The FreeM Manual

The Official Manual of FreeM
Version 0.7.0

John P. Willis

Table of Contents

Introduction

FreeM started its life as *FreeMUMPS*, written for MS-DOS and ported to SCO UNIX by a mysterious individual going by the name of "Shalom ha-Ashkenaz". It was released to MUG Deutschland in 1998. In 1999, Ronald L. Fox ported FreeM to Red Hat Linux 5. Thereafter, maintenance was taken over by the Generic Universal M Project, which changed its name first to Public Standard MUMPS and then by popular request to FreeM.

When GT.M was open-sourced in late 1999, FreeM and GUMP were essentially abandoned. L.D. Landis, the owner of the original GUMP SourceForge project, and one of FreeM's significant contributors, passed maintenance of FreeM and ownership of its SourceForge project to John Willis in 2014. At this point, FreeM would not compile or run on modern Linux systems, so steps were taken to remedy the most pressing issues in the codebase. Limitations on the terminal size (previously hard-coded to 80x25) were lifted, and new `$VIEW` functions were added to retrieve the terminal size information. `$X` and `$Y` intrinsic special variables were updated to support arbitrary terminal sizes, and FreeM was once again able to build and run.

In February of 2020, work began in earnest to build a development and support infrastructure for FreeM and begin the careful process of refining it into a more stable and robust product.

Production Readiness

FreeM is not yet production-ready. There are several show-stopping bugs that preclude a general release for public use:

- SSVs, aside from `^$JOB`, `^$DEVICE`, `^$EVENT`, and `^$ZPROCESS` are not implemented.
- `VIEW` commands and `$VIEW` functions are used extensively to configure and inspect the run-time behavior of FreeM, rather than the "canonical" SSV-based approach.
- Server sockets are not yet implemented.

Contributors

Current contributors denoted with a `+` following their name and role.

- Shalom ha-Ashkenaz (Original Implementer)
- John Best (IBM i and OS/400)
- Jon Diamond (Library, Utilities, Conformance)
- Ronald L. Fox (Initial port to Red Hat 5/libc-6)
- Winfried Gerum (Code, Advice, MTA coordination)
- Greg Kreis (Hardhats coordination, Dependencies)
- Larry Landis (Coordination, Code, Documentation)
- Frederick D.S. Marshall (MDC Standards Conformance) `+`
- Lloyd Milligan (Code, Testing, Documentation)
- Steve Morris (Code, Microsoft)
- John Murray (Code, Conformance)
- Wilhelm Pastoors (Testing, Documentation)

- Kate Schell (Coordination, Conformance, MTA, MDC, Advice)
- Lyle Schofield (Advice, Prioritization, Tracking, Project Management)
- Jim Stefanik (Linux on s390x, IBM AIX, IBM z/OS)
- Axel Trocha (Code, Utilities)
- Dick Walters (Project Lead, Chief Coordinator, MTA)
- David Whitten (QA Test Suite, MDC, Advice) +
- David Wicksell (Debugging, Code, Testing) +
- John Willis (Current Maintainer and Project Lead) +
- Steve Zeck (Code)

1 FreeM Invocation

1.1 Synopsis

```
$ ./freem [OPTIONS...] [[-r <entryref>] | [--routine=<entryref>]]
```

When FreeM loads, it searches the SYSTEM namespace for the ^%ZFREEM routine, and begins executing it.

When -r or --routine are passed on the command line, FreeM will load and run the specified routine instead of ^ZFREEM. Beginning with FreeM 0.1.7, routines invoked in this manner are no longer required to perform their own namespace setup with VIEW commands.

1.2 Command-Line Options

'-h', '--hardcopy'
: Enables hardcopy mode, echoing all output to a disk file. By default, this disk file is '\$freem_base/<*namespace-name*>/freem.hardcopy', but can be changed with the following command:

```
USER> VIEW 13:"'</path/to/hardcopy/file>'"
```

The file used for hardcopy mode may also be specified in '/etc/freem.conf' or '~/.freemrc'.

'-f', '--filter'
: Allows your M routines to be used as UNIX filters.

'-n', '--noclear'
: Disables automatic screen clearing when FreeM loads.

'-s', '--standard'
: Restricts the use of non-standard language features, including $Z... intrinsic special variables, $Z... intrinsic functions, Z... commands, as well as VIEW and $VIEW.

'-i', '--import'
: Causes your UNIX environment variables to be imported into FreeM's local symbol table.

'-r <*entryref*>', '--routine=<*entryref*>'
: Causes <entryref> to be executed at load, instead of ^%ZFREEM.

'-n <*namespace-name*>', '--namespace=<*namespace-name*>'
: Selects the FreeM namespace to be entered on startup. Must be defined in '/etc/freem.conf'.

1.3 Using FreeM for Shell Scripting

FreeM M routines can be used as shell scripts by providing a *shebang* line beginning with #!/path/to/freem as the first line of the routine. The following example presumes that FreeM is installed at '/usr/local/bin/freem' and uses the USER namespace:

```
#!/usr/local/bin/freem
MYSCRIPT ;
 SET ^$JOB($JOB,"NAMESPACE")="USER"
 WRITE "This is output from an M routine used as a shell script.",!
 Q
```

Currently, the script needs to have a '`.m`' file extension. You will also need to select an appropriate namespace in your script using the `SET ^$JOB($JOB,"NAMESPACE")="<`*`namespace`*`>"` command before attempting to call other routines or access globals.

You will also need to set the script's permissions to *executable* in order for this to work:

```
$ chmod +x myscript.m
```

2 The FreeM Direct-Mode Environment

The FreeM direct-mode environment is the mode entered when FreeM is loaded without the use of '`-r <entryref>`' or '`--routine=<entryref>`':

```
Coherent Logic Development FreeM
Version 0.7.0-x86_64-Linux (commit 4ececff; jpw AT pasithea Tue 13 Oct 2020 09:03:27 AM MDT)

           *
          * *
         *   *
    ***************
     * *       * *
      *  FreeM  *
     * *       * *
    ***************
         *   *
          * *     Copyright (C) 1998 MUG Deutschland
           *      Copyright (C) 2014, 2020 Coherent Logic Development LLC

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.

   PID:                  3343
   Principal I/O:        0:"/dev/pts/5"


USER>
```

The prompt (`USER>`) indicates the currently-active namespace. If any uncommitted direct-mode transactions have been started, the prompt will change to reflect the current value of `$TLEVEL`:

```
TL1:USER>
```

In the above example, `TL1` indicates that `$TLEVEL` is currently *1*.

2.1 Direct-Mode Commands

When you are in direct mode, in addition to M commands, a number of internal commands are available to help developers be more productive:

`?` Accesses FreeM online help. Requires GNU `info(1)` to be installed on your local system.

`events` Writes a list of *event classes* and their `ABLOCK` counts:

```
USER> events

Event Class          Processing Mode ABLOCK Count
-----------          --------------- ------------
COMM                 Disabled        0
HALT                 Disabled        0
IPC                  Disabled        0
INTERRUPT            Disabled        0
POWER                Disabled        0
TIMER                Disabled        0
USER                 Disabled        0
WAPI                 Disabled        0
```

`history` Prints a list of all the direct-mode commands you have entered across all sessions.

`rcl <history-index>`
Allows you to recall command number <*history-index*> and run it again. Obtain the value for <*history-index*> from the output of the `history` command.

`!<external-command>`
Invokes a shell to run <*external-command*> from within FreeM. This temporarily disables `SIGALRM` handling in FreeM, which may interrupt the use of event-driven M programming commands including `ESTART` and `ESTOP`.

If the `<` character is supplied immediately preceding <*external-command*>, FreeM will append the contents of M local variable `%` to <*external-command*> as standard input.

If the `>` character is supplied immediately preceding <*external-command*>, FreeM will take the standard output stream of <*external-command*> and store it in M local variable `%`.

`%` contains the number of lines in the input or output. `%(1)..%(n)` contains the data for lines 1-*n*.

`tdump` Writes detailed information about the status of any pending transactions to `$PRINCIPAL`.

If you issue a `HALT` command at the direct-mode prompt, you will exit out of FreeM. However, if you issue a `HALT` command when `$TLEVEL` is greater than zero, you will be given the opportunity to commit or rollback any pending transactions:

```
USER> TSTART


TL1:USER> SET ^MYGLOBAL=1
```

```
TL1:USER> HALT

UNCOMMITTED TRANSACTIONS EXIST:

 $TLEVEL 1*
  Operations for Transaction ID: k8xj1de
  1:  action = 0  key = ^MYGLOBAL  data = 1

Would you like to c)ommit or r)ollback the above transactions and their operations? ($TLEVEL = 1) r


Transactions have been rolled back.
```

In the above example, the user selected `r` to rollback the single pending transaction.

2.2 REPL Functionality

FreeM direct mode allows you to enter M expressions directly from the direct-mode prompt, as long as they begin with a number:

```
USER> S DENOM=10


USER> 100/DENOM

10
USER>
```

Such expressions will be immediately evaluated, and the result printed on `$PRINCIPAL`.

3 Intrinsic Special Variables

3.1 $DEVICE

Returns the status of the device currently in use, and is writable.

If `$DEVICE` returns *1*, an error condition exists on the current device.

3.2 $ECODE

Returns a comma-delimited list of error conditions currently present, and is writable. An empty `$ECODE` indicates no errors.

3.3 $ESTACK

Returns the depth of the program execution stack since the last time `$ESTACK` was `NEW`ed. `NEW`-able, but not `SET`-able. Differs from the `$STACK` ISV in that it is `NEW`-able, and resets to a value of 0 when `NEW`ed.

3.4 $ETRAP

Sets or retrieves the M code that is run when an error is encountered or `$ECODE` is set to a non-blank value. `$ETRAP` code executes when `$ECODE` becomes non-blank.

3.5 $HOROLOG

Returns a string containing the current date and time as `<days>,<seconds>`, where `<days>` represents the number of days since the M epoch (midnight on 31 December 1840), and `<seconds>` represents the number of seconds since the most recent midnight.

3.6 $IO

Represents the current input/output device. Read-only.

3.7 $JOB

Represents the process ID of the FreeM instance currently in use.

3.8 $KEY

Represents the sequence of control characters that terminated the last `READ` command on `$IO`.

3.9 $PRINCIPAL

Represents the primary input/output device. Usually a terminal or virtual terminal.

3.10 $QUIT

If the current execution context was invoked as an extrinsic function, `$QUIT` returns *1*. Otherwise, returns *0*.

When `$QUIT` returns *1*, a subsequent `QUIT` command must have an argument.

3.11 $STACK

Represents the current stack level.

3.12 $STORAGE

Represents the number of bytes of free space available in FreeM's heap.

3.13 $SYSTEM

Returns the MDC system ID of FreeM.

3.14 $TEST

`$TEST` is a writable, `NEW`-able ISV that is *1* if the most recently evaluated expression was *true*. Otherwise, returns *0*.

`$TEST` is implicitly `NEW`ed when entering a new stack frame for extrinsic functions and argumentless `DO`. `$TEST` is *not* implicitly `NEW`ed when a new stack frame is entered with an argumented `DO`.

3.15 $TLEVEL

Returns a numeric value indicating the current level of transaction nesting in the process. When `$TLEVEL` is greater than *0*, uncommitted transactions exist.

3.16 $TRESTART

Returns an empty string, as FreeM transaction processing does not yet support restartable transactions.

3.17 $X

Represents the current column position of the FreeM cursor.

> *Non-Standard Behavior*
> In FreeM, setting `$X` will move the FreeM cursor.

3.18 $Y

Represents the current row position of the FreeM cursor.

> *Non-Standard Behavior*
> In FreeM, setting `$Y` will move the FreeM cursor.

3.19 $ZA

On the `HOME` device, always `0`. On other devices, represents the byte offset to the beginning of the file.

3.20 $ZB

Represents the last keystroke.

3.21 $ZCONTROLC

3.22 $ZDATE

Returns the current date, in `YYYY/MM/DD` format.

3.23 $ZERROR

Returns the last error message.

3.24 $ZF

3.25 $ZHOROLOG

Output `$HOROLOG`-style time, with the addition of milliseconds.

3.26 $ZINRPT

Gets or sets the interrupt enable/disable flag.

3.27 $ZJOB

Returns the `$JOB` value of the parent process (used in subroutines started with the `JOB` command).

3.28 $ZLOCAL

Returns the last local variable referenced.

3.29 $ZMATCHCONTROL

Returns control characters.

3.30 $ZMATCHNUMERIC

Returns all numbers `0-9`.

3.31 $ZMATCHPUNCTUATION

Returns all punctuation characters.

3.32 $ZMATCHALPHABETIC

Returns all alphabetic characters.

3.33 $ZMATCHLOWERCASE

Returns all lowercase characters.

3.34 $ZMATCHUPPERCASE

Returns all uppercase characters.

3.35 $ZMATCHEVERYTHING

Returns control characters, numbers, punctuation, and alphabetic characters.

3.36 $ZPRECISION

Gets or sets the number of digits of numeric precision used for fixed-point decimal arithmetic. Defaults to 100 digits.

3.37 $ZREFERENCE

Returns the last *glvn* referenced.

3.38 $ZSYSTEM

3.39 $ZTIME

Returns the system time in `HH:MM:SS` (24-hour) format.

3.40 $ZTRAP

Sets or retrieves the entryref to be executed when an M program execution error occurs under FreeM-style or DSM 2.0-style error processing.

In FreeM-style error processing, `$ZTRAP` is specific to each program execution stack level.

In DSM 2.0-style error processing, `$ZTRAP` is the same for all program execution stack levels.

When FreeM encounters an error, if `$ZTRAP` is nonempty and `$ETRAP` is empty, FreeM will perform an implicit `GOTO` to the entryref indicated in `$ZTRAP`.

If `$ETRAP` is nonempty when FreeM encounters an error, the value of `$ZTRAP` is ignored, whether FreeM-style or DSM 2.0-style error processing is enabled.

3.41 $ZVERSION

4 Intrinsic Functions

4.1 $ASCII

Returns the ASCII code (in decimal) for one character in a string.

```
SET RESULT=$ASCII(<string>[,<index>])
```

If <*index*> is not supplied, `$ASCII` will return the ASCII code of the first character. Otherwise, returns the ASCII code of the character at position <*index*>.

4.2 $CHAR

Returns a string of characters corresponding to a list of ASCII codes.

```
SET RESULT=$CHAR(<ascii-code>[,<ascii-code>,...])
```

4.3 $DATA

Returns a numeric value 0, 1, 10, or 11, depending on whether a referenced node is defined, has data, or has children:

```
SET RESULT=$DATA(<node>)
```

The return values are as follows:

```
0: <node> is undefined
1: <node> has data but no children
10: <node> has children but no data
11: <node> has children and data
```

4.4 $EXTRACT

4.5 $FIND

4.6 $FNUMBER

4.7 $GET

4.8 $JUSTIFY

4.9 $LENGTH

4.10 $NAME

4.11 $NEXT

4.12 $ORDER

4.13 $PIECE

4.14 $QLENGTH

4.15 $QSUBSCRIPT

4.16 $QUERY

4.17 $RANDOM

4.18 $REVERSE

4.19 $SELECT

4.20 $STACK

Returns information about the program execution stack. The `$STACK` intrinsic function has both a one-argument form and a two-argument form.

Syntax (One-Argument)

```
$STACK(<num>)
```

If *num* is `0`, returns the command with which this FreeM instance was invoked.

If *num* is `-1`, returns the current program execution stack level.

If *num* represents a valid program execution stack depth above `0`, returns one of the following values indicating the reason for which the referenced program execution stack level was created:

`$$` If `$STACK(<num>)="$$"`, program execution stack level `num` was created as the result of an extrinsic function call

`<`*m-command*`>`
: If `$STACK(<num>)` returns a valid M command, the referenced program execution stack level was created as a result of the *m-command* command.

Syntax (Two-Argument)

```
$STACK(<num>,"[ECODE|MCODE|PLACE]")
```

Returns the error codes, M program code, or entryref applicable to the action that created program execution stack level *num*.

4.21 $TEXT

4.22 $TRANSLATE

4.23 $VIEW

4.24 $ZBOOLEAN

Performs *boolean-operation* on numeric arguments *A* and *B*.

Syntax

```
SET RESULT=$ZBOOLEAN(A,B,boolean-operation)
```

`$ZBOOLEAN` Operations (*boolean-operation* values)

`0`	Always *false*
`1`	`A AND B`
`2`	`A AND NOT B`
`3`	`A`
`4`	`NOT A AND B`
`5`	`B`
`6`	`A XOR B`
`7`	`A OR B`
`8`	`A NOR B`
`9`	`A EQUALS B`
`10`	`NOT B`
`11`	`A OR NOT B`
`12`	`NOT A`
`13`	`NOT A OR B`
`14`	`A NAND B`
`15`	Always *true*

4.25 $ZCALL

4.26 $ZCR

4.27 $ZCRC

4.28 $ZDATA

4.29 $ZDATE

4.30 $ZEDIT

4.31 $ZHOROLOG

4.32 $ZHT

4.33 $ZKEY

4.34 $ZLENGTH

4.35 $ZLSD

4.36 $ZM

4.37 $ZNAME

4.38 $ZNEXT

4.39 $ZORDER

4.40 $ZPIECE

4.41 $ZPREVIOUS

4.42 $ZREPLACE

4.43 $ZSYNTAX

4.44 $ZSORT

4.45 $ZTIME

4.46 $ZZIP

5 Commands

5.1 ABLOCK

Increments the event block counter for one or more event classes. While the block counter for an event class is greater than zero, registered event handlers for that event class will not execute, and will instead be queued for later execution once the block counter reaches zero (all blocks removed).

An implicit `ABLOCK` on all event classes occurs when an event handler subroutine is executing. As soon as a `QUIT` is reached within an event handler, an implicit `ABLOCK` will occur.

Syntax

`ABLOCK:postcondition`

In its argumentless form, `ABLOCK` increments the block counter for *all* event classes, provided the optional *postcondition* is either *true* or omitted.

`ABLOCK:postcondition evclass1...,evclassN`

In its inclusive form, `ABLOCK` increments the block counters for all event classes named in the list, provided the optional *postcondition* is either *true* or omitted.

`ABLOCK:postcondition (evclass1...,evclassN`

In its exclusive form, `ABLOCK` increments the block counters for all event classes *except for* those named in the list, provided the optional *postcondition* is either *true* or omitted.

5.2 ASTART

Enables asynchronous event handling for one or more event classes.

Syntax

`ASTART:postcondition`

In its argumentless form, `ASTART` enables asynchronous event handling for all event classes, provided the optional *postcondition* is either *true* or omitted.

`ASTART:postcondition evclass1...,evclassN`

In its inclusive form, `ASTART` enables asynchronous event handling for all event classes named in the list, provided the optional *postcondition* is either *true* or omitted.

`ASTART:postcondition (evclass1...,evclassN)`

In its exclusive form, `ASTART` enables asynchronous event handling for all event classes *except for* those named in the list, provided the optional *postcondition* is either *true* or omitted.

5.3 ASTOP

Disables asynchronous event handling for one or more event classes.

Syntax

`ASTOP:postcondition`

In its argumentless form, `ASTOP` disables asynchronous event handling for all event classes, provided the optional *postcondition* is either *true* or omitted.

```
ASTOP:postcondition evclass1...,evclassN
```

In its inclusive form, `ASTOP` disables asynchronous event handling for all event classes named in the list, provided the optional *postcondition* is either *true* or omitted.

```
ASTOP:postcondition (evclass1...,evclassN)
```

In its exclusive form, `ASTOP` disables asynchronous event handling for all event classes *except for* those named in the list, provided the optional *postcondition* is either *true* or omitted.

5.4 AUNBLOCK

Decrements the event block counter for one or more event classes.

Syntax

```
AUNBLOCK:postcondition
```

In its argumentless form, `AUNBLOCK` decrements the block counter for *all* event classes, provided the optional *postcondition* is either *true* or omitted.

```
AUNBLOCK:postcondition evclass1...,evclassN
```

In its inclusive form, `AUNBLOCK` decrements the block counters for all event classes named in the list, provided the optional *postcondition* is either *true* or omitted.

```
AUNBLOCK:postcondition (evclass1...,evclassN
```

In its exclusive form, `AUNBLOCK` decrements the block counters for all event classes *except for* those named in the list, provided the optional *postcondition* is either *true* or omitted.

5.5 BREAK

Interrupts running routine to allow interactive debugging.

Syntax

```
BREAK:postcondition
```

In its argumentless form, `BREAK` suspends execution of running code, provided the optional *postcondition* is *true* or omitted.

```
BREAK:postcondition breakflag
```

In its single-argument form, `BREAK` sets *Ctrl-C* handling and error handling characteristics, provided the optional *postcondition* is *true* or omitted. The following table enumerates the possible values of *breakflag*

`0` Disables *Ctrl-C* handling

`-2` Enables normal FreeM error handling

`2` Enables *Digital Standard MUMPS* v2 error handling

`"default"`
Enables *Ctrl-C* handling

5.6 CLOSE

Closes an input/output device.

Syntax

```
CLOSE:postcondition
```

In its argumentless form, `CLOSE` closes all I/O devices except for device 0 (the `HOME` device), provided the optional *postcondition* is *true* or omitted.

```
CLOSE:postcondition channel
```

In its single-argument form, `CLOSE` closes the I/O device associated with channel *channel*, provided that *channel* represents a currently-open device, and the optional *postcondition* is *true* or omitted.

5.7 DO

5.8 ELSE

5.9 FOR

5.10 GOTO

5.11 HALT

5.12 HANG

5.13 IF

5.14 JOB

5.15 KILL

5.16 KSUBSCRIPTS

Kills only the descendant subscripts (but not the data value) of a referenced global, local, or SSV (where allowed).

Syntax

```
KSUBSCRIPTS:postcondition var1,...
```

In the above *inclusive* form, `KVALUE` will kill the descendant subscripts at each local, global, or SSV node specified in the list (provided that the optional *postcondition* is *true* or omitted), but will leave the data value intact.

> *Note* The below *argumentless* and *exclusive* forms of `KSUBSCRIPTS` are not implemented in FreeM, as of version 0.3.3, but are planned for a future release.

```
KSUBSCRIPTS:postcondition
```

In the above *argumentless* form, `KSUBSCRIPTS` will kill the descendant subscripts at the root of each local variable (provided that the optional *postcondition* is *true* or omitted), but will leave data values intact.

```
KSUBSCRIPTS:postcondition (var1,...)
```

In the above *exclusive* form, `KSUBSCRIPTS` will kill the descendant subscripts of all local variables, *with the exception of* those named in the list, provided that the optional *postcondition* is *true* or omitted, while leaving their data values intact.

5.17 KVALUE

Kills only the data value (but not descendant subscripts) of a referenced global, local, or SSV (where allowed).

Syntax

```
KVALUE:postcondition var1,...
```

In the above *inclusive* form, `KVALUE` will kill the data values at each local, global, or SSV node specified in the list (provided that the optional *postcondition* is *true* or omitted), but will leave descendant subscripts intact.

> *Note* The below *argumentless* and *exclusive* forms of `KVALUE` are not implemented in FreeM, as of version 0.7.0, but are planned for a future release.

```
KVALUE:postcondition
```

In the above *argumentless* form, `KVALUE` will kill the data values at the root of each local variable (provided that the optional *postcondition* is *true* or omitted), but will leave descendant subscripts intact.

```
KVALUE:postcondition (var1,...)
```

In the above *exclusive* form, `KVALUE` will kill the data values of all local variables, *with the exception of* those named in the list, provided that the optional *postcondition* is *true* or omitted, while leaving their descendant subscripts intact.

5.18 LOCK

5.19 MERGE

Merges the contents of one global, local, or SSV subtree to another global, local, or SSV.

Syntax

```
MERGE A=^$JOB
```

The above example will merge the `^$JOB` SSV into the `A` local. Note that the FreeM implementation of `MERGE` does not yet support multiple merge arguments. Returns error `M19` if either the source or the target variable are descendants of each other.

5.20 NEW

5.21 OPEN

Opens sequential or socket I/O devices and files and associates them with a numeric FreeM input/output channel.

Syntax (Sequential Files)

`OPEN`*`:postcondition channel`*`:"`*`filename/access-mode`*`"`

Opens *filename* for reading and/or writing, and associates the file with FreeM I/O channel *channel*, provided that the optional *postcondition* is *true* or omitted. The below table lists the valid options for *access-mode*:

`r` Read-only access

`w` Create a new file for write access

`a` Write access; append to existing file

`r+` Read/write access

> *I/O Path*
>
> You cannot specify a fully-qualified filesystem path in the FreeM `OPEN` command. By default, FreeM will assume that *filename* exists in the directory indicated in `^$JOB($JOB,"CWD")`. If you wish to access files in other directories, you must first set the *I/O Path* in `^$JOB($JOB,"IOPATH")`.
>
> The following example will set the I/O path to `/etc`:
>
> ```
> SET ^$JOB($JOB,"IOPATH")="/etc"
> ```

If *channel* was already `OPEN`ed in the current process, calling `OPEN` on the same channel again implicitly closes the file or device currently associated with *channel*.

Syntax (Network Sockets)

Network sockets use a dedicated range of FreeM I/O channels ranging from 100-255. `OPEN`ing a socket I/O channel does *not* implicitly connect the socket. Connecting the socket to the specified remote host is accomplished by the `/CONNECT` control mnemonic supplied to the `USE` command.

`OPEN`*`:postcondition socket-channel`*`:"`*`hostname-or-address`*`:`*`port`*`:`*`address-family`*`:`*`connection-type`*`"`

Socket Parameters

socket-channel
: The socket I/O channel to use. This must be in the range of 100-255.

hostname-or-address
: The hostname or IP address to connect to. If a hostname is supplied, `OPEN` will implictly do a name lookup, the mechanism of which is typically determined by the configuration of `/etc/nsswitch.conf` on most UNIX and UNIX-like platforms.

port
: The TCP or UDP port to which the socket will connect on the remote host.

address-family
: The address family to use. Either *IPV4* or *IPV6*.

connection-type
Which connection type to use. Either *TCP* or *UDP*.

If you do not specify the address family and connection type, they will default to *IPV4* and *TCP*, respectively.

5.22 QUIT

5.23 READ

5.24 SET

5.25 TCOMMIT

5.26 TRESTART

5.27 TROLLBACK

5.28 TSTART

5.29 USE

Sets `$IO` to a particular FreeM I/O channel, allowing `READ`s from and `WRITE`s to the associated terminal, sequential file, or network socket. Also sets various device parameters.

Syntax (Terminal)

`USE:postcondition io-channel[:(right-margin:input-field-length:device-status-word:position:line-terminator:break-key)]`

For terminals, *io-channel* must be 0.

Semantic and functional description of each device parameter TBA.

Syntax (Sequential Files)

`USE:postcondition io-channel[:seek-position:terminator:nodelay)]`

For sequential files, *io-channel* must be in the range 1-99.

Semantic and functional description of each device parameter TBA.

Syntax (Network Sockets)

`USE:postcondition io-channel`

The above syntax will set `$IO` to *io-channel*, directing successive `READ`s and `WRITE`s to *io-channel*, provided the optional *postcondition* is *true* or omitted.

`USE:postcondition io-channel:/CONNECT`

The above syntax will set `$IO` to *io-channel*, as in the prior example, but will also attempt to connect to the host and port specified for *io-channel* when it was `OPEN`ed. The `/CONNECT` control mnemonic is only valid for socket channels whose connection type is `TCP`. Using `/CONNECT` on a `UDP` socket channel will throw `SCKAERR` (error code 55).

For network sockets, *io-channel* must be in the range 100-255.

5.30 VIEW

5.31 WRITE

5.32 XECUTE

5.33 ZALLOCATE

5.34 ZASSERT

Triggers error `ZASSERT` if the supplied truth-valued expression *tvexpr* is *true* (*1* is *true*, and *0* is *false*).

The `ZASSERT` error is catchable whether using standard-style, FreeM-style, or DSM 2.0-style error processing.

Syntax

```
ZASSERT <tvexpr>
```

Example

```
USER> ZASSERT 1=1


USER> ZASSERT 1=0


>> Error ZASSERT:  programmer assertion failed in SYSTEM::^%ZFREEM  [$STACK = 0]
>> ZASSERT 1=0
               ^
```

5.35 ZBREAK

5.36 ZDEALLOCATE

5.37 ZGO

5.38 ZHALT

5.39 ZINSERT

5.40 ZJOB

5.41 ZLOAD

5.42 ZNEW

5.43 ZPRINT

5.44 ZQUIT

5.45 ZREMOVE

5.46 ZSAVE

5.47 ZTRAP

5.48 ZWATCH

Sets a watchpoint on a global, local, or SSV node.

Syntax

In its *argumentless* form, `ZWATCH` toggles watchpoints on and off, provided the optional *postcondition* is *true* or omitted.

```
ZWATCH:postcondition
```

In its *inclusive* form, `ZWATCH` adds, removes, or examines watchpoints, provided the optional *postcondition* is *true* or omitted.

A `+` adds a new watchpoint to the following variable.

A `-` removes an existing watchpoint for the following variable.

A `?` examines the status of a watchpoint for the following variable.

```
ZWATCH:postcondition [+|-|?]var1...,[+|-|?]varN
```

The following example demonstrates turning watchpoint processing on and adding a watchpoint for global variable `^jpw(1)`. It then changes the value of `^jpw(1)`.

```
USER [LEGACY]> ZWATCH

Watchpoints enabled.

USER [LEGACY]> ZWATCH +^JPW(1)

Added '^JPW("1")' to the watchlist.

USER [LEGACY]> SET ^JPW(1)="new value"

>> WATCHPOINT:  ^JPW("1") => 'new value' (changed 1 times)
```

The following example will remove that watchpoint:

```
USER [LEGACY]> ZWATCH -^JPW(1)

Removed '^JPW("1")' from the watchlist.

USER [LEGACY]> ZWATCH ?^JPW(1)
```

```
'^JPW("1")' is not being watched.
```

5.49 ZWRITE

Writes the names and values of M variables to `$IO`.

Syntax

`ZWRITE`*`:postcondition`*

In the argumentless form, writes the names and values of all local variables to `$IO` if the optional *postcondition* is *true* or omitted.

`ZWRITE`*`:postcondition ArrayName`*`,...`

In the inclusive form, writes the names and values of all local, global, or structured system variables specified in the list of *ArrayName*s to `$IO` if the optional *postcondition* is *true* or omitted.

`ZWRITE`*`:postcondition`*` (`*`ArrayName`*`,...)`

In the exclusive form, writes all local variables *except* those specified in the list of *ArrayNames* to `$IO` if the optional *postcondition* is *true* or omitted.

6 Structured System Variables

SSV subscripts are each described in the following format:

<ssvn-subscript-name> +/-R +/-U +/-D

The R, U, and D flags represent Read, Update, and Delete. A minus sign indicates that the given operation is *not* allowed, and a plus sign indicates that the given operation *is* allowed.

6.1 ^$CHARACTER

The `^$CHARACTER` SSV is not yet implemented.

6.2 ^$DEVICE

FreeM implements several important pieces of functionality in the `^$DEVICE` SSV.

The first subscript of `^$DEVICE` represents the I/O channel of an `OPEN`ed device.

The following values for the second subscript are supported:

`EOF` +R -U -D
: Returns `1` if the I/O channel has encountered an end-of-file condition; `0` otherwise. Only valid if the I/O channel is connected to a sequential file.

`LENGTH` +R -U -D
: Returns the length of the file connected to the I/O channel. Only valid if the I/O channel is connected to a sequential file.

`MNEMONICSPACE` +R -U -D
: Returns the current *mnemonic-space* in use for the referenced I/O channel. Always `X364` for terminals and blank for sequential files.

`DSW` +R +U -D
: Sets or returns the current *Device Status Word* controlling terminal characteristics. Only valid for I/O channel 0.

`TERMINATOR` +R +U -D
: Sets or returns the `READ` terminator for the specified I/O channel. Must be either `$C(13,10)` or `$C(10)`. Currently only supported for socket devices (those having an I/O channel of 100-255).

Example

The following example M code opens `/etc/freem.conf` and reads its contents line-by-line until the end of the file is reached.

```
SET ^$JOB($JOB,"IOPATH")="/etc"   ; set I/O path to /etc
OPEN 1:"freem.conf/r" ; open freem.conf for reading
;
; read until we run out of lines
;
FOR  USE 1 READ LINE USE 0 QUIT:^$DEVICE(1,"EOF")  D
. WRITE LINE,!
```

```
;
CLOSE 1
QUIT
```

6.3 ^$DISPLAY

The `^$DISPLAY` SSV is not yet implemented.

6.4 ^$EVENT

The `^$EVENT` SSV is not yet implemented.

6.5 ^$GLOBAL

The `^$GLOBAL` SSV is not yet implemented.

6.6 ^$JOB

FreeM fully implements `^$JOB` per ANSI X11.1-1995, as well as several extensions proposed in the M Millennium Draft Standard.

The first subscript of `^$JOB` represents the `$JOB` of the process.

If you `KILL` a first-level subscript of `^$JOB`, the `SIGTERM` signal will be sent to the corresponding UNIX process, causing pending transactions to be rolled back and the process to be terminated. If the targeted process is in direct mode, the user will be prompted with options of either rolling back or committing any pending transactions.

The following subscripts are supported:

`CHARACTER` +R -U -D
: Returns the character set of the job.

`CWD` +R +U -D
: Returns or sets the current working directory of the job.

`EVENT` +R +U +D
: The subtree contained under `^$JOB($J,"EVENT")` defines asynchronous event handlers for the current job. Please see *Asynchronous Event Handling* for more information.

`GLOBAL` +R -U -D
: Returns the global environment of the job.

`IOPATH` +R +U -D
: Returns or sets the *I/O path* to be used by the `OPEN` command.

`PRIORITY` +R +U -D
: Returns or sets the *nice* value of the FreeM job.

`ROUTINE` +R -U -D
: Returns the name of the routine currently being executed by the job.

`$PRINCIPAL` +R -U -D
: Returns the value of `$PRINCIPAL` for the job.

`$TLEVEL` +R -U -D
: Returns the current transaction level (value of `$TLEVEL` for the job.

`$IO` +R -U -D
: Returns the current value of `$IO` for the job.

`USER` +R -U -D
: Returns the UID of the user owning the job.

`GROUP` +R -U -D
: Returns the GID of the group owning the job.

`NAMESPACE` +R +U -D
: Returns or sets the name of the job's currently-active namespace.

6.7 ^$LOCK

The `^$LOCK` SSV is not yet implemented.

6.8 ^$PDISPLAY

The `^$PDISPLAY` SSV is not yet implemented.

6.9 ^$ROUTINE

The `^$ROUTINE` SSV is not yet implemented.

6.10 ^$SYSTEM

The `^$SYSTEM` SSV is not yet implemented.

6.11 ^$WINDOW

The `^$WINDOW` SSV is not yet implemented.

6.12 ^$ZPROCESS

Provides access to `procfs`, which is a filesystem-like abstraction for UNIX process metadata contained in `/proc`, as well as features for examining and controlling the state of processes external to the FreeM interpreter.

The first subscript always represents the *process ID* of the external process being acted upon.

The following values for the second subscript are supported:

`EXISTS` +R -U -D
: Returns 1 if the referenced process exists; 0 otherwise.

`ATTRIBUTES` +R -U -D
: Exposes the `/proc` files as descendant subscripts, i.e., `WRITE ^$ZPROCESS(2900,"ATTRIBUTES","cmdline"),!` would print the initial command line used to invoke process ID 2900.

`SIGNAL` -R +U -D

Allows signals to be sent to the referenced process. The following subscript is an integer value corresponding to the desired signal number. You may obtain a list of signal numbers on most UNIX systems with the command `kill -l`.

7 Operators

7.1 Unary +

7.2 Unary -

7.3 + (Add)

7.4 += (Add/Assign)

7.5 ++ (Postfix Increment)

7.6 - (Subtract)

7.7 -= (Subtract/Assign)

7.8 – (Postfix Decrement)

7.9 * (Multiply)

7.10 *= (Multiply/Assign)

7.11 / (Divide)

7.12 /= (Divide/Assign)

7.13 \ (Integer Divide)

7.14 \= (Integer Divide/Assign)

7.15 # (Modulo)

7.16 #= (Modulo/Assign)

7.17 ** (Exponentiate)

7.18 **= (Exponentiate/Assign)

7.19 < (Less Than)

7.20 <= (Less Than or Equal To)

7.21 > (Greater Than)

7.22 >= (Greater Than or Equal To)

7.23 _ (Concatenate)

7.24 _= (Concatenate/Assign)

7.25 = (Equals)

7.26 [(Contains)

7.27] (Follows)

7.28]] (Sorts After)

7.29 ? (Pattern Match)

7.30 & (Logical AND)

7.31 ! (Logical OR)

7.32 ' (Logical NOT)

7.33 @ (Indirect)

8 Sequential I/O

9 Network I/O

Network I/O in FreeM is supplied through I/O channels 100-255. The normal `READ` and `WRITE` syntax will work with network sockets, with a few exceptions.

9.1 Opening and Connecting a Client Socket

To open a client socket and connect to it, you will need to call the `OPEN` command and the `USE` command:

```
;
; Set socket read terminator to LF
;
SET ^$DEVICE(100,"TERMINATOR")=$C(10)
;
; Open an IPv4 TCP socket to mail.mydomain.com on port 25 (SMTP)
; and connect to it
;
OPEN 100:"mail.mydomain.com:25:IPV4:TCP"
USE 100:/CONNECT
;
; Read a line of input from the remote host and write it to the terminal
;
NEW LINE
READ LINE
USE 0
WRITE LINE,!
;
; CLOSE the socket and disconnect
;
CLOSE 100
QUIT
```

10 Asynchronous Event Handling

Asynchronous event handling in FreeM follows the specifications of the unpublished MDC *Millennium Draft Standard*.

10.1 Setting Up Async Event Handlers

Asynchronous handlers are configured through the `^$JOB` SSV. In order to become proficient in writing asynchronous event handling code, you need to be aware of several important concepts:

Event Classes
: *Event classes* denote particular categories of events. These include `COMM`, `HALT`, `IPC`, `INTERRUPT`, `POWER`, `TIMER`, and `USER` event classes. At present, only `INTERRUPT` events are supported.

Event Identifiers
: *Event identifiers* denote the precise nature of the event that has occurred. For instance, resizing the terminal window in which a FreeM job is running will send an event of class `INTERRUPT` with an event identifier of `SIGWINCH` (short for *SIGnal WINdow CHange*).

Event Handlers
: *Event handlers* are M routines or subroutines that can be registered to run when an event of a certain event class occurs.

Event Registration
: *Event registration* is the process of modifying the `^$JOB` SSV to associate a particular event class and event identifier with an event handler routine or subroutine.

Event Block
: *Event blocking* is the means by which asynchronous event handling can be temporarily suspended. For example, asynchronous events are temporarily and implicitly blocked for the duration of event handler execution, unless explicitly un-blocked within the event handler. Event handling can also be blocked and unblocked programatically from M code using the `ABLOCK` and `AUNBLOCK` commands.

The following sections of this chapter will take you step-by-step through setting up an event handler for `SIGWINCH` signal handling.

10.2 Registering an Asynchronous Event Handler

To register an event handler, use the following syntax:

```
SET ^$JOB($JOB,"EVENT",event-class,event-identifier)=entryref
```

For example, use the following to register `^RESIZE` as an asynchronous event handler for `SIGWINCH` events:

```
SET ^$JOB($JOB,"EVENT","INTERRUPT","SIGWINCH")="^RESIZE"
```

This by itself will not enable asynchronous event handling, as it merely *registers* an event handler, associating it with event class `INTERRUPT` and event identifier `SIGWINCH`.

10.3 Enabling Asynchronous Event Handling

In order to enable asyncronous event handling, the `ASTART` command is used. In the following example, we will enable asynchronous event handling for the `INTERRUPT` event class:

```
ASTART "INTERRUPT"
```

Omitting the `"INTERRUPT"` argument will enable asynchronous event handling for *all* event classes. See `ASTART` in the commands section for more details.

Once this is done, any event handlers registered for the `INTERRUPT` event class in `^$JOB` will be executed asynchronously as appropriate.

10.4 Disabling Asynchronous Event Handling

To disable asynchronous event handling, the `ASTOP` command is used. In the following example, we will disable asynchronous event handling for the `INTERRUPT` event class:

```
ASTOP "INTERRUPT"
```

Omitting the `"INTERRUPT"` argument will disable asynchronous event handling for *all* event classes. See `ASTOP` in the commands section for more details.

You may also disable asynchronous event handling for a specific event identifier by `KILL`ing the appropriate node in the `^$JOB` SSV, which unregisters the event handler altogether. The following example will unregister the event handler for the `SIGWINCH` event identifier:

```
KILL ^$JOB($JOB,"EVENT","INTERRUPT","SIGWINCH")
```

10.5 Temporarily Blocking Asynchronous Event Handling

To temporarily block processing of specific event classes, you will use the `ABLOCK` command. `ABLOCK` functions incrementally, that is, each successive call to `ABLOCK` will increment a counter of blocks held for the specified event class or classes, and each successive call to `AUNBLOCK` will decrement that counter. Event handling for the specified event classes will be blocked as long as the `ABLOCK` counter for those classes is greater than zero. Thus, event blocking is cumulative, in a manner similar to M incremental locks.

The following example blocks asynchronous event handling for the `INTERRUPT` event class:

```
ABLOCK "INTERRUPT"
```

Note that entering an event handler causes an implicit `ABLOCK` of *all* event classes, to prevent event handlers from interrupting other event handlers during their execution. This may be overridden by calling `AUNBLOCK` for one or more event classes within an event handler. However, unblocking event handling during an event handler should be done with great caution, as this can make the flow of code execution somewhat unpredictable, especially if M globals are modified inside of an event handler routine or subroutine.

Modifying M globals within event handlers is allowed but strongly discouraged, as doing so can lead to logical corruption of the database. If you must modify an M global within an event handler, guard all such operations with prodigious and careful use of `LOCK`s, ensuring that such modifications occur in the desired logical order.

11 Synchronous Event Handling

12 GUI Programming with MWAPI

13 User-Defined Z Commands

14 User-Defined Z Functions

15 User-Defined SSVs

16 System Library Routines

16.1 ^%ZCOLUMNS

This routine is the implementation of the `$ZCOLUMNS` intrinsic special variable.

16.2 ^%ZFREEM

This routine is the default startup routine for FreeM running in direct mode.

Running `DO INFO` from direct mode will use this routine to display information about the current FreeM status and namespace configuration.

16.3 ^%ZHELP

This routine implements the online help feature of FreeM, invoked by typing `?` in direct mode. It simply asks the underlying system to execute the command `info freem`.

16.4 ^%ZROWS

This routine is the implementation of the `$ZROWS` intrinsic special variable.

17 Error Processing

FreeM exposes three means of processing M program execution errors:

FreeM-style error processing
: FreeM-style error processing exposes a read/write error trap in `$ZTRAP`. The contents of `$ZTRAP` must be either empty or a valid M entryref, to which FreeM will `GOTO` if an error occurs. Each program stack execution level can have its own `$ZTRAP` error handler enabled.

DSM 2.0-style error processing
: DSM 2.0-style error processing emulates the `$ZTRAP` behavior of Digital Standard MUMPS v2. It has the same behavior as FreeM-style error handling, with the exception that in DSM 2.0-style error processing, only one `$ZTRAP` error handler is set across all program stack execution levels.

Standard error processing
: Standard error processing uses the `NEW`-able `$ETRAP` variable to store error handler code, which may be any valid M code. The code in `$ETRAP` will run when an error occurs or the `$ECODE` ISV becomes non-empty. Stack information for standard error handling is provided by the `$STACK` ISV, the `$STACK()` intrinsic pseudo-function, and the `NEW`-able `$ESTACK` ISV.

  If `$ETRAP` is non-empty when an error condition occurs, `$ZTRAP` is ignored, regardless of whether FreeM-style or DSM 2.0-style error processing is enabled at the time of the error.

For further information on switching between FreeM-style and DSM 2.0-style `$ZTRAP` error handling, see the documentation for the `BREAK` command.

18 Debugging

19 System Configuration

19.1 Installing FreeM

19.2 Namespaces Overview

Configuration and administration of FreeM and the applications it hosts centers around the concept of *namespaces*, which represent a collection of M routines and globals existing within a well-defined directory hierarchy.

Beneath the FreeM installation directory (typically `/var/local/freem`) exists a number of subdirectories, each corresponding to a single FreeM namespace.

In the example below, two namespaces have been defined, named `SYSTEM` and `USER`. This is a fairly typical configuration, and routines and globals whose names begin with the `%` character, which are generally considered to be code and data to be shared among all namespaces in a FreeM system, are typically stored in the `SYSTEM` namespace, while each individual application or related set of applications will be managed beneath another namespace, such as the `USER` namespace presented below:

```
$freem_base
+- SYSTEM
|  +- routines
|  |  +- %ZFREEM.m
|  |  +- %ZCOLUMNS.m
|  |  +- %ZFRMXEC.m
|  |  +- %ZFRMSAMP.m
|  |  +- %ZROWS.m
|  |  +- %ZHELP.m
|  +- globals
|     +- ^%SYS
+- USER
   +- routines
   |  +- MYAPP.m
   +- globals
      +- ^MYGLOBAL
```

19.3 Listing Namespaces

To list all namespaces defined in '`freem.conf`', type the following command:

```
$ namespace list

Namespaces Defined in /etc/freem.conf:

SYSTEM
USER
```

In this example, the `SYSTEM` and `USER` namespaces are the only ones defined.

19.4 Adding Namespaces

When adding new applications to your FreeM installation, it is important to plan an appropriate namespace configuration. In general, it is advisable to place each application in its own namespace, as this will prevent conflicts in routine and global names, which can easily lead to data corruption. However, there are cases where it is preferable (or even essential) to combine multiple applications into a single namespace, i.e., when two applications rely on the ability to access each other's routines and/or globals, both applications must reside in a shared namespace.

In order to add a namespace to FreeM, you use the `namespace add` command. In the following example, we will add a new namespace called `MVTS`, for installing the M Validation and Test Suite:

```
$ namespace add MVTS
Adding namespace MVTS...


Namespace MVTS has been created.
Access it with the following command:

 $ freem --namespace=MVTS

Or if FreeM is already running:

 SYSTEM> SET ^$JOB($JOB,"NAMESPACE")="MVTS"
```

The `namespace` utility supports customization of a great many namespace options, including configuration of journaling and lock table location, among others.

For more information on the `namespace` utility, please consult the relevant section of the manual in Appendix A (FreeM Utilities).

19.5 Removing Namespaces

Not yet implemented.

19.6 Importing Routines

FreeM fully supports the `%RO`/`%RI` distribution format for the transport of collections of application routines. The `ri` utility, located in `$freem_base/sbin/ri`, will allow you to import such a file directly into a defined namespace with minimal effort.

In this example, we will create a `VPE` namespace and import the *Victory Programming Environment* into it:

```
$ namespace add VPE
Adding namespace VPE...


Namespace VPE has been created.
Access it with the following command:
```

```
 $ freem --namespace=VPE
```

Or if FreeM is already running:

```
 SYSTEM> SET ^$JOB($JOB,"NAMESPACE")="VPE"
```

$ *ri --namespace=VPE --file=VPE15P2.RSA*

```
FreeM Routine Import from 'VPE15P2.RSA'

  * Using FreeM namespace VPE
  * Percent routines will be loaded into /home/jpw/.freem/SYSTEM/routines
  * User routines will be loaded into /home/jpw/.freem/VPE/routines

Routines


Routines:

 XVEMBLDA
 XVEMBLDB
 XVEMBLDL
 XVEMBLD
 XVEMD1
 XVEMDC
 XVEMSYN
```

...lines omitted...

```
 XVVMI00S
 XVVMINI1
 XVVMINI2
 XVVMINI3
 XVVMINI4
 XVVMINI5
 XVVMINIS
 XVVMINIT
 XVVMVPE


Loaded 246 user routines and 0 percent routines (246 total).
```

20 Accessing FreeM from C Programs

FreeM provides a library, ‘libfreem.so’, as well as corresponding header file ‘freem.h’, allowing C programmers to write programs that access FreeM globals, locals, structured system variables, subroutines, and extrinsic functions. This functionality can be used to implement language bindings and database drivers for external systems.

In order to be used in your C programs, your C programs must link with ‘libfreem.so’ and include ‘freem.h’. This will allow your C code access to the function prototypes, data structures, and constants required for calling the ‘libfreem.so’ APIs.

You must exercise caution in developing programs that interface with FreeM through ‘libfreem.so’ to ensure that all ‘libfreem.so’ API calls are serialized, as FreeM and the ‘libfreem.so’ library are neither thread-safe nor reentrant.

You must also avoid setting signal handlers for SIGALRM, as FreeM uses SIGALRM to manage timeouts for LOCK, READ, and WRITE.

20.1 freem_ref_t Data Structure

The libfreem API uses a struct of type freem_ref_t in order to communicate state, pass in values, and return results.

The data structure, defined in ‘freem.h’, looks like this:

```
typedef struct freem_ref_t {

    /*
     * The 'reftype' field can be one of:
     *
     *  MREF_RT_LOCAL
     *  MREF_RT_GLOBAL
     *  MREF_RT_SSV
     */
    short reftype;

    /*
     * The 'name' field is the name of the local variable,
     * global variable, or SSV (without ^ or ^$).
     */
    char name[256];

    /*
     * Returned data goes in a string, so you've got to figure out the
     * whole M canonical number thing yourself. Good luck. :-)
     */
    char value[STRLEN];

    short status;

    unsigned int subscript_count;
```

```
    char subscripts[255][256];

} freem_ref_t;
```

freem_ref_t Members

'reftype' The 'reftype' member determines whether we are operating on a local variable, a global variable, or a structured system variable. It may be set to any of following constants: MREF_RT_LOCAL, MREF_RT_GLOBAL, or MREF_RT_SSV.

'name' The 'name' member contains the name of the global, local, or SSV to be accessed. You *must not* include leading characters, such as ^ or ^$.

'value' This member contains the value read from or the value to be written to the global, local, or SSV.

'status' This member gives us various API status values after the API call returns. In general, this value is also returned by each API function.

'subscript_count'
The number of subscripts to be passed into the API function being called. This value represents the maximum index into the first dimension of the subscripts array.

'subscripts'
A two-dimensional array containing the subscripts to which we are referring in this API call.

20.2 freem_ent_t Data Structure

The freem_function() and freem_procedure() APIs in libfreem use the freem_ent_t struct in order to indicate the name of the entry point being called, any arguments being passed to it, and the return value of the called function (not used for freem_procedure()).

The data structure, defined in 'freem.h', looks like this:

```
typedef struct freem_ent_t {

    /* name of function or procedure entry point */
    char name[256];

    /* return value */
    char value[STRLEN];

    /* value of ierr on return */
    short status;

    /* argument count and array */
    unsigned int argument_count;
    char arguments[255][256];

} freem_ent_t;
```

freem_ent_t Members

'name' The 'name' member contains the name of the extrinsic function or procedure to be called.

'value' This member contains the value returned by the function called. Not used by `freem_procedure()`.

'status' This member gives us the value of `ierr` after the function or procedure call returns. The possible values of `ierr` are listed in `merr.h`.

'argument_count'
The number of arguments to be passed into the extrinsic function or procedure being called. This value represents the maximum index into the first dimension of the `arguments` array.

'arguments'
A two-dimensional array containing the arguments to be passed into the extrinsic function or procedure being called.

20.3 freem_init()

Initializes `libfreem` in preparation for calling other APIs.

Synopsis

`pid_t freem_init(char *namespace_name);`

Parameters

`namespace_name`
Specifies the namespace to use.

Return Values

Returns the process ID of the `libfreem` process on success, or `-1` on failure.

Example

This example prompts the user to enter a FreeM namespace and then attempts to initialize `libfreem` to use the selected namespace.

```
#include <stdio.h>
#include <string.h>
#include <freem.h>

int main(int argc, char **argv, char **envp)
{
    char namespace[256];

    /* get the namespace name to use */
    printf("Enter FreeM namespace to use: ");
    fgets(namespace, 255, stdin);

    /* remove the trailing newline */
    namespace[strcspn(buffer, "\n")] = '\0';

    /* initialize libfreem using the provided namespace */
```

```
    if(freem_init(namespace) == TRUE) {
        printf("\nSuccess\n");
    }
    else {
        printf("\nFailure\n");
    }

    return 0;
}
```

20.4 freem_version()

Returns the version of FreeM in use.

Synopsis

`short freem_version(char *result);`

Parameters

`result` The `result` parameter is a pointer to a buffer in which the FreeM version information will be returned. The caller must allocate memory for this buffer prior to calling this API. It should be at least 20 bytes in length.

Return Value

Returns `0`.

Example

This example will display the FreeM version on standard output.

```
#include <stdio.h>
#include <string.h>
#include <freem.h>

int main(int argc, char **argv, char **envp)
{
    char version[20] = {0};

    freem_init(‘‘USER’’);
    freem_version(version);

    printf(‘‘FreeM version:  %s\n’’, version);

}
```

20.5 freem_set()

Sets a FreeM local node, global node, or writable SSV node.

Synopsis

`short freem_set(freem_ref_t *ref);`

Parameters

`freem_ref_t`
: This parameter is a pointer to a `freem_ref_t` struct. The caller must allocate the memory for this struct.

Return Value

Returns `OK` on success, or one of the other error values defined in `merr.h`.

Example

This example sets the value `blue` into global node `^car("color")`.

```
#include <stdio.h>
#include <string.h>
#include <freem.h>

int main(int argc, char **argv, char **envp)
{
    freem_ref_t ref;

    /* we're setting a global */
    ref.reftype = MREF_RT_GLOBAL;

    /* access global "car" */
    strcpy(ref.name, "car");

    /* set up the subscripts */
    ref.subscript_count = 1;
    strcpy(ref.subscripts[0], "color");


    /* use the USER namespace */
    freem_init("USER");

    /* write the data out */
    freem_set(&ref);

}
```

20.6 freem_get()

Retrieves a FreeM local node, global node, or writable SSV node.

Synopsis

`short freem_get(freem_ref_t *ref);`

Parameters

`freem_ref_t`
: This parameter is a pointer to a `freem_ref_t` struct. The caller must allocate the memory for this struct.

Return Value

Returns `OK` on success, or one of the other error values defined in `merr.h`.

Example

This example retrieves the character set of the current process.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <freem.h>

int main(int argc, char **argv, char)
{
    pid_t pid;
    freem_ref_t ref;

    /* get the PID of this process */
    pid = getpid();

    /* we want to access an SSV */
    ref.reftype = MREF_RT_SSV;

    /* set up the name and subscripts */
    strcpy(ref.name, "JOB");

    ref.subscript_count = 2;
    sprintf(ref.subscripts[0], "%d", pid);
    strcpy(ref.subscripts[1], "CHARACTER");

    /* initialize libfreem, using the USER namespace */
    freem_init("USER");

    /* call libfreem API */
    freem_get(&ref);

    /* output the character set info */
    printf("PID %d character set is '%s'\n", pid, ref.value);
}
```

20.7 freem_kill()

Deletes a FreeM local node, global node, or killable SSV node, as well as all of its children.

*short freem_kill(freem_ref_t *ref);*

Parameters

`freem_ref_t`
: This parameter is a pointer to a `freem_ref_t` struct. The caller must allocate the memory for this struct.

Return Value

Returns `OK` on success, or one of the other error values defined in `merr.h`.

Example

```
#include <stdio.h>
#include <string.h>
#include <freem.h>

int main(int argc, char **argv, char **envp)
{
    freem_ref_t ref;

    /* we're killing a global node */
    ref.reftype = MREF_RT_GLOBAL;

    /* access global "car" */
    strcpy(ref.name, "car");

    /* set up the subscripts */
    ref.subscript_count = 0;

    /* use the USER namespace */
    freem_init("USER");

    /* kill the global and all its descendant subscripts */
    freem_kill(&ref);
}
```

20.8 freem_data()

20.9 freem_order()

20.10 freem_query()

20.11 freem_lock()

20.12 freem_unlock()

20.13 freem_tstart()

20.14 freem_trestart()

20.15 freem_trollback()

20.16 freem_tlevel()

20.17 freem_tcommit()

20.18 freem_function()

20.19 freem_procedure()

Appendix A FreeM Utilities

A.1 Global Compactor (gcompact)

Compacts the specified global in place.

Syntax

`gcompact /path/to/global/file`

A.2 Block Examiner (gfix)

The *gfix* interactive utility program permits navigation of the B-tree structure of the specified global a block at a time.

Syntax

`gfix </path/to/global/file>`

A.3 Global Lister (gl)

This utility lists the contents of the specified global in various formats to standard output.

Syntax

`gl [OPTIONS] </path/to/global/file>`

'`-k`' Show global keys alone on a separate line.

'`-d`' Show global data alone on a separate line.

'`-n`' Show global keys and data, each on a separate line, with the keys in naked indicator form relative to the previous key.

If none of these options are supplied, the output has each node's key and data on the same line separated by an `=` sign. The '`-k`' and '`-d`' switches may be combined as '`-kd`' in order to output keys and data on alternating lines.

A.4 Lock Examiner (glocks)

This utility lists the contents of the lock table to standard output.

Syntax

`glocks [OPTIONS...] [</path/to/global/file>]`

Options

'`-pid <process-id>`'
Causes all locks owned by <*process-id*> to be cleared from the lock table.

A.5 Global Repair Tool (grestore)

This utility will fix problems with the specified global.

Syntax

`grestore </path/to/global/file>`

A.6 Global Validator (gverify)

This utility checks the specified global file for database corruption and inconsistencies.

Syntax

```
gverify <path/to/global/file>
```

A.7 Namespace Manager (namespace)

Adds, removes, lists, or displays the configuration of FreeM namespaces, allowing the user to specify all relevant configuration options.

Syntax

```
namespace [add | remove | show] [-n <namespace> | --namespace=<namespace>]
          [OPTIONS...]
namespace list
```

Options

'`-h`', '`--help`'
: Displays a summary of `namespace` syntax and command-line options.

'`-n` <*namespace*>', '`--namespace=`<*namespace*>'
: Sets the namespace being acted upon.

'`-m [inactive | read | write]`', '`--jnlmode=[inactive | read | write]`'
: Sets the journaling mode for the namespace; '`inactive`' by default.

'`-f` <*journal-file*>', '`--jnlfile=`<*journal-file*>'
: Sets the file that FreeM will use for journaling; '`/tmp/freem.journal`' by default.

'`-l` <*locktab-file*>', '`--locktab=`<*locktab-file*>'
: Sets the file that FreeM will use for maintaining the `LOCK` table; '`/tmp/freem.locktab`' by default.

'`-z` <*zalloctab-file*>', '`--zalloctab=`<*zalloctab-file*>'
: Sets the file that FreeM will use for maintaining the `ZALLOCATE` table; '`/tmp/freem.zalloctab`' by default.

'`-c` <*hardcopy-file*>', '`--hardcopy=`<*hardcopy-file*>'
: Sets the file FreeM will use when invoked in hardcopy mode; '`/tmp/freem.hardcopy`' by default.

'`-p` <*freem-path*>', '`--path=`<*freem-path*>'
: Informs this program of the location where FreeM is installed. You must upply the '`-p`' or '`--path`' option if the `$freem_base` environment variable is not set.

A.8 Routine Import (ri)

Allows the user to import routines in the `%RO/%RI` distribution format into a FreeM namespace.

Syntax

```
ri --file=<archive-file> --namespace=<namespace-name> [--overwrite]
```

Options

'--file=<*archive-file*>'
: Specifies the %RO/%RI-format file whose routines you wish to import.

'--namespace=<*namespace-name*>'
: Specifies the FreeM namespace into which routines from <*archive-file*> will be loaded.

'--overwrite'
: Indicates that ri should overwrite any routines from <*archive-file*> that already exist in <*namespace-name*>. Use with a preponderance of caution.

Appendix B FreeM VIEW Commands and Functions

B.1 VIEW 16: Total Count of Error Messages/View Single Error Message

Unknown semantics

B.2 VIEW 17: Intrinsic Z-Commands

Allows the user to retrieve or specify the list of intrinsic Z-commands that FreeM will attempt to run internally, allowing intrinsic Z-commands implemented internally to be replaced with M equivalents implemented as %-routines in the `SYSTEM` namespace.

B.3 VIEW 18: Intrinsic Z-Functions

Allows the user to retrieve or specify the list of intrinsic Z-functions that FreeM will attempt to run internally, allowing intrinsic Z-functions implemented internally to be replaced with M equivalents implemented as %-routines in the `SYSTEM` namespace.

B.4 VIEW 19: Intrinsic Special Variables

Allows the user to retrieve or specify which special variables are implemented internally.

B.5 VIEW 20: Break Service Code

Allows the user to view or specify the code that will be run when a `BREAK` is encountered.

B.6 VIEW 21: View Size of Last Global

Allows the user to view the size of the last referenced global.

B.7 VIEW 22: Count VIEW 22 Aliases

Retrieves the number of VIEW 22 aliases in effect.

B.8 VIEW 23: View Contents of Input Buffer

Retrieves the contents of the I/O input buffer.

B.9 VIEW 24: Maximum Number of Screen Rows

Retrieves the maximum number of screen rows supported in the current FreeM build.

B.10 VIEW 25: Maximum Number of Screen Columns

Retrieves the maximum number of screen columns supported in the current FreeM build.

B.11 VIEW 26: DO/FOR/XECUTE Stack Pointer

Retrieves the `DO`, `FOR`, and `XECUTE` stack pointer.

B.12 VIEW 27: DO/FOR/XECUTE Stack Pointer (On Error)

Retrieves the `DO`, `FOR`, and `XECUTE` stack pointer (on error).

B.13 VIEW 28: Switch Symbol Table

Switches the symbol table? We aren't currently aware of what this means.

B.14 VIEW 29: Copy Symbol Table

Copies the symbol table? We aren't currently aware of what this means.

B.15 VIEW 30: Inspect Arguments

Retrieves the arguments passed to the `freem` executable.

B.16 VIEW 31: Count Environment Variables

Allows the user to inspect the number of variables in the process environment table.

Syntax

```
WRITE $VIEW(31),!
```

Appendix C Implementation Limits

Appendix D US-ASCII Character Set

Code	Character
000	<NUL>
001	<SOH>
002	<STX>
003	<ETX>
004	<EOT>
005	<ENQ>
006	<ACK>
007	<BEL>
008	<BS>
009	<HT>
010	<LF>
011	<VT>
012	<FF>
013	<CR>
014	<SO>
015	<SI>
016	<DLE>
017	<DC1>
018	<DC2>
019	<DC3>
020	<DC4>
021	<NAK>
022	<SYN>
023	<ETB>
024	<CAN>
025	<EM>
026	<SUB>
027	<ESC>
028	<FS>
029	<GS>
030	<RS>
031	<US>
032	<space>
033	!
034	“
035	#

Appendix E FreeM Project Coding Standards

E.1 Module Headers

Module headers should adhere to the following format:

```
/*
 *                            *
 *                           * *
 *                          *   *
 *                     ***************
 *                      * *       * *
 *                       *  MUMPS  *
 *                      * *       * *
 *                     ***************
 *                          *   *
 *                           * *
 *                            *
 *
 *   mlib.h
 *    Function prototypes, structs, and macros for FreeM
 *    binding library
 *
 *
 *   Author: John P. Willis <jpw@coherent-logic.com>
 *    Copyright (C) 1998 MUG Deutschland
 *    Copyright (C) 2020 Coherent Logic Development LLC
 *
 *   Last modified: 29 February 2020
 *
 **/
```

The Star of David in module headers is a convention started by Shalom ha-Ashkenaz, the unidentified original author of FreeMUMPS/FreeM. We will continue to employ it in honor of his most valuable contribution to the M community.

E.2 Variable Naming

Variables should be named in all lowercase letters, and words within them delimited by underscores, such as `my_useful_variable`. `PascalCase` and `camelCase` are not to be used in this codebase under any circumstances.

Constants defined via the C preprocessor should be in all uppercase letters, with words within them likewise delimited by underscores, such as:

```
#define MY_USEFUL_CONSTANT 1
```

E.3 Indentation and General Layout

This project uses four spaces for indentation. Tabs are not to be used under any circumstances, and all source files must use a linefeed character to delineate lines. If you are working on a Windows machine, you must take care to follow this, as Windows will use a carriage return followed by a linefeed by default.

This project follows a modified version of what is known as the Stroustrup indentation style.

E.4 Brace Placement (Functions)

We use modern, ANSI-style function prototypes, with the type specifier on the same line as the function name. You may encounter other styles in the code, but we are transitioning to the new style as time permits.

Below is a correct example:

```
int main(int argc, char **argv, char **envp)
{

}
```

E.5 Brace Placement (if-for-while-do)

The `if` keyword should be followed by one space, then the opening paren and conditional expression. We also use Stroustrup-style `else` blocks, rather than the K&R 'cuddled' `else`:

```
if (x) {
...
}
else {
...
}

while (1) {
...
}

for (i = 1; i < 10; i++) {
...
}

do {
...
} while (x);
```

Single-statement if blocks should be isolated to a single line:

```
if (x) stmt();
```

not:

```
if(x)
    stmt();
```

Notice that there is a space between `if` and `(x)`, but not between `stmt` and `()`. This should be followed throughout the code.

If an `if` block has an `else if` or `else`, all parts of the construct must be bracketed, even if one or more of them contain only one statement:

```
if (x) {
    foo();
}
else if (y) {
    bar();
}
else {
    bas();
}
```

E.6 Labels and goto

Labels must begin in column 1, and have two lines of vertical space above and one beneath.

E.7 Preprocessor Conditionals

E.8 coding standards, preprocessor conditionals

I have struggled with this, but have settled upon the standard practice of keeping them in column 1.

E.9 Overall Program Spacing

- Variable declarations fall immediately beneath the opening curly brace, and should initialize the variable right there whenever initialization is used.
- One line between the last variable declaration and the first line of real code.
- The `return` statement of a function (when used as the last line of a function) should have one blank line above it and none below it.
- Really long functions (those whose entire body is longer than 24 lines) should have a comment immediately following the closing curly brace of the function, telling you what function the closing brace terminates.

E.10 The switch() Statement

We indent `case` one level beneath `switch()`, and the code within each `case` beneath the `case`. Each `case` should have one line of vertical whitespace above it:

```
switch(foo) {

    case some_const:
        foo();

        break;

    case some_other_const:
        bar();

        break;
```

```
    default:
        exit(1);

        break;
}
```

E.11 Comments

We use C-style comments (`/* comment */`) exclusively, even on single-line comments. C++ comments (`// comment`) are not permitted.

Appendix F Conformance Clause

Index

$

^

A

B

C

D

E

F

G

H

I

T

U

V

W

X

Z