must define MISE_BACKDOOR_ACCOUNT in your CUSTMISE.H file. This symbol should be set to a string that will serve as the username for the backdoor account. We suggest that you choose a username that includes a character that is not permitted in standard usernames. You must also include a IMAPCheckBackdoorPassword() function in your IMAPCNF.C module. The model for this function is:

```c
/* ************************************************************
** IMAPCheckBackdoorPassword: Checks a password sent by a client for
** access to the server's backdoor account.
**
** This function must return:
**   TRUE => the client should be allowed to login
**   FALSE => the login should be refused
**
** This function need be present only when MISE_BACKDOOR_ACCOUNT is
** defined.
**
*************************************************************************/

BOOL IMAPCheckBackdoorPassword(
    ISD *pISD,
    const char *pszLoginPassword)
{
    return IsThisPasswordValid(pszLoginPassword);
}
```

To include your own backdoor commands, you must define those commands in the module BACKDOOR.INC following the instructions therein and then write a processing function for each following our examples in BACKDOOR.C.
Subscription

This section describes how MISE handles IMAP subscription. Subscription is a feature of IMAP that allows a user to select a distinguished set of mailboxes/folders that he or she usually works with. Exactly how subscription works in practice depends on the particular IMAP client concerned. Many give the user two alternative views of the server’s folder tree: subscribed folders only and all folders. Having the more limited subscribed view can be very handy when the total number of folders is very large. Some IMAP clients, for example Eudora, do not support subscription at all. Subscription information is normally stored by the server thus giving users a consistent view from any client they happen to use – but, of course, that is only as consistent as clients are consistent which, in some cases, is not much!

Subscription is operated by three IMAP commands, SUBSCRIBE which adds a folder name to the user’s subscription list, UNSUBSCRIBE which removes a folder name, and LSUB which lists selected folder names.

By default, MISE handles subscription itself. It stores a list of subscribed folders in a simple text file in each user’s private directory. Two #defined symbols affect this: MISE_SUBSCRIBE_NOOP and MISE_NOOP_DUPLICATE_SUBSCRIBES. If MISE_SUBSCRIBE_NOOP is defined then MISE does not store any subscription information and treats all subscription-related commands as noops. Use of this is appropriate in some circumstances.

We plan to add more subscription options in future including one to allow the MBM to handle subscription and a way to allow/disallow subscription on a per-account basis.
**Caching**

MISE offers two levels of caching, Basic Caching and Message Caching.

**Basic Caching**

The IMAP protocol transfers two pieces of summarized information about messages. One is called the *BODYSTRUCTURE* and expresses the MIME structure of a message. The other, called *ENVELOPE*, contains a summary of the information in the message’s headers. In the absence of caching, MISE builds each of these structures whenever required from the raw text of a message. For large messages, doing this can be an expensive operation and one that can be worth not doing repeatedly. We have therefore implemented an optional caching system in MISE so that these structures need be generated only once.

The benefit of caching is performance while the cost is disk space. It is impossible to suggest in general the performance improvement that will result from caching; it depends on many factors including the particular IMAP clients used, patterns of usage and disk speed. Some further considerations about this are given below.

MISE’s caching system is enabled by defining the symbol `MISE_CACHING` (as noted in the configuration section above). Various other things govern how caching is actually done. The most critical of these is the `uidFolder` field in the `MBM_FolderStatus` structure (see the MBM documentation). This value is used only when caching is enabled. It provides a unique id that MISE uses to identify its cache files for the folder in question. Thus, full-scale use of caching depends on your being able to assign each of a user’s folders such an id; normally this would be a number incremented and assigned whenever a new folder is created. If you set this value to zero then caching will not be done for the folder. This arrangement provides great flexibility. You could, for example, implement caching only for each user’s Inbox by assigning a unique id for that folder and none other – and then the Inbox id can actually be fixed, say given a value of 1. Since, in many typical IMAP situations, users access their Inboxes every time they login and access other folders only occasionally, this may be a good compromise.

MISE creates its cache files in the user’s working subdirectory, that is a directory created by MISE named for the user’s account under its working directory. The cache files are named using the folder id and the extensions `.imapci` and `.imapcd`. It is perfectly safe for you to delete these files indiscriminately (say in an administrative utility for your server) as long as the server is not running at the time. When an account is removed, you *should* delete all the cache files associated with it – and that can be safely done while the server is running provided the account is fully disabled first and there are no active sessions for the account.

Since these cache files can end up consuming a large amount of space, we provide two mechanisms within MISE for keeping their growth in check. As implied above, you might want to provide external means of culling them as well. By calling the function `IMAP4ManageCache` you can establish two parameters for MISE’s management of cache files. The first argument is a number of days; if this is non-zero then MISE will delete cache files that have not been used within the last so many days. The second argument is a maximum size for each user’s cache in megabytes. If either argument is zero then no culling will be performed on the basis in question. The function returns void. If you choose to use this function (there is no requirement to) then we suggest you call it once following your call to `IMAP4Init`. Here is an example:

```c
IMAP4ManageCache(31, 24);
```

This would tell MISE to delete cache files that have not been referenced for a month and to limit each user’s cache to 24MB. It is not currently possible to set different quotas for different users but we could certainly add that if there is demand. Note that MISE performs cache culling only when a user logs out; during a session, a user can easily far exceed the established quota.

For culling by age, MISE uses a minimum of 4 days. That means if you call `IMAP4ManageCache` with a first argument of 1, 2 or 3, MISE will substitute 4. This serves to avoid anomalies that could be caused by such
things as changes to and from daylight savings times, and the low resolution of last-accessed dates stored by some file systems (as low as one day). Very large changes to the system clock (on the order of days or weeks) made while the server is running could still result in cache anomalies but we believe the chances of such are tiny enough that we can safely ignore them. It seems prudent anyway not to alter clocks on servers while they are in production.

You can #define two additional custom symbols to impose more control over how BODYSTRUCTURE and ENVELOPE caching are performed. MISE’s basic caching design makes use of a fixed-size buffer for each of these items. If a particular message’s data does not fit into this buffer then it is not cached but rebuilt every time it is needed. The philosophy behind this is to keep basic caching simple and efficient at the cost of eliminating certain unusually large messages from its benefit. You can specify a size for each of these two cache buffers using:

```c
#define MISE_BODYSTRUCTURE_CACHE_BUFFER_SIZE 1800
#define MISE_ENVELOPE_CACHE_BUFFER_SIZE 1500
```

If you do not #define these symbols then default sizes are used. Please inspect the sources should you wish to determine the current values for these defaults. To select sizes that work well for your server, we suggest starting with the defaults and then adjusting according to the number of messages that are excluded (which you can count from debug entries in the log file).

**Message Caching**

With version 1.74 of MISE, we added an optional secondary level of caching, message caching. This is enabled by #defining the preprocessor symbol MISE_MESSAGE_CACHING. You can control which messages are actually cached at both the folder level and the message level.

The idea behind message caching is to improve performance for servers with message stores that do not store messages in RFC 2822/MIME format and also servers that have to retrieve messages from remote stores. In the first case, a message need be converted from the store’s internal form into RFC 2822/MIME format just once and then subsequent demands for the message can be satisfied from the cache. For this reason, message caching should generally not be used in situations where messages are stored locally and do not have to be converted.

At the folder level, you can control message caching by setting the new bCacheMessages field of the MBM_FolderStatus structure when a folder is opened by your MBMSelectFolder function. If you set this field to TRUE then, when MISE needs a message that is not already cached, it will call your MBMCacheMessage function. Set it to FALSE and no message caching will be attempted for the folder. Thus you might, for example, choose to implement message caching only for Inboxes. When your MBMCacheMessage function is called, you are completely free to cache the message or not. See the MBM API document for details.

Each message that is cached is stored on its own file in the same directory as all the other cache files. Cached-message files are given names of the form:

```
FFFFFFFF-VVVVVVVV-MMMMMMMM.imapcm
```

where the main part of the filename consists of three 8-digit hex numbers separated by hyphens. $FFFFF$ is the UID of the folder, $VVVVV$ is the UIDVALIDITY of the folder and $MMMM$ is the UID of the message. You can delete these files at will as long as the IMAP server is not running.

Cached-message files are culled by age exactly like other cache files (see above). They are not currently included in the quota cull pending further consideration of how they can best be so included.
To cache or not to cache?

The choice of whether or not to use message caching should usually be a clear-cut one. The decision of whether or not to use basic caching may be more difficult and, in many cases, it is likely to seem less than worthwhile. So, let us consider the pros and cons of basic caching in some more depth.

The critical thing about basic caching is that the data being cached by the server is data that should, in the ideal situation, be cached by the IMAP client. So, in an ideal world, an IMAP server would never waste its resources caching the data we are talking about. Well, except for one thing: the server could generate those BODYSSTRUCTURE and ENVELOPE digests when new messages enter the store so that they are ready and waiting when the client comes along and asks for them. This would smooth load on the server (assuming the generation is done by a background thread) and improve response time to the client.

There are at least four cases where basic caching is worth some consideration. But there may well be situations where all four together do not create enough reason to do this caching.

The first is WebMail clients, that is web sites that act as IMAP-to-WebMail gateways. These were used more back when caching was added to MISE than they are today, so they probably add little weight. The second is that some clients cache data only for the user's Inbox (and perhaps other special folders) but not for folders in general. (We have not researched to determine how many clients fall into this category.) The third is that some people will use multiple IMAP clients. We do not have any statistics about this but would not be surprised if there are plenty who use two. Probably very few use more than two. The fourth is that you might use MISE to front something that is more like a public library (for example newsgroups) than a set of personal email accounts. This would almost certainly be the strongest motivation for basic caching.

If you are unsure about basic caching, we recommend that you start without it and then, if you feel a need later on, try it experimentally and see if it makes a difference.
SSL/TLS Support using OpenSSL

MISE includes support for SSL encryption using the OpenSSL library. If you wish to use MISE with another SSL library, please discuss that with us.

An IMAP client can create an encrypted session with a server that supports SSL in either of two ways. One is by connecting to a port assigned for the purpose, normally port 993. Using this method, the session is encrypted from the outset. The other way is to open a session that starts out unencrypted (typically by connecting to the standard IMAP port, 143) and then issuing the STARTTLS command which initiates a negotiation to switch into encryption. For full details, see RFC 2595. MISE supports these two options when you build it with the symbol MISE_TLS #defined. (There is no explicit way to build with one option and not the other although, of course, which port or ports your server listens on is up to you.)

For information about the OpenSSL project, please refer to www.openssl.org. From this Web site you can download the source code of the library. For Windows use, this builds as a pair of DLLs, ssleay32.dll and libeay32.dll or ssleay64.dll and libeay64.dll. For Linux, it is built by default as a pair of static libraries. You can alternatively build shared libraries by running the config utility with the SHARED option.

You can direct MISE to link to the SSL libraries either statically or dynamically. The former is the default but we recommend the latter. To link dynamically you must #define SSL_LINK_DYNAMIC.

In summary, all you need do to include SSL support in your IMAP server is:

1. Build MISE with the symbol MISE_TLS #defined and SSL_LINK_DYNAMIC if appropriate.
2. If linking dynamically, make a call from your server's initialization code to MISEInstallOpenSSL. This function (defined in MISE.h) allows you to pass MISE paths for the two SSL libraries. (This function supercedes an older and now obsolescent function called TlsInstall.)
3. Make a call from your server's initialization code to MISEInitializeOpenSSL. This function (also defined in MISE.h) provides the OpenSSL code with the paths to your certificate and private-key files. (This function supercedes an older and now obsolescent function called TlsServerInit.)
4. Listen for client connections on port 993, initiate the each session using SSL, and call the MISE function IMAP4StartServerThread with the bSSL parameter set to TRUE.
5. Ship the OpenSSL DLLs/shared libraries with your product. If shipping with the OpenSSL DLLs for Windows, install them either in the application directory or in the Windows System directory. If shipping the OpenSSL shared libraries for Linux, we recommend installing them in /usr/lib.
Expunged Messages

There is one issue that may prove thorny in your IMAP implementation. This concerns how you handle expunged messages.

The issue arises when an IMAP session has a mailbox selected and a third party deletes a message from that mailbox. The third party could be another IMAP session, a client session that interfaces with the mailstore through some other mechanism (such as WebMail), or something like an administrative program that culls expired messages. The problem arises because a standard-compliant IMAP implementation must guarantee each client session that no message will disappear from a selected mailbox until and unless that client takes action to make it disappear.

We need to delve into some details of IMAP to explain this fully. First, each IMAP session can be working in one (and only one) mailbox at a time. This mailbox is called the session's "selected" mailbox. Second, message deletion in IMAP is a two-step process consisting of flagging a message as deleted and later issuing an "expunge" that makes all flagged-as-deleted messages in the selected mailbox actually go away. Third, if multiple sessions are working in the same mailbox then, when one session expunges messages, those messages should remain present to the other sessions.

All this demands some care in handling mailboxes. There are several options for doing so:

1. You can design your MBM to properly handle multiple concurrent sessions for the same mailbox, doing whatever housekeeping is necessary so that any expunged message is only truly removed once all IMAP sessions that were active when the expunge occurred release the mailbox. You must make sure that any new sessions that select the mailbox during such a period do not see the expunged messages at all. The details of managing this are not onerous; they would normally involve keeping an expunged flag and a reference count for each message. Nevertheless, this can work only if you can control all relevant access to the mailstore. If the mailstore is autonomous or anything prevents your controlling third-party deletion, then another strategy must be used.

2. An equivalent mechanism is to make a working copy of each mailbox as an IMAP session selects it. Read operations can then be made against the copy while write operations must be made against both. This is a simple solution if copying a mailbox is itself a simple operation (for example, when a mailbox corresponds to a file) and if there is plenty of disk space. If you choose this option then it is up to your MBM to fully manage the mailbox copies.

3. You can avoid the whole nasty business by disallowing more than one IMAP session from using any given mailbox at a time. This is not as severe as it may seem at first glance but how satisfactory it would be in practice will obviously depend on the particular patterns of usage you expect your server to have. It is significant that the University of Washington IMAP server (UW-IMAP), a popular and well respected server, handles things in this fashion.

4. You can also essentially ignore the whole nasty business, let messages disappear from under IMAP sessions, and hope for the best. MISE will support your doing this by simply sending nothing when a client asks for any information pertaining to an expunged message. Such behavior just happens if you let it. When MISE asks your MBM to retrieve some information about a message and your MBM returns an error indication, MISE assumes that the message has been expunged, flags it as such, and returns nothing to the client. In practice, this seems to work acceptably. Another successful IMAP server, the Cyrus server, also works in this manner.

So, the bottom line is that, should you find a need to go to the trouble, you can do a lot to avoid expunge problems but, if you see no need, then you can safely do nothing and MISE will take care of things in a manner that is typically quite adequate although, make no mistake, against the rules.
IMAP Extensions
Here we summarize the many IMAP extensions supported by MISE. Some of these are enabled unconditionally, basically those that add value with negligible cost. Others, especially those that require extra code in the MBM, are optional and included only when a particular preprocessor symbol is #defined.

IDLE Command
The IDLE command is an IMAP4 extension defined in RFC 2177. The purpose is to make it easier for certain clients to receive asynchronous updates from the server.

Support for IDLE was added to MISE in version 1.7 (May 2004). It is enabled by building with the symbol MISE_IDLE defined and including the module idle.c (which may otherwise be omitted).

When a client sends an IDLE command, the IMAP server goes into a state where it does nothing but send status updates to the client. Such update indications are generated whenever messages are added or removed from the selected mailbox and whenever message flags change. The client can terminate the IDLE state any time it chooses to do so and, if it does not, the server will time out automatically after 30 minutes and terminate the session.

Obviously, in the IDLE state, the server must learn of changes to the mailbox from the MBM. There are two different ways that this can happen. The preferred way is for the MBM to support the async-update capability. This requires that the MBM create a thread of its own and make calls to the MBMAasyncUpdate function in the server whenever changes occur. If the MBM does not support the async-update capability then the server will call the MBMCheckUpdates function at one-minute intervals. More information relevant to this is given in the MBM manual and you might like to inspect the code in idle.c.

UIDPLUS
This is one of the earliest and simplest extensions to IMAP4. By default, when a client adds a message to an IMAP store, either by importing it using an APPEND command or by copying it within the store with the COPY command, the client is not informed of the UID of the new message. And it would usually be very valuable to have that UID to keep track of the message. This is especially true since, in a situation where other agents are creating new messages at the same time, determining the UID after the fact can be both tricky and slow. The UIDPLUS extension enhances the APPEND and COPY commands so that the server can return the UID(s) in its response. Arguably this should have been a part of the original protocol.

Code for supporting UIDPLUS is included unconditionally in MISE and enabled on a per-session basis when the MBM indicates that it is ready to support it – this means that it is then required to output the relevant UIDs from the MBM functions that implement import and copy. We designed it this way so that an MBM can have flexibility to the extent of supporting UIDPLUS for one user and not another if there is some obstacle that necessitates such a restriction. It is such a valuable feature however that we highly recommend that you support it if at all possible.

Login Referrals
Login Referrals are documented in RFC 2221. If you need your server to provide such referrals then you should build it with MISE_LOGIN_REFERRALS #defined. This causes "LOGIN-REFERRALS" to be included in the server’s list of IMAP capabilities. You may then provide referral information either at connection time or when a user logs in.

To make referrals at connection time, just return the appropriate message from your CustomServerGreeting() function. For example, "[REFERRAL IMAP://user;AUTH="*@SERVER2/] Server not accepting connections. Try SERVER2".
To make a referral at login time, set the appropriate structure fields as described for MBMLogin() in the MBM API manual.

**LOGINDISABLED**

The base IMAP specification requires that a server implement the LOGIN command which allows a client to authenticate a user by supplying a username and password in plain text. Requirements for Internet protocols in general that were published at a later date require that such insecure mechanisms as LOGIN be disabled. In the case of IMAP this would be in favour of the AUTHENTICATE command which supports session authentication using a variety of more secure mechanisms. Because LOGIN is historically required, a special case is needed to tell clients that a server does not actually support it, and that is the purpose served by this extension. Specifically, it serves to tell clients via an IMAP CAPABILITY response that LOGIN is not available. Depending on the environment your server will operate in, you may or may not need to disable LOGIN. Plenty of IMAP servers still support it. One advantage of LOGIN is that it makes authentication easy for manual sessions (that is sessions operated using a telnet client). You might consider enabling LOGIN in debug builds of your server but not for release builds. You tell MISE which authentication methods to support via a Custom function (CustomAuthMethods) documented elsewhere in this manual. If you do not include LOGIN as one of the methods to be supported then MISE will act with LOGIN disabled.

**ID Extension**

In basic IMAP there is no way for a client to tell what particular IMAP server it is talking to nor for an IMAP server to determine the particular client software operating a session. The original philosophy was that there should be no need for such knowledge and such knowledge might be a bad thing because it could lead to both clients and servers making special accommodations for particular servers and clients. There are however really useful reasons for being able to glean such information, especially for the purpose of directing bug reports to where they belong. So the ID extension was added which defines an ID command and response that serve to exchange such information as software product name, version and vendor.

The ID Extension is included unconditionally in MISE which means that any client that also implements it should send the server an ID command near the start of a session. MISE records the client information in this command and makes it available to the MBM if it wants it. By default, MISE sends an ID response with no information about the server. If you wish it to send some real information then #define the symbol MISE_ID_RESPONSE and equate it to a suitable text string (see documentation in the custmise.h header file.)

**BINARY Extension**

The original IMAP protocol was very carefully and deliberately designed to send any and all information as lines of text. Since Internet email messages themselves are also composed of lines of text, this was very reasonable. It means however that binary content in messages, which basically means images, sound and binary attachments are sent in encoded form so that they appear as text, typically using “base64” encoding. With the explosion of use of mobile devices for email, typically with relatively low channel capacities, it was seen as valuable to allow binary content to be sent as is, in true binary form. So the BINARY extension was born and allows clients to request that message elements be sent as binary data. Because the cost of this for the server is minimal (it takes just a little processor time to decode the text), this extension is hard-wired into MISE.

**UNSELECT command**

IMAP uses a two-step deletion process. As a first step, a message is flagged as deleted. A flagged-as-deleted message remains where it is in the mailstore but is typically hidden from the user by the client. The message is only actually removed from the mailstore when the second step, called an expunge, is performed. An EXPUNGE command can remove either specified or all flagged-as-deleted messages. When a client finishes working in a particular folder, it normally closes the folder using IMAP’s CLOSE command which forces an implicit expunge for all flagged-as-deleted messages in the folder. Sometimes a
client may want to close a folder without having this expunge done and the UNSELECT command was added for this purpose. Since UNSELECT is useful and costs nothing it is an integral part of MISE.

**Namespaces**

The original IMAP spec was written with the notion that a user might have access to multiple message stores, not just his or her own email repository but mailboxes belonging to other people and shared stores used for such content as newsgroups. There was however no mechanism for IMAP clients to discover what alternate stores are available to a user and these could be made known only by user configuration. RFC 2342 ("IMAP4 Namespace") describes an IMAP extension that facilitates access to multiple message stores by providing a command for an IMAP client to learn about them.

A "namespace" in this context can be thought of as an analog to a file system (or volume). Each namespace would typically correspond to a mail store and be represented as a tree of folders. Namespaces are nominally categorized into three groups: personal, other users' and shared. As a concrete example, in a Microsoft Exchange environment, a user would typically have just one personal namespace (his or her own mailbox), possibly access to one of two other user mailboxes (e.g. a personal assistant would have access to his boss'), and in many cases access to Public Folders on the server. Every IMAP session has a default namespace which would normally be the user's primary, private mail repository but could be a shared store.

Alternate namespaces are identified by a namespace prefix that is added to every folder path. It is conventional to create such prefixes starting with the octothorpe/hash character (#). If you can follow this convention, it is probably wise. If it is inconvenient to do so then that is probably okay but we do not have experience in this realm. The default namespace has no such prefix. RFC 3501 will give you some examples that may make this clearer but, be warned, much of what it has to say about namespaces beyond the examples is confusing.

Even with the Namespace extension, namespaces are not first-class items in IMAP. There is no sharp distinction between a root folder in the default namespace and an alternate namespace. Since namespaces are one of unfortunately many things that some IMAP clients do not handle well, we recommend that you avoid using them if you can. However, if you need to provide access to multiple mailstores then you probably will want to represent them as IMAP namespaces. An alternative is to meld multiple mailstores into a single hierarchy but this can be undesirable for three reasons: (1) mailstores may need to have different hierarchy characters which can only be done by making each a separate namespace, (2) effectively merging mailstores may make folder trees unwieldy in the sense of creating very lengthy responses to LIST commands, and (3) it may just not fit with user expectations if they want to see each store as a separate tree in their client.

The namespaces available to an IMAP user can conveniently be advertised via the NAMESPACE command but do not have to be. If your server needs to support access to multiple stores then you will probably want to enable MISE’s optional support for calls to the MBM to retrieve namespace information (done by #defining MISE_NAMESPACE in the custmise.h header file). The NAMESPACE command is supported in any case, since some clients expect it even though it is defined as optional. If MISE’s MISE_NAMESPACE option is not selected then a stock response is provided to a NAMESPACE command that indicates a single default personal namespace.

A limitation of MISE is that all namespaces must use the same hierarchy character. We plan to relax this in future but that will require some architectural changes.

**LIST-STATUS**

Basic IMAP has one command ("LIST") to find what folders are in a mailstore and another ("STATUS") to find out information about a folder such as how many messages it contains, how many unread messages and a few other such things. The LIST-STATUS extension essentially fuses the two commands so that a client can find out about folders and their contents with many fewer command-response transactions. Since
there is no cost for using this extension it is enabled by default but only when MISE is built with the
MISE_VERSION symbol defined as 4 (the default is 3). This is because the MBM function
MBMEnumerateFolders is changed in version 4 to accommodate LIST-STATUS.

**Extended SEARCH (ESEARCH)**

IMAP’s search command is very powerful in that it can be used to search through a folder using complicated
Boolean queries. It is limited however in that it only ever responds with a list of ids of messages that match.
Sometimes a client wants to know just the number of messages that match or the id of the highest matching
message. And this is where the ESEARCH extension comes in. It allows a client to specify a variety of bits
of information that it wants as the result of a SEARCH. Support for this is hard-coded into MISE since it is
so valuable and requires no support from the MBM.

**LANGUAGE Command**

The LANGUAGE extension enables an IMAP server to return human-readable text to a client in any natural
language that a user wishes and that a server supports. It is sufficiently important that it has a section all to
itself later in this manual.

**Conditional STORE**

Conditional STORE (“CONDSTORE”) is one of the heftiest additions to the IMAP protocol. It requires
associating with every message a generation counter that is updated whenever the message’s flags are
changed. Normally this counter is a 64-bit value that must be managed by the MBM. CONDSTORE allows
IMAP clients to track changes to messages more efficiently. Support in MISE is enabled by #defining
MISE_CONDITIONAL_STORE and implementing a few new MBM functions that allow MISE to obtain the
generational counter values. [This section requires expansion.]

**Quick Mailbox Resynchronization**

QRESYNC is an extension to CONDSTORE that enables a client to more efficiently learn about what has
changed in a folder since it last examined it. While MISE contains full support for QRESYNC, it has neither
been well tested nor used in any production server. We therefore suggest caution in using it. There is very
little support for QRESYNC among IMAP clients; in fact the only one we know of that uses it is Trojítá.
Authentication

Here we discuss the various authentication mechanisms available in IMAP and MISE. Please note that we are not experts in the areas of authentication and encryption.

IMAP basically supports three ways of authenticating (or “logging in”) a user: “preauth”, login and authentication.

PREAUTH

The “preauth” mechanism serves to automatically login a user when an IMAP session commences. A certain user could be authorized in this manner by virtue of, for example, the IP address that he or she connects from. A totally public IMAP server could automatically login any and every user with this mechanism. Note that this mechanism logs in a session without a password.

“Preauth” is little used in practice but MISE supports it via the custom function CustomPreAuthorize. Your version of this function can preauthorize a session based on whatever conditions you deem appropriate. It must store an appropriate username in the session structure (ISD).

LOGIN

IMAP’s LOGIN command is the most commonly used mechanism. It carries a plaintext username and password from the client to the server. (Of course if used in an SSL session then this command will, in fact, be encrypted like everything else.) An example is:

    0001 LOGIN jsmith "my secret"

Note that with LOGIN and all authentications mechanisms used by IMAP, a username and password are the only credentials used. There is no way to specify, for example, an account as there is in FTP. IMAP does allow a password to be null.

AUTHENTICATION

IMAP provides a structure for using arbitrary authentication mechanisms under the auspices of its AUTH command. These may be standard authentication methods defined by SASL (see RFC 2222) or “external” methods (which encompass those that are private and proprietary). A server advertises the methods it supports in its responses to a client’s CAPABILITY command; it may also include such information in its greeting message.

In practice, the mechanisms typically supported by IMAP clients and servers are those that are commonly used by Internet protocols such as PLAIN and GSSAPI. MISE supports PLAIN (as required by RFC 3501), CRAM-MD5, DIGEST-MD5 and, optionally, NTLM. CRAM-MD5 authentication is a simple means supported by a range of clients; MISE has included it since its inception. DIGEST-MD5 was added with version 2.6. NTLM demands some more discussion and we devote a section to it below.

Choosing what Authentication Methods to Support

In choosing what authentication methods to support in your server, you should probably be considering some combination of the following factors:

- The level of security required
- The methods supported by the clients most popular among your user base
- The methods provided by MISE
- How user validation is performed on the target system
The last of these deserves some discussion because authentication methods fall into two categories. First there are those that carry the password from the client to the server. These include the LOGIN command and PLAIN authentication. Second there are those that use the password at the client end to generate some data that is then validated using the password stored at the server. These include CRAM-MD5 and NTLM. The second group are much more secure than the first but depend on the server’s being able to retrieve any user’s password. There are many circumstances where a server cannot obtain passwords. On UNIX systems for example, assuming that authentication is done using system credentials (as opposed to some set of credentials private to the server), passwords are not even stored; all any program can do is call the operating system to validate a password (which is done using a hash). Another case where this limitation may arise is when authentication must be done on a system separate from the one on which the IMAP server runs.

There is actually a third category of authentication method typified by Kerberos. This uses public and private keys in addition to a username and password. We know little about Kerberos and few IMAP clients support it.

**Authentication Methods supported by MISE**

The basic authentication methods supported by MISE are “preauth”, the LOGIN command and AUTH PLAIN. (AUTH PLAIN is a minimal requirement for IMAP.) In addition it supports three of those methods that work only when the server can retrieve passwords: CRAM-MD5, DIGEST-MD5 and NTLM.

If you need to support an additional authentication method, please discuss your need with us. If the method in question is a standard one then we may be prepared to add it to MISE. Also, as a future enhancement, we plan an extension mechanism to allow licensees to plug in their own authentication functions.

**Authentication Functions**

MISE requires you to provide three functions to support authentication. These are in the Custom group of functions, not the MBM set. (In many cases, authentication will be intimately tied to the message store and therefore the MBM but we make no assumptions about this. We have one licensee, for example, who has to implement authentication on a separate system from the one that houses the server and mailstore.)

The functions are CustomAuthMethods, CustomGetAccountInfo and CustomValidateAccount. All are documented in the custom.c source module.

**NTLM Authentication**

We support NTLM authentication due to demand that arises because it is the one and only secure mechanism that is supported by Microsoft Outlook and Outlook Express. When you check the “Log on using Secure Password Authentication” option in one of these clients, what it actually means is “use NTLM authentication”. Here is a bunch of caveats about NTLM:

1. We added NTLM support to MISE in November of 2004. It has been well tested in the lab but not in the field. Because of its newness, we suggest treating it with a little caution.
2. NTLM is not a standard but a mechanism that was developed by, and which is proprietary to, Microsoft. As such, there is no authoritative public specification for NTLM. (In spite of this, Microsoft does appear to encourage outside support for NTLM.) We based our implementation on the document “The NTLM Authentication Protocol” written by Eric Glass and available at http://davenport.sourceforge.net/ntlm.html.
3. Because NTLM is not part of the IMAP standard, some terms of the MISE licensing agreement may not apply to our NTLM code. Should Microsoft change NTLM in future we do not, and indeed cannot, guarantee to embrace those changes in MISE.
4. Our support for NTLM is partial. Our code validates LM and NTLM responses but not LMv2 and NTLMv2 responses.
5. Our NTLM code depends upon external code to generate MD4 hashes and DES encryptions. We include some public-domain code to handle DES and calls to OpenSSL for MD4. Should you prefer to use alternative code for these purposes, it should be easy to plug that in.

6. The way that Outlook and Outlook Express perform NTLM authentication deserves some attention. Each time an authentication is attempted, the client first attempts it using the credentials of the logged-in Windows user. If that fails, it tries again using the username and password configured for the email account. MISE is not designed to validate Windows accounts but to work with its own set of accounts. Hence each authentication will involve these two steps. This is unfortunate since it compromises the user’s Windows account and wastes time attempting an authentication that never succeeds.

7. NTLM authentication does not work with null passwords.

8. NTLM authentication will and can work only when usernames follow the syntactic rules of Windows usernames. Do not enable NTLM authentication if you support usernames such as “//X:Jones” or “<\Pete>”.

9. When told to use NTLM, Outlook and Outlook Express will prompt the user for a domain name in addition to a username and password. With MISE a domain is not used and the field should be left empty.

To include support for NTLM authentication in your server, #define the symbol MISE_AUTH_NTLM and include the module NTLM.C in your build. MISE currently supports NTLM authentication only for Windows builds.

NTLM authentication is really a family of authentication methods (LM, LMv2, NTLM, NTLMv2 and NTLM2). Which method or methods are actually used in any instance is a matter negotiated between the client and server. The LM method is the only mandated one; it also happens to be the weakest cryptographically. As noted above, MISE currently supports only LM and NTLM. You may choose to build MISE’s NTLM code with support for LM only. The reason would be to avoid having to provide OpenSSL or some other MD4 function. The way to do this is to #define the symbol MISE_AUTH_NTLM_LM_ONLY. Then the NTLM code will be self-contained.
Internationalization

Version 3.0 of MISE adds support for internationalization as per RFC 5255. Currently this is available for Windows builds only. The first step in this endeavor is to support the LANGUAGE command. Other internationalization features will be implemented in due course.

The LANGUAGE Command

RFC 5255 states, somewhat ungrammatically:

The LANGUAGE command requests that human-readable text emitted by the server be localized to a language matching one of the language range argument as described by Section 2 of [RFC4647].

In other words, this command allows an IMAP client to select a particular natural language, from those on offer by the IMAP server, for human-readable text to be sent in. Basically such text is restricted to that on tagged IMAP responses and is most useful in providing diagnostic information to users in their own language when something goes wrong. In what way and to what degree such text is actually made visible (or audible?) to a user depends of course on the design of the client in question. This facility does not extend to translating command names, folder names or anything like that.

You engage support for LANGUAGE in MISE by # defining the MISE_LANGUAGE_COMMAND symbol and providing resource-only DLLs containing text in various languages. Each DLL is given a name of the form "LANGUAGE <list>.dll" where <list> is a space-separated list of language tags that the DLL provides support for. Each language DLL may contain text only in one language but that one language may be used for several language tags. For example, you might have "LANGUAGE Fr-Ca.dll" and "LANGUAGE Fr-Fr Fr.dll", the first to support Canadian French and the second to support not only French French but all other French varieties as well.

Every new session is initialized to use the default language, US English, to remain consistent with traditional IMAP behavior. Default response texts in US English are hard-coded into MISE for efficiency but can be overridden if required by a custom En-US DLL (using a means still to be determined). A default language may also be set for each IMAP account but a switch to this language is made only on request by the client.

At the source level, each language DLL consists of a STRINGTABLE containing UTF-16 strings equated to a set of resource symbols defined in the header file "MISE_Resources.h". Please look at the file "En-Us.rc" for an example. A sample entry from this file is:

```
MISE_STRING_AUTHENTICATE_ACCOUNT_DISABLED L"The given account is disabled"
```

This provides the text to be returned to the client when an attempted authentication fails because the account in question has been disabled. (Note that clients should select the language for a session prior to authentication so that diagnostics such as this one can be supplied in the most appropriate language.)

When a MISE instance starts up with LANGUAGE enabled, it scans its executable directory for language DLLs (those matching the pattern "LANGUAGE *.dll") and records all the languages provided. When a LANGUAGE command arrives with no arguments, a list of the available languages is returned. When a LANGUAGE command comes with a requested language or list of such, MISE finds the most appropriate DLL and switches the session in question to use that language. In the process, it calls a new MBM function (MBMSetLanguage) to tell the MBM what language is in use; this allows the MBM to provide its own response texts in the appropriate language.

Instructions for building language-support DLLs for MISE are provided in a separate document.
**Thread Pooling**

Currently this is available for Windows only. By default, each IMAP session is run by a thread created for and dedicated to that session. This arrangement works well for servers that accommodate up to a hundred or so simultaneous sessions. Supporting many hundreds or thousands of sessions is a problem though given how Windows manages threads. The absolute limit on the number of threads Windows will create per process is 2,028 but, because each requires some dedicated physical memory, it may prove more efficient to create a much smaller number and share them as much as possible. Ideally, in fact, one would create one thread per processor core and share them.

Using one thread per session makes programming easy because Windows then manages all of the business of dispatching threads. Using one thread per processor is potentially most efficient but demands that the server take responsibility for dispatching threads – and doing that efficiently is very challenging indeed. What we set out to do in our implementation of thread pooling is to strike a compromise between these two extremes.

Windows includes support for thread pooling in such a manner that the operating system will dynamically create and destroy threads for an application that chooses to use it. We choose to employ a hybrid arrangement where Windows’ internal thread pooling is used for certain purposes while a pool of threads explicitly created and managed by MISE serves other purposes. Specifically, we use Windows thread pooling for socket i/o completion and our own thread pool for everything else.

Ideally, MISE would be micro-tasked such that every operation that can block in the operating system (every file read, every critical section entry, every dispatch of data to the client) would be performed asynchronously such that no thread in the pool would ever be tied up waiting for something to happen. In practice, we perform socket i/o asynchronously and everything else synchronously. This means that a thread is never tied up waiting for input to arrive from the IMAP client or for output to be dispatched to the client – and these are typically the long, long waits, possibly minutes in duration. Threads will get blocked waiting for such things as file reads and writes to complete but these should take only milliseconds.

**When to use thread pooling**

Thread pooling may prove valuable in some servers but it is not for everyone. Specifically it will not work if sessions involve resources that must be owned by threads devoted to them.

**Enabling thread pooling**

You turn on MISE’s thread pooling by defining the preprocessor symbol MISE_THREAD_POOL and equating it to the number of threads to be created for the pool. For example, as a compiler option:

```
/DMISE_THREAD_POOL=400
```

The size of the pool is fixed. In future we may enhance thread pooling to manage a pool that grows and shrinks dynamically according to demand.

Three new C modules must be included in your build. These are async.c, ipasync.c and pool.c.

You must also provide two MBM functions, MBMInitializePoolThread and MBMTerminatePoolThread. These are documented in the MBM Developers Guide.

**Limitations of thread pooling**

As mentioned above, thread pooling is unsuitable if your MBM works in such a way that resources that nominally belong to the IMAP session are actually owned by the thread that starts that session.
Finally, it is probably inadvisable to use thread pooling if any of your MBM functions can block for many seconds or minutes at a time – which could be the case, for example, if it works with a mailstore that is remotely located.
Limiting the number of IMAP sessions

Early versions of MISE offered no option for putting a ceiling on the number of concurrent IMAP sessions – although, of course, such a limitation could be imposed in the code that listens for and accepts IMAP connections. With version 2.0, we provide the MISE_MAXIMUM_SESSION_COUNT option that allows you to set a limit within the IMAP server itself. If you do not #define this symbol, then no fixed limit is imposed. If you do #define it as in, for example:

```
/DMISE_MAXIMUM_SESSION_COUNT=800
```

then the server will politely refuse sessions beyond the 800th, suggesting that users try later.

There are at least three reasons that you may choose to restrict the number of sessions. First, it serves to limit the possible damage that can be caused by denial-of-service attacks. Second, it can be used to avoid the possibility of sessions failing due to lack of memory or other limited resources. And third, it can be applied in order to guarantee a certain performance level for users at the expense of occasionally telling a user or two to come back later.
Hooking into the Logging System

When you build MISE with the MISE_LOGGING symbol defined, the server incorporates our proprietary logging system. This serves primarily to have the server log information to a file recording the main events of the sessions it conducts. (This logged output is described elsewhere in this document.) You can hook this into your own logging system as follows.

#include the “Logger.h” header file supplied with MISE. This defines the logging functions that you can call. In your MBMInitialize function, cast the void pointer passed in to a MacleanLogger * and call LogSetCallback with that log pointer as the first argument. The third argument is the address of a callback function in your code that will then be called for each log message issued. The second argument is a reference value (of type void *) that is passed back to you in the callback; this is intended for establishing context. The last argument allows you to specify a set of options for the callback. To disable the callback, call LogSetCallback again with NULL for the callback pointer.

The callback function is written something like this:

```c
void LoggerCallback(
    void *ReferenceValue,  // [in] your reference value
    char Tag,              // [in] tag character (‘E’, ’I’, ’W’, etc)
    const char *Message)   // [in] log message (in UTF-8)
{
    clsMyLogger *pLogger = (clsMyLogger *) ReferenceValue;
    switch (cTag)
    {
    case 'D': pLogger->LogDebug(Message); break;
    case 'E': pLogger->LogError(Message); break;
    case 'I': pLogger->LogInformation(Message); break;
    case 'W': pLogger->LogWarning(Message); break;
    }
}
```

The set of callback options is:

- `MM_LOGBACK_INCLUDE_TIMESTAMP` include timestamp
- `MM_LOGBACK_INCLUDE_TAG` include tag
- `MM_LOGBACK_SKIP_FILING` do not issue messages to log file

Our logger writes each message to its log file prefixed by a timestamp, as in this example:

```
06-25-10 17:51:50.365 (I) 0B70/IMAP4 Session terminating
```

By default, the message passed to the callback does not include either this timestamp or the tag, (I) in the example. To have the tag included, select the `MM_LOGBACK_INCLUDE_TAG` option; to have both included, select the `MM_LOGBACK_INCLUDE_TIMESTAMP` option.

By default also, every log message is written to the server’s log file. To prevent this, select the `MM_LOGBACK_SKIP_FILING` option. This serves to inhibit the writing of log entries to the file as long as the callback is established.

Should you need additional options, please tell us.
MISE Structures
Fundamental to an understanding of the inner workings of the Engine, is an understanding of two key structures, the IMAP Session Descriptor or "ISD" and the IP Support structure or "IPS". (This section does not discuss MBM structures; they are described in the MBM document.)

The IMAP Session Descriptor (ISD)
All the data pertinent to an IMAP server session is held in an ISD. This includes information about its connection which is held in a contained structure, the IPS described next. Almost all MISE code depends on the ISD and most functions pass its address around as their first parameter.

An ISD is allocated (in the module imaps.c) when an IMAP session is established. The structure itself is defined in the file imap.h. We suggest you take a look at imap.h to get a feel for what is contained in the ISD.

The IP Support structure (IPS)
The IPS structure is defined in the file ipsupp.h. It maintains data about the IP connection associated with each IMAP session and is passed to all functions in the IP-support module (ipsupp.c).

The Account Information Structure (MISE_AccountInfo)
This structure (defined in custom.h) is passed to the CustomGetAccountInfo function to retrieve various pieces of configuration information about a server account. Your CustomGetAccountInfo function must put the username and password into this structure, and can set any other fields that are pertinent. If you support the IMAP LANGUAGE extension then you can use a field in this structure to specify the default language for the account.

The Server Parameters Structure (MISE_ServerParameters)
This structure (also defined in custom.h) is passed to the CustomGetServerParameters function to retrieve a few critical values that govern how the MISE server works at the highest level. It includes the maximum number of concurrent sessions to be supported and the server-level default language for human-readable response text.
Miscellaneous Matters

Strings
MISE is designed so that strings sent to the client for viewing by the user are defined as resources that can be loaded at run-time. In addition, the IMAP LANGUAGE extension allows strings to be supplied in multiple natural languages. Currently this extension is supported only for Windows builds but we will add support for it to Linux if and when there is demand.

Modifying the Sources
We strongly urge you not to modify any of the MISE sources other than the modules designed for customization. This will make it easy for you to integrate future updates. If you find a need to modify any MISE source then, we figure, it must either be to correct a problem (in which case the change should be transmitted to us for retrofitting into our sources) or to make a customization or extension (in which case we request that you ask us to add a hook for a custom function that will do what you require).

Aborting Sessions In Progress
If your server has an operator interface, you might want to include within it the ability for the operator to abort an IMAP session. This could be useful, for example, in the case of a session that is stuck in some manner or a session that is suspected of being nefarious. To do this, your code should call the MISE function IMAP4AbortSession. It requires one parameter which is a pointer to the session’s control structure (ISD).

Sending ALERT Messages
IMAP allows for an operator at the server end to send a text message to the client. This can be used, for example, to warn users that the server is going to be taken down. The client is supposed to display such a message to the user. In practice not all clients do this (not to mention that an automated client may not have a user as such) but enough do that it may be worth implementing this feature. To post an alert message, call the MISE function IMAP4PostAlertMessage. This takes two parameters, an ISD pointer or NULL and a pointer to the message which should be a null-terminated pure ASCII string. If the first parameter is NULL then the alert message is posted for all active sessions. MISE makes no guarantee that the message will ever actually be sent; it will however send it at the first opportunity. An alert message may be up to 511 characters (any additional characters will be discarded).
Frequently Asked Questions

Is there any limitation to having either POP3 or IMAP access the same email account? That is, today I use a POP3 client, tomorrow a IMAP and the day after, back to POP3?
The only limitation is one of concurrence. While a POP3 session is active, no other POP3 session must be allowed to the same account and no IMAP session should be allowed access to the account's Inbox. By the same token, when an IMAP session is active and working in an account's Inbox then a POP3 session for the account should be refused. It would probably be acceptable in practice to generalize that a bit and simply disallow simultaneous POP3 and IMAP access to an account.

As far as alternating serial access is concerned, it may not be commonly used, but there is no reason not to allow it and we feel strongly that it should be allowed. One hypothetical scenario where it might be used is with someone who prefers IMAP and uses that regularly when "in the office" but who might want to use POP while "on the road" because that is the only convenient option. The POP server can work using the internal equivalent of IMAP's \Seen flag; generally it should never delete messages but just turn off their \Seen flag once retrieved. Of course if a user uses only POP then retrieved messages do need to be purged at some point so you need something like a POP-only flag on the account.