The FreeM Manual

The Official Manual of FreeM
Version 0.64.0-rc1

Serena Willis

This manual is for FreeM, (version 0.64.0-rc1), which is a free software implementation of the M programming language.

Print-optimized versions of this book are typeset in *Computer Modern* by the author using the *GNU Texinfo* tools.

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 the Red Hat Linux 5 distribution of the GNU/Linux operating system. 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 Serena 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 application.

For more information on FreeM history, see `*$PREFIX*/share/freem/doc/freem_history.*` (distributed in PostScript, PDF, and plain text formats).

Production Readiness

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

- `Section 12.33 [VIEW], page 45` commands and `Section 9.25 [$VIEW()], page 24` functions are used extensively to configure and inspect the run-time behavior of FreeM, rather than the "canonical" SSVN-based approach.
- Server sockets are not yet implemented.
- There are some situations that can result in segmentation faults and/or lock-ups.
- In spite of our best efforts, this manual is not yet complete.

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)

- Rick Marshall (Testing, MDC 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 (GNU/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) +
- Serena Willis (Current Maintainer and Project Lead) +
- Steve Zeck (Code)

1 Document Conventions

1.1 Formatting Conventions

This manual uses the following formatting conventions:

- Code examples, filesystem paths, and commands are presented in `monospace`
- Placeholders where the reader is expected to supply a replacement value are presented in *`monospace italics`*, and depending on context, may be surrounded by angle brackets
- New terminology is introduced in *proportional italics*

1.2 Definitions

FreeM uses abbreviations for common language elements:

$PREFIX	Refers to the base filesystem location within which FreeM is installed. For most distribution methods of FreeM, *$PREFIX* represents either `/` or `/usr/local`.
dlabel	Refers to a label in an M routine, beginning in the first column of the line. Can be a *name* or an *intlit*.
entryref	Refers to an M routine entry point, denoted in the format *`dlabel`* `[+`*`intexpr`*`] [^`*`routine`*`]`.
expr	Refers to any expression. Often presented in the format *expr V* `<`*type*`>`, where *V* means *giving*; e.g., *expr V lvn* means *expression giving local variable name.*
glvn	Refers to the name of an M global, local, or structured system variable.
gvn	Refers to the name of an M global variable.
intexpr	Refers to an integer expression.
intlit	Refers to an integer literal.
ISV, *isv*	Refers to an M intrinsic special variable; `$JOB` and `$IO` are examples of ISVs.
L	Indicates a *list* of the following item, e.g., *L gvn* means *list of global variable names.*
lvn	Refers to the name of an M local variable.
postcondition	A *tvexpr* immediately following a command verb affecting that command's execution.
strlit	Refers to an M string literal.
ssvn	Refers to the name of an M structured system variable.
tvexpr	Refers to a truth-valued expression, i.e., an expression interpreted as a truth value.

2 A Note on Standards

FreeM attempts to implement as many features as possible from the M Development Committee's unpublished *Millennium Draft Standard*, as well as its predecessors.

The maintainer of FreeM (who is also the author of this book) is largely in favor of standardization efforts, and hopes that the MDC will resume activities, and will happily participate if it does so in an open, public, transparent, and democratic manner. Until then, however, FreeM will attempt to improve the M language, in cooperation with other free software M implementers where possible. Any breaking changes introduced in future MDC releases of the *Standard* (such as the rumored *M5*) which prove incompatible with FreeM will be handled via the `$DIALECT` special variable (to be changed to `$ZDIALECT` in a coming release).

The conformance document required per the *Standard* should be installed as a `man` page on any computer system where FreeM is made available. Simply type `man freem_conformance` to access.

3 FreeM Invocation

3.1 Synopsis

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

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

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

3.2 %SYSINIT Routine

The `%SYSINIT` routine runs every time a FreeM interpreter process starts. This routine defines some useful constants, enables handling of `TRIGGER` events, and handles the execution of code passed via the `-x|--execute` or routines passed via `-r|--routine`.

Do not modify the supplied `%SYSINIT` routine to add site-specific startup items. Instead, create a `LCLINIT` routine in the `USER` namespace of one or more environments. `%SYSINIT` will automatically run `LCLINIT` each time it starts.

3.3 Command-Line Options

`-d`, `--daemon`
: Starts the FreeM environment daemon, exactly one of which must be running at all times in order for FreeM interpreter and fmadm processes to function.

`-e`, `--environment`
: Selects the environment to be used. If no environment is specified, `DEFAULT` is used.

`-k`, `--nofork`
: When used with `-d` or `--daemon`, causes the FreeM environment daemon to run instead in the foreground. Useful for debugging.

`-S`, `--shmsize`
: When used with `-d` or `--daemon`, specifies the number of bytes of shared memory FreeM will allocate for the `LOCK` table, job table, and IPC table. This will determine the maximum number of concurrent FreeM processes and `LOCK`s available in this environment.

`-c`, `--config`
: Specify a configuration file other than `$PREFIX/etc/freem.conf`.

`-h`, `--help`
: Display a help message showing valid FreeM options.

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

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

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

`-r <entryref>`, `--routine=<entryref>`
: Causes `<entryref>` to be executed at load, instead of `%SYSINIT`.

`--standard=<standard>`
: Sets the default FreeM dialect to use for new routine buffers.

  Valid values for `<standard>` are as follows:

`M77`	Restricts FreeM to use only features specified by the 1977 M standard.
`M84`	Restricts FreeM to use only features specified by the 1984 M standard.
`M90`	Restricts FreeM to use only features specified by the 1990 M standard.
`M95`	Restricts FreeM to use only features specified by the 1995 M standard.
`MDS`	Restricts FreeM to use only features proposed by the Millennium Draft Standard.
`M5`	Restricts FreeM to use only features proposed by the upcoming M5 standard.
`FREEM, EXTENDED`	Removes all standards-based restrictions and allows full access to all FreeM features. This is the default value of `$DIALECT`.

  Please note that FreeM is not entirely standards-compliant, regardless of the value of `<standard>`.

`-v`, `--version`
: Displays FreeM version information.

`-x <mcode>`, `--execute=<mcode>`
: Executes M code `<mcode>` at startup.

3.4 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
```

4 The FreeM Environment Daemon

The FreeM environment daemon manages shared resources for a given FreeM environment. These include the lock table, job table, inter-process communication, and concurrency control for transaction processing. Unlike some M implementations, the FreeM environment daemon does *not* function as a write daemon for global storage.

One daemon process is required per FreeM environment, and can be started in the following ways, in order of preference:

```
$ sudo fmadm start environment [-e=<environment-name>]
```

```
$ freem --daemon [--nofork] [--environment=<environment-name>] [--shmsize=<bytes>]
```

If the daemon is started with `--nofork`, it will run in the foreground and its output will be reflected on the terminal. Otherwise, the daemon will run as a child process in the background and immediately return terminal control to the shell. The latter option is recommended in most cases.

The `--environment` option will start the daemon for the specified *environment-name*. The default environment, if unspecified, is called `DEFAULT`. If using an environment other than `DEFAULT`, interpreter processes that wish to also connect to the same environment must also use the `--environment` option when starting, and `libfreem` clients must also pass the environment name as the first argument to the `freem_init()` function. Environments allow you to run multiple, isolated instances of FreeM on the same machine, whose globals and routines are distinct and unique.

The `--shmsize` option specifies the size in bytes of the FreeM shared memory segment. The default is 4194304 bytes. Increasing the size of the FreeM shared memory segment will, at the cost of increased memory usage, increase the number of concurrent jobs and lock table entries available to the environment; decreasing the size of the segment will have the expected opposite effect. Note that you must also pass `--shmsize` with the same number of bytes to any interpreter process to be used with an environment whose daemon uses a non-default shared memory segment size.

Attempting to start a FreeM interpreter process without a daemon running with the same environment name will result in an error.

5 The FreeM Direct-Mode Environment

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

```
Coherent Logic Development FreeM version 0.64.0-rc1 (x86_64-pc-linux-gnu)
Copyright (C) 2014, 2020, 2021 Coherent Logic Development LLC


USER>
```

The prompt (`DEFAULT.USER>`) displays the current environment and namespace, `DEFAULT` and `USER`, respsectively. If any uncommitted direct-mode transactions have been started, the prompt will change to reflect the current value of `Section 8.18 [$TLEVEL], page 16`:

```
TL1:DEFAULT.USER>
```

In the above example, `TL1` indicates that `Section 8.18 [$TLEVEL], page 16` is currently *1*.

5.1 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:

```
DEFAULT.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
TRIGGER              Disabled        0
```

`trantab` Displays information about any uncommitted transactions currently in-flight for this process.

`trantab` Displays statistics about globals that have been opened in the current FreeM process.

`jobtab` Displays a summary of the FreeM job table.

`locktab` Displays a list of `LOCK`s held in the current environment.

`rbuf` Lists the status of all FreeM routine buffers.

`wh` Forces an immediate flush of this process's `readline` history buffer to disk.

`shmstat` Displays the configuration of FreeM shared memory. Intended only for advanced debugging of the FreeM environment.

`shmpages` Lists the status of each FreeM shared memory page. Intended only for advanced debugging of the FreeM environment.

`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.

`!!` Launches a subshell within the FreeM direct mode, allowing the user to run operating system commands.

```
DEFAULT.USER> !!

Type Ctrl-D to exit from the shell
$ uname -a
Linux hesperos 4.19.0-17-amd64 #1 SMP Debian 4.19.194-3 (2021-07-18) x86_64
$ exit

DEFAULT.USER>
```

`!<`*`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 `ASTART` and `ASTOP`.

If the `>` character is supplied immediately preceding <*external-command*>, FreeM will append the contents of an M local or global variable referenced in `^$JOB($JOB,"PIPE_GLVN")` to the standard input stream of <*external-command*>.

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 or global variable referenced by `^$JOB($JOB,"PIPE_GLVN")`.

The data value in the unsubscripted M local or global contains the number of lines in the input or output. Subscripts `(1)..(`*`n`*`)` contain the data for lines 1-*n*.

If you issue a `Section 12.14 [HALT], page 37` command at the direct-mode prompt, you will exit out of FreeM. However, if you issue a `Section 12.14 [HALT], page 37` command when `Section 8.18 [$TLEVEL], page 16` is greater than zero, you will be given the opportunity to commit or rollback any pending transactions:

```
DEFAULT.USER> TSTART


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

```
TL1:DEFAULT.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? ($


Transactions have been rolled back.
```

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

5.2 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:

```
DEFAULT.USER> S DENOM=10


DEFAULT.USER> 100/DENOM

10
DEFAULT.USER>
```

Such expressions will be immediately evaluated, and the result printed on `Section 8.7 [$IO], page 15`.

6 Debugging

6.1 Debugging Synopsis

FreeM includes an interactive debugger, entered using the `BREAK "DEBUG"` command. The debugger is also entered if `Ctrl-C` is pressed, `Ctrl-C` handling is enabled, and you are in direct mode.

If you would like to enter the debugger automatically each time an error is encountered, add the following to your `LCLINIT` routine:

```
     S $ETR="B ""DEBUG"""
```

6.2 Debugging Commands

The debugger uses its own unique command language, where M commands are unavailable. Commands are as follows:

`exit`, `quit`
: Exits the debugger and returns to direct mode or normal program execution.

`e` *glvn*, `examine` *glvn*
: Prints the value of *glvn* to the terminal.

`t`, `trace`
: Toggles *trace mode* on and off. When trace mode is on, FreeM will display information about each `DO` or `GOTO` command encountered, including the routine which invoked the branch, which type of branch was invoked, and the target of the branch.

`s`, `step`
: Single-steps through FreeM code command-by-command.

`n`, `next`
: Single-steps through FreeM code line-by-line.

`c`, `cont`, `continue`
: Resumes normal program execution, disabling single-step mode.

`bt`, `backtrace`
: Produces a stack trace.

`h`, `halt`
: Halts the process being debugged and returns control to the operating system.

`w [[+|-|?]<`*`glvn`*`>]`, `watch [[+|-|?]<`*`glvn`*`>]`
: With no arguments, toggles watchpoints on and off. With `+`, adds <*glvn*> to the watchlist. With `-`, removes <*glvn*> from the watchlist. With `?`, queries the watch status of <*glvn*>.

7 Directives

In FreeM, a directive is an instruction embedded in an M comment, and passed to the interpreter to affect a change that is specific to the current routine only.

The format of a directive is `;%<directive-name>`, where `<directive-name>` is one of the directives listed below.

7.1 %DIALECT

Sets the M dialect in effect for the current routine buffer; also sets the `$DIALECT` special variable to match. See also Section 8.2 [$DIALECT], page 14.

Syntax

```
;%DIALECT <dialect>
```

Valid values for `<dialect>` are as follows:

`M77`	Restricts FreeM to use only features specified by the 1977 M standard.
`M84`	Restricts FreeM to use only features specified by the 1984 M standard.
`M90`	Restricts FreeM to use only features specified by the 1990 M standard.
`M95`	Restricts FreeM to use only features specified by the 1995 M standard.
`MDS`	Restricts FreeM to use only features proposed by the Millennium Draft Standard.
`M5`	Restricts FreeM to use only features proposed by the upcoming M5 standard.
`FREEM, EXTENDED`	Removes all standards-based restrictions and allows full access to all FreeM features. This is the default value of `%DIALECT`.

Please note that FreeM is not entirely standards-compliant, regardless of the value of `%DIALECT`.

8 Intrinsic Special Variables

8.1 $DEVICE

Returns the status of the device currently in use.

If `$DEVICE` returns *1*, an error condition exists on the current device. In this case, there will be two additional fields separated by commas, indicating the internal FreeM error code representing the error present on the device and a text explanation of the error.

8.2 $DIALECT

Returns or sets the language dialect of the current routine.

Valid values for `$DIALECT` are as follows:

`M77`	Restricts FreeM to use only features specified by the 1977 M standard.
`M84`	Restricts FreeM to use only features specified by the 1984 M standard.
`M90`	Restricts FreeM to use only features specified by the 1990 M standard.
`M95`	Restricts FreeM to use only features specified by the 1995 M standard.
`MDS`	Restricts FreeM to use only features proposed by the Millennium Draft Standard.
`M5`	Restricts FreeM to use only features proposed by the upcoming M5 standard.
`FREEM, EXTENDED`	Removes all standards-based restrictions and allows full access to all FreeM features. This is the default value of `$DIALECT`.

Please note that FreeM is not entirely standards-compliant, regardless of the value of `$DIALECT`.

8.3 $ECODE

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

Writing a value in the format `,<error-code>,` into `$ECODE` will raise that error condition.

8.4 $ESTACK

Returns the depth of the program execution stack since the last time `$ESTACK` was `NEW`ed. `NEW`-able, but not `SET`-able. Differs from the `Section 8.14 [$STACK], page 15` ISV in that it is `Section 12.23 [NEW], page 40`-able, and resets to a value of 0 when `Section 12.23 [NEW], page 40`ed.

8.5 $ETRAP

Sets or retrieves the M code that is run when an error is encountered or `Section 8.3 [$ECODE], page 14` is set to a non-blank value. `$ETRAP` code executes when `$ECODE` becomes non-blank.

8.6 $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.

> *FreeM Extension*
>
> In FreeM, `$HOROLOG` is `Section 12.27 [SET], page 43`table. Setting `$HOROLOG` will set the system clock if your user account has the appropriate permissions. If your user account does not have permissions to modify the system clock, FreeM will raise a `ZPROTECT` error.

8.7 $IO

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

8.8 $JOB

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

8.9 $KEY

Represents the sequence of control characters that terminated the last `Section 12.26 [READ], page 42` command on `Section 8.7 [$IO], page 15`.

8.10 $PDISPLAY

Represents the current principal display for M Windowing API operations. Commonly used as an index into the `Section 13.3 [^$DISPLAY], page 57` structured system variable.

8.11 $PRINCIPAL

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

8.12 $REFERENCE

Returns the last *glvn* referenced. Can be `Section 12.27 [SET], page 43`, and also stacked with `Section 12.23 [NEW], page 40`.

8.13 $QUIT

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

When `$QUIT` returns *1*, a subsequent `Section 12.25 [QUIT], page 42` command must have an argument.

8.14 $STACK

Represents the current stack level.

8.15 $STORAGE

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

8.16 $SYSTEM

Returns the MDC system ID of FreeM, as well as the environment ID of the current environment.

8.17 $TEST

`$TEST` is a writable, `Section 12.23 [NEW], page 40`-able ISV that is *1* if the most recently evaluated expression was *true*. Otherwise, returns *0*.

`$TEST` is implicitly `NEW`ed when entering a new stack frame for extrinsic functions and argumentless `Section 12.10 [DO], page 34`. `$TEST` is *not* implicitly `NEW`ed when a new stack frame is entered with an argumented `DO`.

For single-line `Section 12.16 [IF], page 37` or `Section 12.11 [ELSE], page 34` expressions, you may use `Section 12.29 [THEN], page 43` to stack `$TEST` until the end of the line. All new code should employ `THEN` in this manner, as stacking `$TEST` prevents a wide range of coding errors that can be very challenging to detect and eliminate.

8.18 $TLEVEL

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

8.19 $WITH

Returns the variable prefix set by the `Section 12.54 [ZWITH], page 53` command.

8.20 $X

Represents the current column position of the FreeM cursor.

> In FreeM, setting `$X` will move the FreeM cursor.

8.21 $Y

Represents the current row position of the FreeM cursor.

> In FreeM, setting `$Y` will move the FreeM cursor.

8.22 $ZA

On the `HOME` device, always `0`. On other devices, returns the current position of the file opened on I/O channel `Section 8.7 [$IO], page 15`.

8.23 $ZB

Represents the last keystroke.

8.24 $ZCONTROLC

Returns the status of the Ctrl-C flag and resets it to *false*.

8.25 $ZDATE

Returns the current date, in the preferred representation for the current system locale.

8.26 $ZERROR

Returns the last error message.

8.27 $ZHOROLOG

Output `Section 8.6 [$HOROLOG], page 15`-style time, with the addition of milliseconds.

8.28 $ZINRPT

Gets or sets the interrupt enable/disable flag.

8.29 $ZJOB

Returns the `Section 8.8 [$JOB], page 15` value of the parent process if the current process was started by a `Section 12.17 [JOB], page 38` command. Otherwise, returns an empty string.

8.30 $ZLOCAL

Returns the last local variable referenced.

8.31 $ZNAME

Returns the name of the current routine.

8.32 $ZPRECISION

Gets or sets the number of digits of numeric precision used for fixed-point decimal arithmetic. If `^$JOB($JOB,"MATH")` is `IEEE754`, `$ZPRECISION` defaults to 16 digits, with a maximum of 16 digits. If `^$JOB($JOB,"MATH")` is `FIXED`, `$ZPRECISION` defaults to 100 digits, with a maximum of 20,000 digits.

See Section 13.6 [^$JOB], page 58.

8.33 $ZREFERENCE

Returns the last *gvn* referenced.

8.34 $ZSYSTEM

Represents the return value of the last external command run with `!`.

8.35 $ZTIME

Returns the system time in the preferred representation for the current system locale.

8.36 $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 `Section 12.13 [GOTO], page 36` 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.

8.37 $ZUT

Returns the number of microseconds elapsed since the UNIX epoch (Jan 1, 1970 0:00:00).

8.38 $ZVERSION

Returns the version of FreeM in use, as well as the GNU host triplet for the current FreeM build.

See *https://wiki.osdev.org/Target_Triplet*.

9 Intrinsic Functions

9.1 $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*>.

9.2 $CHAR

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

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

9.3 $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
```

9.4 $EXTRACT

Extracts a substring of a string.

The first argument is the source string.

The optional second argument specifies the starting position of the substring to extract, and defaults to `1`.

The optional third argument specifies the ending position of the substring to extract, and defaults to the value of the second argument, or `1`.

This example will extract the string *FreeM* into the local variable `M`.

```
SET NAME="FreeM is the best!"
SET M=$EXTRACT(NAME,1,5)
```

It is also possible to use `$EXTRACT` on the left-hand side of a `SET` assignment in order to modify a substring:

```
DEFAULT.USER> SET FOO="ABCDEFG"


DEFAULT.USER> SET $EXTRACT(FOO,1,3)="XYZ"
DEFAULT.USER> WRITE FOO

XYZDEFG
```

9.5 $FIND

Finds the character immediately following the first occurence of a substring within a string.

The first argument is the source string.

The second argument is the substring to be located.

The optional third argument indicates the position within the source string at which to begin searching.

9.6 $FNUMBER

Formats a number according to a particular set of formatting codes.

The first argument is the number to format.

The second argument is the series of formatting codes:

'P' or 'p'	Will display negative numbers within parentheses instead of showing a minus sign.
, (comma)	Will add commas as thousands separators.
+	Will include a plus sign for positive numbers. Not compatible with 'P' or 'p'.
-	Will remove the minus sign from negative numbers. Not compatible with 'p' or 'P'.
't' or 'T'	Will place the sign after the number instead of before the number.

The optional third argument is a number indicating how many digits to which the fractional part of the number will be zero-padded.

9.7 $GET

Returns the value of a local, global, or SSVN if the specified item is defined, or a default value otherwise.

The first argument is the local, global, or SSVN to be examined.

The optional second argument is the default value to be returned if the referenced item is undefined, and defaults to the empty string.

9.8 $INSTANCEOF

Returns `1` if the specified *lvn* is an instance of class *class*, or `0` otherwise.

The first argument is a string representing a valid FreeM local variable.

The second argument is a string representing a valid FreeM class.

```
DEFAULT.USER> N STR=$$^%STRING

DEFAULT.USER> W $INSTANCEOF("STR","^%STRING")
1
```

9.9 $JUSTIFY

Right-justifies a string based on a specified fixed length.

The first argument is the source string.

The second argument is the character length of the output.

The optional third argument controls the number of fractional digits to be included in the output, and defaults to the number of digits specified in the first argument.

9.10 $LENGTH

Returns the length of a string, or the number of items in a list delimited by a specified character (as used by `Section 9.14 [$PIECE()], page 22`).

The first argument is the source string.

The optional second argument is the list delimiter to be used. When this argument is omitted, the length of the string in characters is returned.

9.11 $NAME

Returns the canonical name reference along with some or all of its subscripts.

The first argument is the source name.

The optional second argument indicates the maximum subscript count to be returned, and defaults to the subscript count of the source name.

9.12 $NEXT

Deprecated. Use `$ORDER` instead. Returns the next numeric subscript of the specified glvn.

Syntax

```
$NEXT(glvn)
```

Example

Assume the following array:

```
^foo(1)=""
^foo(2)=""
```

And the following code:

```
 W $ZNEXT(^foo(1)) ; => 2
```

9.13 $ORDER

Returns the previous subscript or next subscript in a local, global, or a subset of structured system variables.

The first argument is the subscripted local, global, or SSVN.

The optional second argument can be `1` to retrieve the next subscript, or `-1` to return the previous.

9.14 $PIECE

Syntax

`$PIECE(`*`s`*`,`*`d`*`[,`*`n`*`[,`*`end`*`]])`

Accesses the `n`th through `end` `d`-delimited pieces of string `s`.

The first argument is the string to be evaluated.

The second argument is the delimiter to be used.

The optional third argument is the first `d`-delimited piece to access, and defaults to `1`.

The optional fourth argument is the final `d`-delimited piece to access, and defaults to the value of the third argument (`n`).

Can be used on the left-hand side of an expression in order to `Section 12.27 [SET], page 43` a value into a `d`-delimited piece of `s`, as in:

```
; ^snw="this^is^a^piece"
SET $PIECE(^snw,"^",2)="isn't" ; => "this^isn't^a^piece"
```

9.15 $QLENGTH

Syntax

```
$QLENGTH(expr V glvn)
```

Returns the number of subscripts in *glvn*.

Example

```
SET SUBCT=$QLENGTH("^GBL(1,2,3)") ; => 3
```

9.16 $QSUBSCRIPT

Syntax

```
$QSUBSCRIPT(expr V glvn,expr V n)
```

In the RHS form, returns the *n*th subscript of *glvn*.

Example

```
SET SUB=$QSUBSCRIPT("^GBL(1,2,3)",2) ; => 2
```

Syntax

```
SET $QSUBSCRIPT(expr V glvn,expr V n)=expr ; => ^GBL(1,4,3)
```

In the LHS form, sets the *n*th subscript of *glvn* to *expr*.

9.17 $QUERY

Returns the next subscripted reference in a global.

Syntax

```
$QUERY(glvn)
```

Example

We will assume the following data structure exists:

```
^snw(1)=1
^snw(1,2)="foo"
```

```
^snw(2)=3
^snw(3)=""
```

The following code will retrieve the next subscripted name after `^snw(1)`:

```
SET NEXTNAM=$QUERY(^snw(1)) ; => ^snw(1,2)
```

9.18 $RANDOM

Syntax

```
$RANDOM(max)
```

Returns a pseudo-random integer in the range of `0..max - 1`

9.19 $REVERSE

Syntax

```
$REVERSE(s)
```

Returns the reverse of string *s*.

Example

```
SET FOO=$REVERSE("ABC") ; => CBA
```

9.20 $SELECT

Returns a value corresponding to the first true condition in a list of conditional expressions. Each argument is an expression, followed by a colon, followed by an expression whose value will be returned if the first expression is true. If no expressions are true, error condition `M4` is raised.

Example

```
SET FOO=$SELECT(1=2:"math is broken",1=1:"the world makes sense") ; => "the world make
```

9.21 $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*.

9.22 $TEXT

Returns a line of code from a routine.

9.23 $TRANSLATE

Replaces characters in a string.

The first argument is a string expression representing the text to be changed.

The second argument is a list of characters to replace.

The third argument is a list of characters to use as the replacements for the characters in the second argument.

Example

```
DEFAULT.USER> W $TRANSLATE("twig","wt","rb")
brig
```

9.24 $TYPE

Returns a string giving the class of the object specified in the parameter.

See Chapter 25 [Object-Oriented Programming], page 83,

9.25 $VIEW

9.26 $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*

9.27 $ZCALL

Purpose unknown.

9.28 $ZCRC

Returns a checksum of `arg1`.

Syntax

```
$ZCRC(arg1)
SET VAR=$ZCRC("MUMPS") ; => 86
```

9.29 $ZDATA

Purpose unknown.

9.30 $ZDATE

Converts a `Section 8.6 [$HOROLOG], page 15` string into a human-readable date.

Syntax

```
    SET VAR=$ZDATE($H[,<format-string>])
```

The optional <*format-string*> follows the same rules as the UNIX `strftime` function. If <*format-string*> is omitted, the value of `^$SYSTEM("ZDATE_FORMAT")` is used (typically `%x`). See Section 13.10 [^$SYSTEM], page 63,

9.31 $ZEDIT

Purpose unknown.

9.32 $ZHOROLOG

Converts date and/or time values producible by `Section 9.30 [$ZDATE()], page 25` or `Section 9.44 [$ZTIME()], page 27` to `Section 8.6 [$HOROLOG], page 15` format.

Syntax

```
    $ZHOROLOG(<date-value>,<format-string>)
```

<*date-value*> is a date or time string compatible with the formats from `Section 9.30 [$ZDATE()], page 25` or `Section 8.35 [$ZTIME], page 18`.

<*format-string*> is a format string of the same format as used by the `strptime(3)` UNIX function.

9.33 $ZKEY

Purpose unknown.

9.34 $ZLENGTH

Purpose unknown.

9.35 $ZLSD

Returns the Levenshtein distance between two arguments. The Levenshtein distance represents the minimum number of edits needed to change the first argument into the second argument.

Syntax

`SET VAR=$ZLSD(arg1,arg2)`

Example

`SET VAR=$ZLSD("KITTENS","MITTENS") ; => 1`

9.36 $ZM

Purpose unknown.

9.37 $ZNAME

Purpose unknown.

This function relies on the value of `$VIEW(71)` being `0` (this is not the default).

9.38 $ZNEXT

Returns a fully-formed variable reference of the next numeric subscript of the specified glvn.

Syntax

```
$ZNEXT(glvn)
```

Example

Assume the following array:

```
^foo(1)=""
^foo(2)=""
```

And the following code:

```
 W $ZNEXT(^foo(1)) ; => ^foo(2)
```

This function relies on the value of `$VIEW(71)` being `1` (this is the default).

9.39 $ZORDER

Purpose unknown.

9.40 $ZPIECE

Purpose unknown.

9.41 $ZPREVIOUS

Purpose unknown.

9.42 $ZREPLACE

Replaces all instances of `arg2` with `arg3` in string `arg1`.

Syntax `$ZREPLACE(arg1,arg2,arg3)`

Example

```
SET VAR=$ZREPLACE("CAT","C","B") ; => BAT
```

9.43 $ZSYNTAX

`$ZSYNTAX` performs a very basic syntax check on *expr V mcode*. Checks only for illegal commands, mismatched brackets, mismatched quotes, missing or surplus arguments, or surplus commas.

Syntax

```
    $ZSYNTAX(expr V mcode)
```

If no syntax error is found, returns the empty string.

If a syntax error is found, returns a number indicating the position in *expr V mcode* at which the error was found, followed by a comma, and the FreeM error code that was found.

9.44 $ZTIME

Converts a `Section 8.6 [$HOROLOG], page 15` string into a human-readable time.

Syntax

```
    SET VAR=$ZTIME($H[,<format-string>])
```

The optional <*format-string*> follows the same rules as the UNIX `strftime(3)` function. If <*format-string*> is omitted, the value of `^$SYSTEM("ZTIME_FORMAT")` is used (typically `%X`).

10 OBJECT Methods

These methods are part of the `^%OBJECT` class, from which all FreeM objects ultimately inherit.

Please note that classes may override `^%OBJECT` methods (or methods of any class) in order to provide results more fitting to the class's abstraction goals.

10.1 $$TONUMBER

Returns (when applicable) a canonical numeric representation of the referenced object.

Syntax

```
     W $$MYOBJECT.TONUMBER(),!
```

If no canonical numeric representation of the object is possible, will return the empty string.

10.2 $$TYPE

Returns the fully-qualified class of the referenced object.

Syntax

```
     W $$MYOBJECT.TYPE()
```

Note that M variables that are created by non-object-oriented means will be objects of the `^%STRING` class.

10.3 $$VALUE

Returns the value of the referenced object.

Syntax

```
     W $$MYOBJECT.VALUE()
```

11 STRING Methods

These are methods inherent to the `^%STRING` class, which is the default class for M variables created without specifying a class.

11.1 $$ASCII

Returns the ASCII code of a character within the string. See Section 9.1 [$ASCII()], page 19.

Syntax

```
W $$MYOBJECT.ASCII(3)
```

The above example returns the ASCII code in position 3 of string object `MYOBJECT`.

11.2 $$DATA

Returns the value of the `$DATA` intrinsic function as performed on the value of the object. See Section 9.3 [$DATA()], page 19.

Syntax

```
W $$MYOBJECT.DATA()
```

11.3 $$DISTANCE

Returns the Levenstein distance between the string and another string. See Section 9.35 [$ZLSD()], page 26.

Syntax

```
W $$MYOBJECT.DISTANCE("someString")
```

11.4 $$EXTRACT

Returns a substring of the string. See Section 9.4 [$EXTRACT()], page 19.

Syntax

```
$$<objectName>.EXTRACT(<start>,<end>)
```

11.5 $$FIND

Finds the character immediately following the first occurence of a substring within a string.

The first argument is the substring to be located.

The second argument is the position within the string at which to begin searching.

See Section 9.5 [$FIND()], page 20.

11.6 $$FNUMBER

Formats a number according to a set of formatting codes.

The argument is a series of formatting codes. See Section 9.6 [$FNUMBER()], page 20, for details.

11.7 $$JUSTIFY

Right-justifies a string based on a specified fixed length.

The first argument is the character length of the output.

The second argument controls the number of fractional digits to be included in the output, and defaults to the number of digits specified in the first argument.

See Section 9.9 [$JUSTIFY()], page 21, for details.

11.8 $$LENGTH

Returns the length of the string.

11.9 $$PIECECOUNT

Returns the number of items in a list delimited by the character specified in the argument.

11.10 $$PIECE

Syntax

`$PIECE(d[,n[,end]])`

Accesses the `n`th through `end` `d`-delimited pieces of the string.

The first argument is the delimiter to be used.

The optional second argument is the first `d`-delimited piece to access, and defaults to `1`.

The optional third argument is the final `d`-delimited piece to access, and defaults to the value of the third argument (`n`).

11.11 $$REPLACE

Syntax `myString.$$REPLACE(arg1,arg2)`

Replaces all instances of `arg2` with `arg3` in `myString`.

11.12 $$REVERSE

Returns the reverse of the string.

11.13 $$TOLOWER

Returns an all-lowercase version of the string.

11.14 $$TOUPPER

Returns an all-uppercase version of the string.

11.15 $$TRANSLATE

Identical to Section 9.23 [$TRANSLATE()], page 24, except that the arguments are shifted left by one, and the input string is implicit (the object).

12 Commands

12.1 @

Executes FreeM code *expr V mcode*.

Syntax

```
@expr V mcode
```

Example (Using Variable)

```
DEFAULT.USER> SET FOO="WRITE ""HELLO WORLD"",!"
DEFAULT.USER> @FOO

HELLO WORLD

DEFAULT.USER>
```

Example (Using String Literal)

```
DEFAULT.USER> @"WRITE ""HELLO WORLD"",!"

HELLO WORLD

DEFAULT.USER>
```

Example (Using Indirection)

```
DEFAULT.USER> SET FOO="BAR"

DEFAULT.USER> SET BAR="WRITE ""HELLO WORLD"",!"

DEFAULT.USER> @@FOO

HELLO WORLD

DEFAULT.USER>
```

12.2 !

FreeM Extension

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*.

12.3 !!

FreeM Extension

Launches a subshell within the FreeM direct mode, allowing the user to run operating system commands.

```
DEFAULT.USER> !!

Type Ctrl-D to exit from the shell
$ uname -a
Linux hesperos 4.19.0-17-amd64 #1 SMP Debian 4.19.194-3 (2021-07-18) x86_64 GNU/Linux
$ exit

DEFAULT.USER>
```

12.4 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.

12.5 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.

12.6 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.

12.7 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.

12.8 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
```

FreeM Extension

In its single-argument form, `BREAK` enters the interactive debugger or 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*

`"DEBUG"`	Enters the interactive debugger

`0`	Disables *Ctrl-C* handling
`-2`	Enables normal FreeM error handling
`2`	Enables *Digital Standard MUMPS* v2 error handling
`any integer value other than 0, 2, or -2`	Enables *Ctrl-C* handling

12.9 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.

12.10 DO

In its inclusive form, transfers program control to one or more specified subroutines, provided the optional *postcondition* evaluates to *true* or is omitted. Line levels of entryrefs specified in the argument list must be one, or error `M14` is raised.

Syntax

```
DO[:postcondition] entryref[:postcondition[,...]]
```

Non-Standard Behavior

FreeM allows `DO` *entryref*s to follow the format of *`+intexpr`*. In this case, the value of *intexpr* will be interpreted as an offset from the first line of the current routine.

In its argumentless form, transfers control to the following block of code where the line level is one greater than the level at which `DO` was encountered, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
DO[:postcondition]
```

12.11 ELSE

Executes the remainder of the line of code on which `ELSE` is encountered only if `$TEST` evaluates to *false*, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
ELSE[:postcondition]
```

Non-Standard Behavior

FreeM allows a *postcondition* on `ELSE`. While explicitly forbidden in the *Standard*, it was decided that FreeM should allow postconditions everywhere, both for the sake of foolish consistency (the likes of which Emerson warned against), and for the benefit of entrants to a hypothetical future obfuscated M contest, and those with a Machiavellian predisposition to wicked perversions and undue cleverness.

Using postconditions on `ELSE` should be strictly avoided in production code, as they have no practical use, and may contribute to technical debt, hardening of the arteries, hobgoblins, a small mind, a surfeit of logic, climate change, *Daily WTF* rants, or the meltdown of global financial markets.

12.12 FOR

In its argumentless form, repeatedly executes the remainder of the line on which `FOR` was encountered until a `QUIT`, `GOTO`, or end-of-line is encountered, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
FOR[:postcondition]
```

Non-Standard Behavior

When `$DIALECT` is set to `FREEM`, FreeM allows a *postcondition* on `FOR`. Much like postconditions on `ELSE` and `IF`, this is explicitly forbidden in the *standard*. The expression contained in the *postcondition* is evaluated on each iteration of the `FOR` loop, and if it does not evaluate *true*, the loop will be immediately exited. The effect is roughly similar to `WHILE` constructs present in other languages, but absent from standard M.

As with all non-standard features of FreeM, please exercise caution when using this feature, especially in code that is expected to run in other, less preternaturally-inclined M implementations.

In its sentinel form, repeatedly executes the remainder of the line and sets a sentinel variable on each iteration, provided the optional *postcondition* evaluates to *true* or is omitted.

On the first iteration of the loop, *glvn* will be set to *initalizer-expression*. On each subsequent iteration, *glvn* will be incremented by *increment-expression*, and the loop will terminate when *glvn* meets or exceeds the value of *max-expression*.

Syntax

```
FOR[:postcondition] glvn=initializer-expression:increment-expression:max-
expression
```

Example

```
DEFAULT.USER> FOR I=1:1:10 WRITE I,!

1
2
```

```
3
4
5
6
7
8
9
10

DEFAULT.USER> FOR I=2:2:10 WRITE I,!

2
4
6
8
10
```

In its explicit parameter form, a variable is set to each of a series of explicit values, once per iteration, provided the optional *postcondition* evaluates to *true* or is omitted. The loop terminates when no more values are available.

Syntax

```
FOR[:postcondition] glvn=expr1[,..exprN]
```

Example

```
DEFAULT.USER> FOR I=60,"FOO",-3,"George",1450,$HOROLOG WRITE I,!

60
FOO
-3
George
1450
66106,52388
```

12.13 GOTO

Transfers program execution to another line of code, provided the optional *postcondition* evaluates to *true* or is omitted. Attempting to `GOTO` a different line level or a different block when the line level of `GOTO` is greater than one will raise error `M45`.

Syntax

```
GOTO[:postcondition] entryref
```

Non-Standard Behavior

FreeM allows `GOTO` *entryref*s to follow the format of `+intexpr`. In this case, the value of *intexpr* will be interpreted as an offset from the first line of the current routine.

12.14 HALT

Halts program execution and frees resources allocated during execution, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
HALT[:postcondition]
```

12.15 HANG

Temporarily suspends the program for *expr* seconds, provided the optional *postcondition* evaluates to *true* or is omitted. Values of *expr* that are zero or less than zero are ignored.

Syntax

```
HANG[:postcondition] expr
```

Non-Standard Behavior

FreeM supports sub-second values for *expr*.

12.16 IF

In its argumented form, allows the remainder of the line of code following `IF` to execute only if all *tvexpr*s evaluate to *true*, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
IF[:postcondition] tvexpr[,...tvexpr]
```

In its argumentless form, allows the remainder of the line of code following `IF` to execute only if `$TEST` evaluates to *1*, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
IF[:postcondition]  command...
```

Style Recommendation

In the interest of readability and maintainability, we recommend avoiding the argumentless form of `IF` in new code. It is an obsolete relic of an era when routine sizes were severely limited, and can be difficult to spot, as the use of whitespace (`IF` ***`command`***) makes the intent of its use non-obvious at a glance. It is also far too easy to inadvertently delete the extra space, leading to program errors easily avoided otherwise.

We recommend explicitly checking the value of `$TEST` instead, as in `IF $TEST` ***`command`*** or ***`command`***`:$TEST ...`, as this makes the intent immediately clear both to M newcomers and seasoned experts, and sacrifices nothing of value, even on the oldest computer systems where FreeM can be used today.

12.17 JOB

Executes *entryref* in a separate process, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
JOB[:postcondition] entryref[:job-parameters[:timeout]]
```

If *timeout* is supplied, FreeM will set `$TEST` to *1* if the child process completes within *timeout* seconds.

12.18 KILL

In its inclusive form, `KILL` deletes the specified *glvn*s and their descendant subscripts, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
KILL[:postcondition] glvn[,...glvn]
```

In its exclusive form, `KILL` deletes all local variables *except* for those specified by *lvn*, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
KILL[:postcondition] (lvn[,...lvn])
```

In its argumentless form, `KILL` deletes all local variables, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
KILL[:postcondition]
```

12.19 KSUBSCRIPTS

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

Syntax

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

In the above *inclusive* form, `KVALUE` will kill the descendant subscripts at each local, global, or SSVN 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 0.64.0-rc1, 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.

12.20 KVALUE

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

Syntax

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

In the above *inclusive* form, `KVALUE` will kill the data values at each local, global, or SSVN 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 0.64.0-rc1, 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.

12.21 LOCK

Acquires or releases ownership of names.

In its argumentless form, `LOCK` releases ownership of all names previously locked by the current process, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
LOCK[:postcondition]
```

In its incremental form, increments or decrements the lock counter for each specified *name*, provided the optional *postcondition* evaluates to *true* or is omitted. Ownership of each *name* is considered to be the current process as long as the lock counter for *name* is greater than zero. If *timeout* is specified, FreeM will wait no more than *timeout* seconds in attempting to acquire ownership of *name*.

If `LOCK` succeeds within *timeout*, `$TEST` is set to *1*. Otherwise, `$TEST` is set to *0*.

Syntax

```
LOCK[:postcondition] [+|-]name[:timeout][,...[+|-]name[:timeout]]
```

Example

This example will increment the lock counter for `^SNW` and decrement the lock counter for `^MJR`.

```
LOCK +^SNW,-^MJR
```

In its non-incremental form, `LOCK` releases all `LOCK`s held by the current process, and then attempts to acquire a lock on each *name*, provided the optional *postcondition* evaluates to *true* or is omitted. If *timeout* is supplied, FreeM will attempt to lock *name* for no more than *timeout* seconds.

If `LOCK` succeeds within *timeout*, `$TEST` is set to *1*. Otherwise, `$TEST` is set to *0*.

Syntax

```
LOCK[:postcondition] name[:timeout][,...name[:timeout]]
```

12.22 MERGE

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

Syntax

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

The above example will merge the `^$JOB` SSVN 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.

12.23 NEW

In all forms of `NEW`, *name* must be a local variable name or `NEW`-able structured or intrinsic system variable.

In its inclusive form, `NEW` saves each specified *name* on the process stack and removes it, provided the optional *postcondition* evaluates to *true* or is omitted. When the current stack frame is exited, the previous values are restored.

Syntax

```
NEW[:postcondition] name[,...name]
```

In its exclusive form, `NEW` saves all local variables *except* those named (each *name*) and removes them, provided the optional *postcondition* evaluates to *true* or is omitted. When the current stack frame is exited, the previous values are restored.

Syntax

```
NEW[:postcondition] (name[,...name])
```

In its argumentless form, `NEW` saves all local variables and removes them, provided the optional *postcondition* evaluates to *true* or is omitted. When the current stack frame is exited, the previous values are restored.

Syntax

```
NEW:postcondition name=expr
```

In its initializing form, `NEW` stacks variable *name* and sets its value to *expr*, provided the optional *postcondition* evaluates to *true* or is omitted. When the current stack frame is exited, the previous value is restored.

Syntax

```
NEW:postcondition name=$%^CLASS(initializer-list)
```

In its object-oriented form, `NEW` creates an instance of class *^CLASS* in local variable *name* and calls the constructor of *^CLASS*, passing *initializer-list* as its argument(s).

12.24 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:connectio
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.

12.25 QUIT

`QUIT` will end execution of the current process level, optionally returning *expr*, provided the optional *postcondition* evaluates to *true* or is omitted.

`QUIT` with *expr* when an argument is not expected will raise error `M16`; `QUIT` without *expr* when an argument is expected will raise error `M17`.

Argumentless `QUIT` may also be used to exit a `FOR` loop occurring on the same line.

Syntax

```
QUIT[:postcondition] [expr]
```

12.26 READ

The `READ` command takes input from I/O channel `$IO` and stores it into specified variables, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
READ[:postcondition] read-argument[,...read-argument]
```

Each *read-argument* may be one of the following:

String Literal
: String literal *read-argument*s will be output to `$IO` unmodified.

Format Specifier
: One or more of the following:

  `!` (newline)
  : Advances the cursor down by one line and returns it to the first column.

  `#` (form-feed)
  : Advances the screen down by `$ZROWS` and moves the cursor to the upper-left corner of the screen.

  `?n` (position)
  : Advances the cursor and `$X` forward to position *n*.

Single-Character Read (`*variable-name[:timeout]`)
: Reads one character into variable *variable-name*. If the optional *timeout* is specified, will wait *timeout* seconds to retrieve one character. If a character is read within *timeout* seconds, `$TEST` will be set to *1*. If no character is read within *timeout* seconds, `$TEST` will be set to *0*.

Variable-Length Character Read (`variable-name[:timeout]`)
: Reads characters into *variable-name* until the character or character pair in `^$DEVICE(io-channel,"OPTIONS","TERMINATOR")` is encountered. If the optional *timeout* is specified, will wait *timeout* seconds to retrieve characters. If

characters are read within *timeout* seconds, `$TEST` will be set to *1*. If no character is read within *timeout* seconds, `$TEST` will be set to *0*.

Fixed-Length Character Read (`variable-name#count[:timeout]`)
: Reads *count* characters into *variable-name*. If the optional *timeout* is specified, will wait *timeout* seconds to retrieve characters. If characters are read within *timeout* seconds, `$TEST` will be set to *1*. If no character is read within *timeout* seconds, `$TEST` will be set to *0*.

Control Mnemonic (`/control-mnemonic[(arg1[,...argN])]`)
: Outputs X3.64 control mnemonic *control-mnemonic* to `$IO`. Please see the appendix on X3.64 Control Mnemonics for more information.

12.27 SET

The `SET` command places values into one or more variables, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
SET[:postcondition] set-argument[=expression | postfix-operator][,...set-
argument[=expression | postfix-operator]]
```

Each *set-argument* can be:

variable-name
: A local variable, global variable, writable intrinsic special variable, or writable structured system variable.

lhs-function
: `$EXTRACT` or `$PIECE`.

If any grouping of *set-argument*s is surrounded by parentheses, all *set-argument*s in the parenthesized group will be set to the result of *expression*.

If *postfix-operator* is used instead of `=expression`, the results of applying *postfix-operator* to the *set-argument* will be stored in *set-argument*. *postfix-operator* may not be used following a parenthesized group of *set-argument*s.

Example (postfix-operator)

```
SET A++,B-- ; increments A, decrements B
```

12.28 TCOMMIT

Commits all pending transactions to the data files, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
TCOMMIT[:postcondition]
```

12.29 THEN

Saves the value of `$TEST` until the end of the current line, restoring it at the end of the current line or when a `QUIT` is encountered. `THEN` should be used in all new code in conjunction with `IF`.

Example

```
IF 1 THEN  WRITE "HELLO!",!
```

12.30 TROLLBACK

Rolls back all pending transactions for the current process, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
TROLLBACK[:postcondition]
```

12.31 TSTART

Introduces a new transaction level, incrementing `$TLEVEL`, provided the optional *postcondition* evaluates to *true* or is omitted. Any global data file operations encountered when `$TLEVEL` is greater than zero will not be committed to the global data files until `TCOMMIT` is encountered.

If a transaction is restartable, variables in the *variables-list* will be restored to their original values on a restart of the transaction.

Syntax

```
TSTART[:postcondition] <variables-list>:<transaction-parameters>
```

<*variables-list*> can be:

`()`	Do not save off any local variables. Makes the transaction non-restartable.
`*`	Save off all local variables. Makes the transaction restartable.
`variableName`	Saves off only one local variable, *variableName*. Makes the transaction restartable.
`(variableName1,...,variableNameN)`	Saves off all local variables listed. Makes the transaction restartable.

<*transaction-parameters*> can be:

`S[ERIAL]`	Forces ACID properties on the transaction. When `SERIAL` is not selected, transactions occur in batch mode, and no attempt is made to guarantee ACID properties.
`T[RANSACTIONID]=transaction-id`	Sets the ID of the transaction to *transaction-id*

If you are using more than one transaction parameter, surround all of them in parentheses and separate them with commas, e.g.:

```
TSTART (FOO,BAR):(SERIAL,TRANSACTIONID="FOO")
```

12.32 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.

12.33 VIEW

Provides write access to various FreeM internal parameters, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
VIEW[:postcondition] view-number[:view-argument[:view-argument...]]
```

The *view-number* argument can be one of the following:

`21` - Close All Globals

Closes all global data files open in the current process. Takes no arguments.

Syntax

```
VIEW 21
```

`52` - Set G0 Input Translation Table for `$IO`

Syntax

```
VIEW 52:expr V trantab
```

`53` - Set G0 Output Translation Table for `$IO`

Syntax

```
VIEW 53:expr V trantab
```

`54` - Set G1 Input Translation Table for `$IO`

Syntax

```
VIEW 54:expr V trantab
```

`55` - Set G1 Output Translation Table for `$IO`

Syntax

```
VIEW 55:expr V trantab
```

`62` - Set `$RANDOM` Seed Number

Sets the seed number used by `$RANDOM` to *numexpr*.

Syntax

```
VIEW 62:numexpr
```

`63` - Set `$RANDOM` Parameter A

Sets the number used for `$RANDOM` Parameter A to *numexpr*.

Syntax

```
VIEW 63:numexpr
```

`64` - Set `$RANDOM` Parameter B

Sets the number used for `$RANDOM` Parameter B to *numexpr*.

Syntax

```
VIEW 64:numexpr
```

`65` - Set `$RANDOM` Parameter C

Sets the number used for `$RANDOM` Parameter C to *numexpr*.

Syntax

```
VIEW 65:numexpr
```

`66` - Set or Clear `SIGTERM` Handling Flag

Enables or disables handling of `SIGTERM` UNIX signals. If *tvexpr* evaluates to 1 (*true*), `SIGTERM` handling will be enabled. Otherwise, `SIGTERM` handling will be disabled.

Syntax

```
VIEW 66:tvexpr
```

`67` - Set or Clear `SIGHUP` Handling Flag

Enables or disables handling of `SIGHUP` UNIX signals. If *tvexpr* evaluates to 1 (*true*), `SIGHUP` handling will be enabled. Otherwise, `SIGHUP` handling will be disabled.

Syntax

```
VIEW 67:tvexpr
```

`70` - Set `$ZSORT/$ZSYNTAX` Flag

Selects whether `$ZS` resolves to `$ZSORT` or `$ZSYNTAX`.

If *tvexpr* evaluates to *true*, selects `$ZSYNTAX`. Otherwise, selects `$ZSORT`.

Syntax

```
VIEW 70:tvexpr
```

`71` - Set `$ZNEXT/$ZNAME` Flag

Selects whether `$ZN` resolves to `$ZNEXT` or `$ZNAME`.

If *tvexpr* evaluates to *true*, selects `$ZNAME`. Otherwise, selects `$ZNEXT`.

Syntax

```
VIEW 71:tvexpr
```

`72` - Set `$ZPREVIOUS/$ZPIECE` Flag

Selects whether `$ZP` resolves to `$ZPREVIOUS` or `$ZPIECE`.

If *tvexpr* evaluates to *true*, selects `$ZPIECE`. Otherwise, selects `$ZPREVIOUS`.

Syntax

```
VIEW 72:tvexpr
```

`73` - Set `$ZDATA/$ZDATE` Flag

Selects whether `$ZD` resolves to `$ZDATA` or `$ZDATE`.

If *tvexpr* evaluates to *true*, selects `$ZDATE`. Otherwise, selects `$ZDATA`.

Syntax

```
VIEW 73:tvexpr
```

`79` - Set Old `ZJOB` vs. New `ZJOB` Flag

If *tvexpr* evaluates to *true*, sets the `ZJOB` mode to new, otherwise, sets it to old.

Syntax

```
VIEW 79:tvexpr
```

`80` - Set or Clear 8-Bit Flag

If *tvexpr* evaluates to *true*, sets FreeM to 8-bit mode. Otherwise, sets FreeM to 7-bit mode.

Syntax

```
VIEW 80:tvexpr
```

`81` - Set or Clear PF1 Flag

If *tvexpr* evaluates to *true*, sets the `PF1` flag. We do not yet know what this does.

Syntax

```
VIEW 81:tvexpr
```

`83` - Set or Clear Text in `$ZERROR` Flag

If *tvexpr* evaluates to *true*, descriptive error messages will be included in `$ZERROR`. Otherwise, only the short error code (i.e. *ZILLFUN*) will be included in `$ZERROR`.

Syntax

```
VIEW 83:tvexpr
```

`92` - Set Type Mismatch Error Flag on `EUR2DEM`

If *tvexpr* evaluates to *true*, a type mismatch error will be thrown in `EUR2DEM` currency conversions in certain situations that we do not yet understand.

Syntax

```
VIEW 92:tvexpr
```

`93` - Define `ZKEY` Production Rule

We do not know what this does.

`96` - Set Global Prefix

Forces global data filenames to be prefixed with the result of *expr*.

Syntax

```
VIEW 96:expr V string
```

`97` - Set Global Postfix

Forces global data filenames to be postfixed with the result of *expr*.

Syntax

```
VIEW 97:expr V string
```

`98` - Set Routine Extension

Sets the default extension for M routine filenames to the result of *expr*.

Syntax

```
VIEW 98:expr V string
```

`101` - Set `ierr`

Sets the FreeM internal `ierr` value to *intexpr*. Used by some FreeM polyfills (commands or functions implemented in M code).

Syntax

```
VIEW 101:intexpr
```

`102` - Set `ierr` (Deferred)

Sets the FreeM internal `ierr` value to *intexpr*, but only after the current process stack level is exited. Used by FreeM polyfills to throw an error that will appear to come from the user's own code rather than the polyfill implementation M code.

Syntax

```
VIEW 102:intexpr
```

`103` - Signal `MERGE` to `^$WINDOW` Complete

Signals FreeM's MWAPI implementation that a `MERGE` to `^$WINDOW` or descendant subscripts thereof has completed.

Syntax

```
VIEW 103[:subscript]
```

`110` - Set Local `$ORDER/$QUERY` Data Value

Sets the local variable `$ORDER/$QUERY` data value to the result of *expr*. We're not entirely sure what this is.

Syntax

```
VIEW 110:expr
```

`111` - Set Global `$ORDER/$QUERY` Data Value

Sets the global variable `$ORDER/$QUERY` data value to the result of *expr*. We're not entirely sure what this is.

Syntax

```
VIEW 111:expr
```

`113` - Set `termio` Information

We don't know what this does.

`133` - Remember `ZLOAD` Directory on `ZSAVE`

We don't know what this does, but it takes a *tvexpr*.

Syntax

```
VIEW 133:tvexpr
```

12.34 WRITE

12.35 XECUTE

12.36 ZASSERT

FreeM Extension

Triggers error `ZASSERT` if the supplied truth-valued expression *tvexpr* is *false* (*1* is *true*, and *0* is *false*), and that the optional *postcondition* evaluates to *true* or is omitted.

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

Syntax

```
 ZASSERT:postcondition <tvexpr>
```

Example

```
DEFAULT.USER> SET DEBUG=1


DEFAULT.USER> ZASSERT:DEBUG 1=1


DEFAULT.USER> ZASSERT:DEBUG 1=0


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

12.37 ZBREAK

FreeM Extension

Sets or clears the `ZBREAK` flag[1], based on the result of evaluating *tvexpr*.

Syntax

```
ZBREAK tvexpr
```

[1] NOTE: FreeM team needs to investigate how `zbreakon` and `zbflag` affect program execution.

12.38 ZCONST

FreeM Extension

Defines a local *constant*, or variable that cannot be altered after its initial definition, provided the optional *postcondition* is *true* or omitted.

Constants must only be locals, and globals are not supported.

Syntax

```
ZCONST:postcondition mref1=initial-value1,...,mrefN=initial-valueN
```

12.39 ZGOTO

FreeM Extension

In its argumented form, enables `BREAK` mode and branches unconditionally to *entryref*.

Syntax

```
ZGOTO entryref
```

In its argumented form, resumes execution after a `BREAK`.

Syntax

```
ZGOTO
```

12.40 ZHALT

FreeM Extension

In its single-argumented form, `ZHALT` command is used to exit the FreeM process with a specific return value *intexpr*.

Syntax

```
ZHALT intexpr
```

In its argumentless form, `ZHALT` is synonymous with `HALT`.

Syntax

```
ZHALT
```

12.41 ZINSERT

FreeM Extension

12.42 ZJOB

FreeM Extension

When `ZJOB` is used, the semantics are identical to `JOB`, with the exception that the *timeout* is forced to be `0`, regardless of what the user specifies.

For more information, see `JOB`.

12.43 ZLOAD

FreeM Extension

Loads routine <*routine-name*> into FreeM's routine buffer, provided the optional *postcondition* is *true* or omitted.

Syntax

```
ZLOAD:postcondition <routine-name>
```

12.44 ZMAP

Maps global name `gvn` to be mapped to the non-default namespace *expr V namespace*, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
ZMAP[:postcondition] GLOBAL gvn=expr V namespace
```

12.45 ZNEW

FreeM Extension

12.46 ZPRINT

FreeM Extension

Prints the contents of the current routine buffer, provided the optional *postcondition* is *true* or omitted.

Syntax

```
ZPRINT:postcondition
```

12.47 ZQUIT

FreeM Extension

In its single-argument form, quits from *levels* levels of the stack, provided the optional *postcondition* is *true* or omitted.

In its argumentless form, quits from `$STACK` levels of the stack, provided the optional *postcondition* is *true* or omitted.

Syntax

```
ZQUIT:postcondition [levels]
```

12.48 ZREMOVE

FreeM Extension

12.49 ZSAVE

FreeM Extension

12.50 ZTHROW

FreeM Extension

Raises an error condition as long as the optional *postcondition* is *true* or omitted.

Syntax

```
ZTHROW:postcondition expr V error-code
```

Example

```
ZTHROW "M102"
```

12.51 ZTRAP

FreeM Extension

Synonymous with Section 12.50 [ZTHROW], page 52.

12.52 ZUNMAP

Removes any mapping connecting *gvn* to a non-default namespace, provided the optional *postcondition* evaluates to *true* or is omitted.

Syntax

```
ZUNMAP GLOBAL gvn
```

12.53 ZWATCH

FreeM Extension

Sets a watchpoint on a global, local, or SSVN 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 `^snw(1)`. It then changes the value of `^snw(1)`.

```
DEFAULT.USER> ZWATCH

Watchpoints enabled.

DEFAULT.USER> ZWATCH +^SNW(1)

Added '^SNW("1")' to the watchlist.
```

```
DEFAULT.USER> SET ^SNW(1)="new value"

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

The following example will remove that watchpoint:

```
DEFAULT.USER> ZWATCH -^SNW(1)

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

DEFAULT.USER> ZWATCH ?^SNW(1)

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

12.54 ZWITH

FreeM Extension

NOTE: This command may be deprecated and removed in future FreeM releases.

Sets a prefix to be applied to all subsequent local variable or constant references.

Syntax

```
ZWITH:postcondition var-prefix
```

In the above single-argument form, sets the `$WITH` prefix to *var-prefix*, provided that the optional *postcondition* is either *true* or omitted.

The *var-prefix* argument may be a string literal or any valid FreeM expression.

```
ZWITH:postcondition
```

In the above argumentless form, clears the `$WITH` prefix, provided the optional *postcondition* is either *true* or omitted. Equivalent to `ZWITH ""`.

12.55 ZWRITE

FreeM Extension

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.

13 Structured System Variables

SSVN 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.

13.1 ^$CHARACTER

Exposes character set information. As FreeM currently only supports the `M` character set, the first subscript of `^$CHARACTER` must always be `"M"`.

The following values for the second subscript are supported:

`IDENT` +R -U -D
: Returns the empty string.

`COLLATE` +R -U -D
: Returns the empty string.

`INPUT` +R -U -D
: Returns the empty string if the third subscript is `M`, otherwise, raises error `M38`.

`OUTPUT` +R -U -D
: Returns the empty string if the third subscript is `M`, otherwise, raises error `M38`.

13.2 ^$DEVICE

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

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

The following values for the second subscript are supported:

`$DEVICE` Returns the value of `$DEVICE` for the specified I/O channel.

`$X` +R -U -D
: Returns the horizontal cursor position of a terminal device. Only valid if the I/O channel is `0`.

`$Y` +R -U -D
: Returns the vertical cursor position of a terminal device. Only valid if the I/O channel is `0`.

`ROWS` +R -U -D
: Returns the number of character rows on the terminal device. Only valid if the I/O channel is `0`.

`COLUMNS` +R -U -D
: Returns the number of character columns on the terminal device. Only valid if the I/O channel is `0`.

`CHARACTER` +R -U -D
: Returns the character set of the specified I/O channel; always `M` in the current implementation.

`INPUT_BUFFER` +R +U -D
: Returns or sets the contents of the input buffer for the specified I/O channel. Data populated in this node will remain in the buffer until subsequent `READ` command(s) remove it. This can be used to perform input buffer stuffing, i.e., to fill out an interactive form programmatically.

`NAME` +R -U -D
: Returns the operating system's name for the file, device, or socket attached to the specified I/O channel.

`FD` +R -U -D
: Returns the UNIX file descriptor of the specified I/O channel.

`MODE` +R -U -D
: Returns one of `READ`, `WRITE`, `READWRITE`, or `APPEND`, depending on the mode in which the specified I/O channel was opened.

`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.

`NAMESPACE` +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.

`TYPE` +R -U -D
: Returns either `1,FILE`, `2,SOCKET`, or `4,TERMINAL`, depending on the device type associated with the specified I/O channel.

`OPTIONS` -R -U -D
: The following subscripts reside beneath `^$DEVICE(<io-channel>,"OPTIONS")`, and this subscript may not be accessed without one of the following third-level subscripts being specified:

  `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).

  `TERMID` +R -U -D
  : Returns the type of terminal connected to channel 0. Only valid for I/O channel 0.

  `ECHO` +R +U -D
  : Enables or disables local echo of characters typed in a `READ` command. Only valid for I/O channel 0. Corresponds to bit 0 of the Device Status Word.

`DELMODE` +R +U -D
: Enables or disables visual backspace during a `READ` command. Only valid for I/O channel 0. Corresponds to bit 2 of the Device Status Word.

`ESCAPE` +R +U -D
: Enables or disables escape sequence processing during a `READ` command. Only valid for I/O channel 0. Corresponds to bit 6 of the Device Status Word.

`CONVUPPER` +R +U -D
: Enables or disables automatic conversion to uppercase of alphabetical characters during a `READ` command. Only valid for I/O channel 0. Corresponds to bit 14 of the Device Status Word.

`DELEMPTY` +R +U -D
: Enables or disables the automatic deletion of empty strings supplied to a `READ` command. Only valid for I/O channel 0. Corresponds to bit 19 of the Device Status Word.

`NOCTRLS` +R +U -D
: TBD. Only valid for I/O channel 0. Corresponds to bit 20 of the Device Status Word.

`CTRLOPROC` +R +U -D
: Enables or disables *Ctrl-O* processing during `READ` commands. Only valid for I/O channel 0. Corresponds to bit 21 of the Device Status Word.

`NOTYPEAHEAD` +R +U -D
: Enables or disables typeahead buffering during `READ` commands. Only valid for I/O channel 0. Corresponds to bit 25 of the Device Status Word.

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
```

13.3 ^$DISPLAY

Provides information about the specified graphical display. The first subscript corresponds to a display number, which is an integer value, often corresponding to the current value of the `$PDISPLAY` ISV.

The following second-level subscripts and specified descendant subscripts are supported:

`CLIPBOARD` +R +U +D
: Retrieves, sets, or erases the contents of the system clipboard.

`PLATFORM` +R -U -D
: Retrieves the name and version of the underlying window system platform.

`SIZE` +R -U -D
: Retrieves the display resolution of the specified graphical display. For instance, a 1080p display would have a `SIZE` value of `1920,1080`.

`SPECTRUM` +R -U -D
: Retrieves the color depth (number of colors supported) of the specified graphical display.

`COLORTYPE` +R -U -D
: Always returns `COLOR`, as monochrome and grayscale displays are not yet supported in FreeM.

`UNITS` +R -U -D
: Returns the measurement unit of the specified display, i.e., `PIXEL`.

`TYPEFACE` +R -U -D
: The third-level subscripts beneath this subscript represent a list of font families available on this display. The fourth level subscript is a list of sizes supported for the specified typeface, or `0` for vector typefaces, such as TrueType, OpenType, and Adobe Type 1 fonts.

13.4 ^$EVENT

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

13.5 ^$GLOBAL

The `^$GLOBAL` structured system variable provides information about M globals. The first-level subscript is a global name, sans the leading caret symbol.

The following second-level subscripts are supported:

`BYTES` +R -U -D
: Returns the number of bytes this global occupies in fixed storage.

`BLOCKS` +R -U -D
: Returns the number of blocks contained in this global.

`BLOCKSIZE` +R -U -D
: Returns the size of data blocks for this global. Currently, FreeM only supports 1024-byte blocks.

`FILE` +R -U -D
: Returns the full filesystem path to the data file where this global resides in fixed storage.

`NAMESPACE` +R +U +D
: Returns or sets the name of the FreeM namespace to which this global belongs. `SET`ting this node creates a mapping for the specified global name to a non-default namespace. `KILL`ing this node restores the mapping configuration for the specified global to the default.

13.6 ^$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:

`GVNDEFAULT` +R +U +D
: Contains a default expression to be evaluated if a global variable access attempt results in an `M7` error.

  Equivalent to wrapping all global accesses in `$GET(`*`global-name,string-expr`*`)`.

`LVNDEFAULT` +R +U +D
: Contains a default expression to be evaluated if a local variable access attempt results in an `M6` error.

  Equivalent to wrapping all local accesses in `$GET(`*`global-name,string-expr`*`)`.

`LVNQOVAL` +R +U +D
: Contains the data value (if any) at the subscripted local variable reference from the most recent `$ORDER` or `$QUERY` operation.

  This node is useful for code that uses `$ORDER` or `$QUERY` heavily in loops that retrieve successive data values, as it will prevent an additional symbol table scan that would result from retrieving the data value in the usual way, thus improving application performance. However, this optimization comes at the cost of compatibility with other M implementations.

`GVNQOVAL` +R +U +D
: Contains the data value (if any) at the subscripted global variable reference from the most recent `$ORDER` or `$QUERY` operation.

  This node is useful for code that uses `$ORDER` or `$QUERY` heavily in loops that retrieve successive data values, as it will prevent an additional data file scan that would result from retrieving the data value in the usual way, thus improving application performance. However, this optimization comes at the cost of compatibility with other M implementations.

`ZCOMMANDS` +R +U -D

Contains a space-delimited list of `Z`-commands to be treated as intrinsic. Any `Z`-command not appearing in this list will be treated as a user-defined command.

For instance, if command `ZFOO` does *not* appear in this list, FreeM will attempt to run `^%ZFOO` as a subroutine when the `ZFOO` command is encountered in program code.

If you remove a command from this list, you may provide your own private M implementation of the command in the manner described above.

If an argument is passed to a `Z`-command you implement in M, it is made available to your M code in a variable whose name is specified in `^$JOB($JOB,"ZCOMMAND_ARGUMENT_NAME")`, which defaults to `%`.

`PIPE_GLVN` +R +U -D

Contains an M local or global variable to be used as standard input or standard output for the external shell commands run by `!<` and `!>`.

`ZCOMMAND_ARGUMENT_NAME` +R +U -D

Returns or sets the variable name in which arguments to user-defined `Z`-commands are passed. Defaults to `%`.

`ZFUNCTIONS` +R +U -D

Contains a space-delimited list of `Z` functions to be treated as intrinsic. Any `Z` function not appearing in this list will be treated as a user-defined extrinsic function.

For instance, if function `$ZFOO` does *not* appear in this list, FreeM will attempt to return the value of `$$^%ZFOO` called as an extrinsic function.

If you remove a function from this list, you may provide your own private M implementation of the function in the manner described above.

`ZSVS` +R +U -D

Contains a space-delimited list of `Z` special variables to be treated as intrinsic. Any `Z` special variable not appearing in this list will be treated as a user-defined extrinsic function taking no arguments.

For instance, if the special variable `$ZFOO` does *not* appear in this list, FreeM will attempt to return the value of `$$^%ZFOO` called as an extrinsic function.

If you remove a built-in special variable from this list, you may provide your own private M implementation of the special variable in the manner described above.

`BREAK_HANDLER` +R +U -D

Contains M code to be executed when the `BREAK` command is run.

`ROUTINE_BUFFER_SIZE` +R +U -D

Returns or sets the number of bytes allocated to each routine buffer. If `ROUTINE_BUFFER_AUTO_ADJUST` is set to `0`, this determines the maximum size of routines that FreeM will execute.

`ROUTINE_BUFFER_COUNT` +R +U -D
: Returns or sets the number of routine buffers that FreeM will store in memory concurrently. Raising this value will increase memory usage, but will also increase performance if your applications call many different routines repeatedly.

`ROUTINE_BUFFER_AUTO_ADJUST` +R +U -D
: Determines whether or not the size of routine buffers will be automatically adjusted at runtime. If set to `0`, routine buffers will be fixed to the byte size specified in `ROUTINE_BUFFER_SIZE` and may be manually resized using `ROUTINE_BUFFER_SIZE`. If set to `1`, routine buffers will grow automatically as necessary.

`SYMBOL_TABLE_SIZE` +R +U -D
: Returns or sets the number of bytes allocated to each of the two FreeM symbol tables. If `SYMBOL_TABLE_AUTO_ADJUST` is `1`, this value is treated as a default, initial size. If `SYMBOL_TABLE_AUTO_ADJUST` is `0`, this value controls the fixed size of the two symbol tables.

`SYMBOL_TABLE_AUTO_ADJUST` +R +U -D
: Determines whether or not the size of the two FreeM symbol tables will be automatically adjusted at runtime. If set to `0`, the symbol table will be fixed to the byte size specified in `SYMBOL_TABLE_SIZE` and may be manually resized by modifying `SYMBOL_TABLE_SIZE`. If set to `1`, the two symbol tables will grow automatically as necessary.

`USER_DEFINED_ISV_TABLE_SIZE` +R +U -D
: Returns or sets the number of bytes allocated to the FreeM user-defined intrinsic special variable table. If `USER_DEFINED_ISV_TABLE_AUTO_ADJUST` is `1`, this value is treated as a default, initial size. If `USER_DEFINED_ISV_TABLE_AUTO_ADJUST` is `0`, this value controls the fixed byte size of the user-defined intrinsic special variable table.

`USER_DEFINED_ISV_TABLE_AUTO_ADJUST` +R +U -D
: Determines whether or not the size of the FreeM user-defined intrinsic special variable table will be automatically adjusted at runtime. If set to `0`, the user-defined ISV table will be fixed to the byte size specified in `USER_DEFINED_ISV_TABLE_SIZE` and may be manually resized by modifying `USER_DEFINED_ISV_TABLE_SIZE`. If set to `1`, the user-defined ISV table will grow automatically as necessary.

`GVN_UNIQUE_CHARS` +R +U -D
: Returns or sets the number of characters of a global name that make it unique, from 1 to 255.

`GVN_CASE_SENSITIVE` +R +U -D
: Returns or sets the case sensitivity of global names. If set to `0`, global names are case-insensitive. If set to `1`, global names are case-sensitive.

`GVN_NAME_SUB_LENGTH` +R +U -D
: Returns or sets the maximum number of characters of a global name plus all of its subscripts, from 1-255.

`GVN_SUB_LENGTH` +R +U -D
: Returns or sets the maximum number of characters of a single global subscript, from 1-255.

`SINGLE_USER` +R +U -D
: If set to `1`, FreeM will skip all file locking operations on globals. If set to `0`, FreeM will enforce file locking on both.

  Setting `SINGLE_USER` to `1` will improve FreeM performance, but you must *ONLY* use this on systems where you are absolutely sure that only one FreeM process will run at any given time, as running multiple instances of FreeM concurrently when any of them are set to `SINGLE_USER` mode *will* cause global data corruption.

`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.

`OPEN` +R -U -D
: The `^$JOB($JOB,"OPEN",<channel>` subscripts list the open I/O channels in the specified job.

`BERKELEYDB,FLUSH_THRESHOLD` +R +U -D
: Returns or sets the number of write operations that will be cached in the BerkeleyDB global handler prior to flushing BerkeleyDB's cache to disk.

`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.

`REVSTR` +R +U -D
: When set to 1, allows `$EXTRACT` to accept negative values.

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

`SYMTAB` +R +U -D
: Returns or sets the current local variable symbol table in use.

  FreeM supports two unique and independent symbol tables, allowing FreeM programs to maintain two independent sets of identically- or differently-named local variables per process.

  The default symbol table is `0`, and the alternate symbol table is `1`, corresponding to the valid values for `^$JOB($JOB,"SYMTAB")`.

Setting this subscript to values other than `0` or `1` will result in a `ZINVEXPR` error.

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

`MATH` +R +U -D
: Returns or sets the mode in which decimal comparisons and arithmetic calculations are conducted. Valid values are `FIXED`, for fixed-point decimals having up to 20,000 digits of precision, as determined by the `$ZPRECISION` intrinsic special variable, and `IEEE754`, to use IEEE 754 floating-point decimals. When in `IEEE754` mode, floating-point numbers support up to 16 digits of numeric precision.

  `IEEE754` mode will make mathematical calculations significantly faster, especially when accelerated by a floating-point processor, at the expense of precision and accuracy.

  `FIXED` mode is recommended for financial calculations, or where precision and accuracy are valued over performance. `FIXED` is the default mode of FreeM operation.

  Attempting to `SET` this node to values other than `FIXED` or `IEEE754` will set `$ECODE` to `M29`.

13.7 ^$LOCK

The first-level subscript of `^$LOCK` is a lock name. The value at each node is the PID which owns the lock, a comma, and the lock counter for the locked resource.

Attempting to `SET` or `KILL` any node in `^$LOCK` will raise error `M29`.

13.8 ^$OBJECT

13.9 ^$ROUTINE

The `^$ROUTINE` SSVN exposes a list of routines available in the current FreeM namespace, as well as additional attributes further describing each routine.

The first-level subscript is the name of a FreeM routine minus the leading caret symbol.

The following second-level subscripts are supported:

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

`NAMESPACE` +R -U -D
: Returns the name of the FreeM namespace in which the routine resides.

`PATH` +R -U -D
: Returns the full filesystem path to the routine in fixed storage.

13.10 ^$SYSTEM

The `^$SYSTEM` SSVN exposes system-level implementation details.

The following first-level subscripts are supported:

`DEFPSIZE` +R -U -D
: Returns the default size in bytes of the symbol table and routine buffer memory partition.

`DEFUDFSVSIZ` +R -U -D
: Returns the default size in bytes of the user-defined intrinsic special variable table.

`DEFNSIZE` +R -U -D
: Returns the default size of the `NEW` stack, in number of entries.

`MAXNO_OF_RBUF` +R -U -D
: Returns the maximum number of routine buffers.

`DEFNO_OF_RBUF` +R -U -D
: Returns the default number of routine buffers.

`DEFPSIZE0` +R -U -D
: Returns the default size in bytes of each routine buffer.

`NO_GLOBLS` +R -U -D
: Returns the maximum number of globals that can be concurrently opened.

`NO_OF_GBUF` +R -U -D
: Returns the number of global buffers.

`NESTLEVLS` +R -U -D
: Returns the depth of the `DO`, `FOR`, `XECUTE` stack.

`PARDEPTH` +R -U -D
: Returns the maximum depth of the parser's parameter stack.

`PATDEPTH` +R -U -D
: Returns the maximum number of *patatom*s in each pattern.

`TRLIM` +R -U -D
: Returns the trace limit of the `BUILTIN` global handler.

`ARGS_IN_ESC` +R -U -D
: Returns the maximum number of arguments in a terminal escape sequence.

`ZTLEN` +R -U -D
: Returns the maximum length of `$ZTRAP`.

`FUNLEN` +R -U -D
: Returns the maximum length of the `$ZF` (function key) variable.

`NAME_LENGTH` +R -U -D
: Returns the maximum length of variable names in the current FreeM build. Compatible with the same SSVN node in *Reference Standard M*

`STRING_MAX` +R -U -D
: Returns the maximum length of character strings in the current FreeM build. Compatible with the same SSVN node in *Reference Standard M*

`$NEXTOK` +R -U -D
: Returns a value indicating whether or not the `$NEXT` intrinsic function is allowed. In FreeM, `$NEXT` is always enabled, and this SSVN is provided solely for compatibility with *Reference Standard M.* Thus, this SSVN node always returns `1`.

`EOK` +R -U -D
: Returns a value indicating whether or not `E` notation for exponents is allowed. In FreeM, this feature is always enabled, and this SSVN is provided solely for compatibility with *Reference Standard M.* Thus, this SSVN node always returns `1`.

`OFFOK` +R -U -D
: Returns a value indicating whether or not offsets are allowed in `DO` and `GOTO`. In FreeM, this feature is always enabled, and this SSVN is provided solely for compatibility with *Reference Standard M.* Thus, this SSVN node always returns `1`.

`BIG_ENDIAN` +R -U -D
: Returns a 1 if FreeM is running on a big-endian platform, or a 0 otherwise. Compatible with the same SSVN node in *Reference Standard M.*

`NAMESPACE` +R -U -D
: The descendant subscripts of this node list each namespace in the current FreeM environment.

`MAPPINGS,GLOBAL` +R -U -D
: Descendant subscripts of this node represent global name mappings set in `^$GLOBAL(`*gvn*`,"NAMESPACE")`

13.11 ^$WINDOW

The `^$WINDOW` SSVN has no nodes yet defined. However, completing a `MERGE` to this SSVN will cause MWAPI-ish things to happen, and further work is proceeding on MWAPI implementation.

13.12 ^$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.

`ATTRIBUTE` +R -U -D
: Exposes the `/proc` files as descendant subscripts, i.e., `WRITE ^$ZPROCESS(2900,"ATTRIBUTE","cmdline"),!` would print the initial command line used to invoke process ID 2900. Note that the third subscript (the immediate descendant of the `ATTRIBUTE` subscript) is case sensitive.

`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`.

  The constants `%SYS.SIGNAL.HUP`, `%SYS.SIGNAL.INT`, `%SYS.SIGNAL.KILL`, and `%SYS.SIGNAL.TERM` are provided for convenient use of this SSVN subscript.

13.13 ^$ZRPI

The `^$ZRPI` structured system variable provides easy access to general-purpose input/output (GPIO) pins on Raspberry Pi single-board computers.

To initialize the GPIO subsystem, `SET ^$ZRPI("INITIALIZE")=1`.

Individual pins are accessed through `^$ZRPI("GPIO",<pin>,...)`, where `<pin>` represents the desired pin number. Descendant subscripts of `^$ZRPI("GPIO",<pin>)` are as follows:

`MODE` +R +U -D
: Represents the operating mode of the selected pin. One of `INPUT`, `OUTPUT`, `PWM_OUTPUT`, or `GPIO_CLOCK`.

`DIGITAL` +R +U -D
: Reads or writes the selected pin digitally. The value is limited to `1` or `0`.

`ANALOG` +R +U -D
: Reads or writes the selected pin in an analog fashion. The value represents analog voltage.

14 Operators

14.1 Unary +

Forces a number to positive, whether positive or negative. Also forces numeric coercion of strings.

14.2 Unary -

Forces a number to negative, whether positive or negative. Also forces numeric coercion of strings.

14.3 + (Add)

Syntax

```
S X=1+2 ; => 3
```

Adds numbers together.

14.4 += (Add/Assign)

Syntax

```
S X=5
S X+=3 ; => 8
```

Increments the variable on the LHS by the value on the RHS.

14.5 ++ (Postfix Increment)

Increments a variable by 1.

14.6 - (Subtract)

Subtracts one number from another.

14.7 -= (Subtract/Assign)

Syntax

```
S X=5
S X-=3 ; => 2
```

Decrements the variable on the LHS by the value on the RHS.

14.8 – (Postfix Decrement)

Decrements the variable by one.

14.9 * (Multiply)

Multiplies one number by another.

14.10 *= (Multiply/Assign)

14.11 / (Divide)

14.12 /= (Divide/Assign)

14.13 \ (Integer Divide)

14.14 \= (Integer Divide/Assign)

14.15 # (Modulo)

14.16 #= (Modulo/Assign)

14.17 ** (Exponentiate)

14.18 **= (Exponentiate/Assign)

14.19 < (Less Than)

14.20 <= (Less Than or Equal To)

14.21 > (Greater Than)

14.22 >= (Greater Than or Equal To)

14.23 _ (Concatenate)

14.24 _= (Concatenate/Assign)

14.25 = (Equals)

14.26 [(Contains)

14.27] (Follows)

14.28]] (Sorts After)

14.29 ? (Pattern Match)

14.30 & (Logical AND)

14.31 ! (Logical OR)

14.32 ' (Logical NOT)

14.33 @ (Indirect)

15 Routines

A *routine* is a file containing M source code to be processed by FreeM.

Routines exist within a *namespace* (such as `SYSTEM` or `USER`), which in turn exist within an *environment* (such as `DEFAULT`).

15.1 Routine Naming

The routine's filename follows the format `NAME.m`, where `NAME` is the name of the routine, and `.m` is the filename extension.

Routine naming rules are as follows:

- Routine names must begin with an upper- or lower-case letter, or a `%` sign
- Within the routine name, you may have upper-case or lower-case letters or digits
- The entire routine name must not be longer than 255 characters

Routines whose names begin with `%` must be located in the `SYSTEM` namespace. Other routines may be located in any namespace.

16 Types

FreeM supports all *libdatatype* types defined in the former MDC's *Millennium Draft Standard*, with the exception of `MATRIX`, and with extensions supporting object-oriented programming. A notable enhancement in FreeM is that the library data types can be used in the *formallist* of any extrinsic function or subroutine; not only in *libraryelement*s.

16.1 BOOLEAN

The `BOOLEAN` type represents any M value that can be interpreted as a truth-value.

16.2 COMPLEX

The `COMPLEX` type is a complex number represented as a string in the format `<real-part>%<imaginary-part>`, where *real-part* and *imaginary-part* are both `REAL` numbers. See Section 16.4 [REAL], page 70, for more information.

FreeM will attempt to interpret any `COMPLEX` value according to the usual rules for M canonical numbers, i.e., the string `sabc123.345%fbd3.1` would be interpreted as a complex number with the real part being `123.345` and the imaginary part being `3.1`.

16.3 INTEGER

An `INTEGER` is an interpretation of numeric data with any fractional part removed.

16.4 REAL

A `REAL` is a numeric interpretation of data including a fractional part.

16.5 STRING

The `STRING` is the fundamental FreeM data type. Other types are inferred from the context of their usage.

16.5.1 String Rules

The following rules apply to all FreeM strings:

- Must not exceed 255 characters
- Must not contain `$C(0)`, `$C(201)`, or `$C(202)`

16.5.2 String Quoting Rules

Strings in FreeM must be surrounded in double quotes:

```
SET MYSTRING="This is a string literal"
```

If you want to include double quotes inside of a string, simply double them:

```
SET MYSTRING="This is a ""string literal"" with embedded double quotes"
```

16.6 Custom Types (Classes)

See Chapter 25 [Object-Oriented Programming], page 83.

17 Globals

17.1 Globals Overview

FreeM supports typical M globals, which are often described as persistent, hierachical sparse arrays. Globals make it relatively simple to include persistent data in an application without requiring the developer to use an external database management system, and offer syntax and semantics so similar to M local variables and structured system variables that moving from one to the other is seamless.

Each global comprises three elements:

- An alphabetic name beginning with a caret (`^`) or a caret and a percent sign (`^%`)
- Optionally, one or more comma-delimited subscripts, enclosed in parentheses
- A value of up to 255 characters in length

A leading percent sign in the global name will force the named global into the `SYSTEM` namespace of the current FreeM environment.

17.2 Creating Globals

To create a global, you can use the `SET` command:

```
SET ^MYGLOBAL("foo","bar")="this is the data value"
```

17.3 Removing Globals

To remove an entire global, you can use the `KILL` command with the unsubscripted name of the global:

```
KILL ^MYGLOBAL
```

If you only want to remove part of a global, i.e., beginning at a certain subscript level, use the `KILL` command with a subscripted name:

```
KILL ^MYGLOBAL("foo")
```

This will remove only the `"foo"` subscript and all of its children.

If you only want to remove the data value at a specific subscript level, leaving the subscript itself intact, use `KVALUE`:

```
KVALUE ^MYGLOBAL("foo")
```

17.4 Global Storage

FreeM globals are stored in `$PREFIX/var/freem/<`*`environment-name`*`>/<`*`namespace-name`*`>/globals` in a binary format.

Global files have a header of the following format:

```
typedef struct global_header {

    char magic[5]; /* FRMGL */
    int format_version;
    char host_triplet[40];
    char host_id[256];

    unsigned long block_size;
    unsigned long last_transaction_id;

    long created;
    long last_backup;

} global_header;
```

18 Concurrency Control

18.1 Concurrency Control Overview

Multitasking, multi-user FreeM applications must concern themselves with conscientious management of concurrent access to globals in order to maintain logical consistency and prevent concurrent reads and writes from conflicting with each other.

In FreeM, there are two mechanisms provided for managing concurrent global access: *advisory locks*, and *transaction processing.*

Advisory locks allow applications to voluntarily coordinate concurrent access to globals with the `LOCK` command, and require each application to check the `LOCK` status prior to accessing a global.

Transaction processing allows applications to delineate sets of global operations (sets, kills, etc.) as being part of a transaction, in which no operations are performed against the globals contained within the transaction until the transaction is committed. In addition, processes other than the one running the transaction will be forced to wait to access globals for either the duration of the commit phase (*batch mode*), or for the entire duration of the transaction (*serial mode*).

18.2 Advisory Locks

18.3 Transaction Processing

FreeM implements a significant subset of the transaction processing features from *ANSI X11.1-1995.* This allows a series of global operations to be conducted all at once, either in batch mode (where concurrent operation is not interrupted until the last possible moment), or in serial mode (where writes are guaranteed to be atomic, consistent, isolated, and durable).

18.3.1 Theory of Operation

FreeM uses a pessimistic concurrency control mechanism for `SERIAL` transactions, meaning that any `TSTART` command that includes the `SERIAL` transaction parameter will cause the process to acquire the transaction processing mutex, which prevents any process but the one holding the mutex from performing any data access (read or write) until either `TCOMMIT` or `TROLLBACK` is called, either committing or rolling back the transaction, respectively.

Any transaction in between its `TSTART` and `TCOMMIT/TROLLBACK` is said to be *in-flight.* During the in-flight stage, pending global operations are held only in memory and after-image journals.

FreeM maintains a list of all globals affected during a transaction in-flight. When a `TCOMMIT` is reached, FreeM will generate a *checkpoint* of each global data file to be changed by the transaction. These checkpoints allow all FreeM globals to be restored to their pre-transaction state if a `TCOMMIT` should fail part of the way through its operation.

Checkpoints can have one of two modes:

`CP_REMOVE`
: Used for globals that did not exist prior to the beginning of this transaction. Simply marks the entire global data file for deletion in case of `TCOMMIT` failure.

`CP_RESTORE`
: Used for globals that *did* exist prior to the beginning of this transaction. In this case, the entire global data file is copied to a new file with a `.chk` extension. In cases of `TCOMMIT` failure, `CP_RESTORE` checkpoint files will be restored over the partially-modified live data file.

The below example shows a few global operations and checkpoints for a transaction in-flight using the `trantab` direct-mode command:

```
TL1:DEFAULT.USER> trantab
 $TLEVEL 1*
  Operations for Transaction ID: 6ea14aad-b8f1-47f9-9f52-4f513f892bc0 [RESTARTABLE SERIAL]

   OP. NO.   ACTION         KEY/DATA
   -------   ------         --------
   1         SET            ^FOO=3
   2         KILL           ^FOO
   3         SET            ^snw=10
   4         SET            ^BRANDNEW=6

  Global checkpoints:

   GLOBAL                         MODE                FILES
   ------                         ----                -----
   ^BRANDNEW                      CP_REMOVE           IN:   /usr/local/var/freem/USER/global
   ^snw                           CP_RESTORE          IN:   /usr/local/var/freem/USER/global
                                                      OUT:  /usr/local/var/freem/USER/global
   ^FOO                           CP_RESTORE          IN:   /usr/local/var/freem/USER/global
                                                      OUT:  /usr/local/var/freem/USER/global
```

In the above example, `IN` files are the live data file that will be overwritten or removed, and `OUT` files are the checkpoints themselves. Note that `OUT` files are only used for `CP_RESTORE` checkpoints.

18.3.2 Using Transaction Processing

To use transactions in FreeM, you need to be familiar with three commands:

- `TSTART`
- `TCOMMIT`
- `TROLLBACK`

With transaction processing, global variable operations occurring between `TSTART` and `TCOMMIT` commands will be contained within the transaction.

The atomicity, consistency, isolation, and durability facets of FreeM transaction hinge on the transaction mode.

18.3.2.1 BATCH Transactions

`BATCH` transactions offer higher performance, and allow other applications aside from the one doing the transaction to continue normal operations until the transaction is committed with `TCOMMIT`. In batch mode, other processes are only locked out of normal operation during the commit phase of the transaction.

The effect of this is that the operations within the batch transaction will not be interleaved with global writes from other applications, but the entire lifetime of the transaction is not guaranteed to be serialized with respect to the transaction processing activities of other running applications in the environment.

18.3.2.2 SERIAL Transactions

`SERIAL` transactions offer full ACID compliance at the expense of multiprocessing performance. In serial mode, a `TSTART` blocks all activity from all other FreeM processes in the environment, and this blocking effect is not released until the transaction is committed with `TCOMMIT` or rolled back with `TROLLBACK` (or due to abnormal conditions in the environment that preclude the successful completion of the transaction).

19 Local Variables

19.1 Local Variables Overview

FreeM *local variables* have the same data structure as global variables, but are scoped to a single FreeM process, and stored in memory.

Each local comprises three elements:

- An alphabetic name beginning with a letter or a percent sign (`%`)
- Optionally, one or more comma-delimited subscripts, enclosed in parentheses
- A value of up to 255 characters in length

19.2 Creating Local Variables

To create a local variable, use the `SET` command:

```
SET MYLOCAL("foo","bar")="this is the data value"
```

19.3 Removing Local Variables

To remove an entire local variable, you can use the `KILL` command with the unsubscripted name of the variable:

```
KILL MYLOCAL
```

If you only want to remove part of a local variable, i.e., beginning at a certain subscript level, use the `KILL` command with a subscripted name:

```
KILL MYLOCAL("foo")
```

This will remove only the `"foo"` subscript and all of its children.

If you only want to remove the data value at a specific subscript level, leaving the subscript itself intact, use `KVALUE`:

```
KVALUE MYLOCAL("foo")
```

20 Scoping

By default, FreeM local variables and their values are scoped to the entire process, meaning that any function or subroutine can access and modify their values. This can lead to pernicious bugs.

M provides the `NEW` command to work around these issues. When `NEW` is called with a local variable as its argument, FreeM will scope the variable to the process stack frame in which the `NEW` command occured. When exiting the stack frame (i.e. with the `QUIT` command), FreeM will restore the variable to its value prior to being `NEW`ed.

Example

```
MYRTN ;
  S J=1 ; set local variable J to 1
  W J,! ; this will output "1"
  D X   ; execute subroutine X
  W J,! ; this will output "1", as the value of J was restored
  Q
  ;;
X ;
  N J   ; stack J
  S J=6 ; set its value to 6
  W J,! ; this will output "6"
  Q     ; quit from the subroutine, destroying its stack frame
  ;;
```

20.1 Scoping Considerations for $TEST

In M, the truth value of comparisons, logic operations, and certain forms of `LOCK` is stored in the `$TEST` intrinsic special variable, which follows the same rules as any M local variable.

This is probably the most significant design flaw of the language, as the side effects of logic on `$TEST` lead to incredibly difficult bugs. However, M allows `$TEST` to be `NEW`ed, and FreeM provides the `THEN` command[1] to help in the case of conditionals. `THEN` stacks `$TEST` to the end of the line.

When writing new M code in FreeM, we strongly suggest using `THEN` as follows:

```
MYRTN ;
  IF MYVAR=1 THEN DO SUBRT
```

This is instead of the traditional form:

```
MYRTN ;
  IF MYVAR=1 DO SUBR
```

[1] From MDC Type A extension X11/1998-31

Style Recommendation

Note that `THEN` is not in any currently published version of the *Standard*, but is part of MDC Type A extension X11/1998-31. However, we recommend using `THEN` instead of favoring portability, as there is no defensible reason for this incredibly simple feature *not* to be ubiquitous.

If you use other M implementations, you should bug the implementers to implement `THEN`, as it at least partially mitigates an inexcusable flaw in the design of M.

21 Decision Constructs

22 Branch Constructs

23 Loop Constructs

24 Modular Programming

24.1 Subroutines

24.2 Extrinsic Functions

25 Object-Oriented Programming

25.1 Classes

25.1.1 Class Overview

A *class* is the primary organizing concept of FreeM support for object-oriented programming, and in FreeM, is simply an M routine with a few special properties:

```
MYCLASS(THIS,INIT):OBJECT ; Constructor for MYCLASS, inherits OBJECT
  ; two private variables
  S THIS("NUMERATOR"):PRIVATE=$P(INIT,"/",1)
  S THIS("DENOMINATOR"):PRIVATE=$P(INIT,"/",2)
  Q
  ;
DESTROY(THIS) ; This is the destructor
  Q
```

The above example demonstrates general class syntax.

25.1.2 Constructors

A *constructor* is an M entry point that is called when a new instance of a class is created.

A constructor must be the first entry point in a class routine, its tag must match the class/routine name, and it must take two arguments, `THIS` and `INIT`.

`THIS` represents the instance of the object being accessed, and `INIT` represents an initializer that can be used to assign an initial value to the object when instantiating the class.

A constructor looks like this:

```
%FRACTION(THIS,INIT):OBJECT ;
   S THIS("NUMERATOR"):PRIVATE=$P(INIT,"/",1)
   S THIS("DENOMINATOR"):PRIVATE=$P(INIT,"/",2)
   Q
```

Syntax

```
<class-name>(THIS,INIT)[:<superclass>]
```

In the above example, <*superclass*> represents the name of a class from which this class should inherit. In this case, the `FRACTION` class inherits from the `OBJECT` class. Note that this is not strictly necessary in this case, as all classes in FreeM automatically inherit from `OBJECT`.

25.1.3 Destructors

A `destructor` is called when you `KILL` an instance variable. Its tag must be `DESTROY`, and it must take one argument (`THIS`).

The destructor should be used to clean up any resources used by class methods.

A destructor looks like this:

```
DESTROY(THIS) ;
  ; free any resources that should be freed at the end of the object's lifetime
  Q
```

25.2 Inheritance

Every class you create will automatically inherit the methods and functionality of the `OBJECT` class, supplied with FreeM.

When attempting to call a method, FreeM will first search the class routine for a matching entry point, and then follow the inheritance chain upwards until a matching entry point is found. If the final class in the chain does not have a matching entry point, FreeM will try to find a matching entry point in the `OBJECT` class.

Inheritance is achieved by specifying the name of the superclass in the constructor:

```
CLASS(THIS,INIT):SUPERCLASS
```

25.2.1 Runtime Polymorphism

You can achieve runtime polymorphism by subclassing, and defining methods in the subclass that match the names of existing methods in the superclass. Following FreeM inheritance rules, the overridden method in the subclass will be called, and the method in the superclass will not.

Note that the overridden method in the subclass can take a different set or number of arguments than the *formallist* of the superclass method would specify.

25.3 Methods

Class methods are defined as tags with *formallist*s in a class routine, and per the typical FreeM object pattern, must take at least one argument, being `THIS` (representing a reference to the object instance being accessed).

The following class (`MYCLASS`) has a constructor, a destructor, and a method called `MYMETHOD`:

```
%MYCLASS(THIS,INIT) ;
  Q THIS
DESTROY(THIS) ;
  Q
MYMETHOD(THIS) ;
  Q "VALUE"
```

The dot operator is used to invoke class methods:

```
DEFAULT.USER> N MYOBJ=$#^%MYCLASS("")
DEFAULT.USER> W MYOBJ.MYMETHOD()
VALUE
```

25.4 Public and Private Variables

FreeM supports private fields with the `:PRIVATE` specifier in the `SET` command, enforcing classical object-oriented data encapsulation. The `:PUBLIC` specifier is provided for completeness, and is the default.

The below constructor for a `FRACTION` class defines two private fields:

```
%FRACTION(THIS,INIT):OBJECT ;
   S THIS("NUMERATOR"):PRIVATE=$P(INIT,"/",1)
   S THIS("DENOMINATOR"):PRIVATE=$P(INIT,"/",2)
```

```
        Q
```

Either of the following commands will create a public field:

```
        S THIS("VARNAM")="Initial Value"
        S THIS("VARNAM"):PUBLIC="Initial Value"
```

Attempting to access private fields from outside of the class will raise error condition `ZOBJFLDACCV`.

25.5 Instantiating Objects

To instantiate an object (i.e., create an object from a certain class), you will use the `NEW` command as follows:

```
     NEW MYSTR=$#^%STRING("myString")
```

This will create a local variable called MYSTR of type STRING, and initialize it with the value myString.

25.5.1 Determining Object Class

To determine the class of any FreeM local variable, you will use the `$$TYPE()` method:

```
     USER> W MYSTR.$$TYPE()
     ^%STRING
```

The `$$TYPE()` method is a member of the `OBJECT` class.

26 Libraries

27 Sequential I/O

28 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.

28.1 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 ^$DEVICE(100,"OPTIONS","TERMINATOR")=$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
```

29 Extended Global References

29.1 Standard Extended Global References

FreeM supports extended global references, allowing the user to access globals in namespaces other than the current default namespace and the `SYSTEM` namespace, without switching to the other namespace.

For example, if you are in the `USER` namespace, the following code will print the value of `^VA(200,0)` in the `VISTA` namespace:

```
WRITE ^|"VISTA"|VA(200,0),!
```

You may also use an expression that resolves to a string containing a valid namespace name:

```
SET NS="VISTA"
WRITE ^|NS|VA(200,0),!
```

30 Global Aliasing

FreeM provides the ability to set alternative names for M global variables.

To create an alias of `^FOO` named `^BAR`, use the following command:

```
        SET ^$JOB($JOB,"ALIASES","^BAR")="^FOO"
```

If such an alias is set, any reference to global variable `^BAR` will affect `^FOO` instead of `^BAR` until `^$JOB($JOB,"ALIASES","^BAR")` is `KILL`ed. If `^BAR` existed prior to the definition of this alias, its data will be unavailable to and unaffected by application code.

31 Global Mappings

FreeM supports creating persistent mappings through which arbitrary global names may be mapped to specific namespaces. This allows non-`%` globals to be stored in the `SYSTEM` namespace, or `%` globals to be stored in non-`SYSTEM` namespaces.

To map the `^FOO` global to the `SYSTEM` namespace, any of the following will work:

```
MAP GLOBAL ^FOO="SYSTEM"
SET ^$GLOBAL("FOO","NAMESPACE")="SYSTEM"
SET ^$SYSTEM("MAPPINGS","GLOBAL","^FOO")="SYSTEM"
```

There is no functional difference in any of the three approaches; the method you choose is a matter of personal preference.

To remove the above mapping, any of the following examples will also work:

```
UNMAP GLOBAL ^FOO
KILL ^$GLOBAL("FOO","NAMESPACE")
KILL ^$SYSTEM("MAPPINGS","GLOBAL","^FOO")
```

32 Asynchronous Event Handling

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

32.1 Setting Up Async Event Handlers

Asynchronous event handlers are configured through the `^$JOB` structured system variable for job-specific events, and the `^$SYSTEM` structured system variable for system-wide events. 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`, `TRIGGER`, and `USER` event classes. At present, only `INTERRUPT` and `TRIGGER` event classes 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 `^$JOB` or `^$SYSTEM` SSVN to associate a particular event class and event identifier with an event handler routine or subroutine.

Event Blocking and Unblocking
: *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.

32.2 Registering an Asynchronous Event Handler

To register a job-specific event handler that will only execute in the current FreeM process, use the following syntax:

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

To register a system-wide event handler that will execute in every FreeM process, use the following syntax:

```
SET ^$SYSTEM("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`.

32.3 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.

Please note that `ASTART "TRIGGER"` is run implicitly at FreeM startup, to ensure consistency in applications depending on business logic contained in system-wide global triggers. To disable this behavior, add `ASTOP "TRIGGER"` to the `LOCAL.STARTUP` routine in the `USER` namespace. If `LOCAL.STARTUP` does not yet exist in your environment, you may create it by typing `fmadm edit routine USER LOCAL.STARTUP` from your UNIX command-line shell.

32.4 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` SSVN, 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")
```

32.5 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 data. 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.

33 Global Triggers

Global triggers use the FreeM asynchronous event handling subsystem to allow a FreeM process to execute arbitrary M code when a particular action occurs on a particular global.

To set up a global trigger, you must set up an event handler for event class `TRIGGER`. The event identifier must be in the format of `"<action>:<gvn>"`, where <*gvn*> is a global variable name, and <*action*> is one of the following:

`DATA`	Trigger will fire when the `$DATA` intrinsic function is called on <*gvn*>.
`GET`	Trigger will fire when <*gvn*> is read from.
`INCREMENT`	Trigger will fire when intrinsic function `$INCREMENT` is called on <*gvn*>.
`KILL`	Trigger will fire when <*gvn*> is `KILL`ed.
`NEXT`	Trigger will fire when intrinsic function `$NEXT` is called on <*gvn*>.
`ORDER`	Trigger will fire when intrinsic function `$ORDER` is called on <*gvn*>.
`QUERY`	Trigger will fire when intrinsic function `$QUERY` is called on <*gvn*>.
`SET`	Trigger will fire when `SET <gvn>=value` occurs.
`ZDATA`	Trigger will fire when intrinsic function `ZDATA` is called on <*gvn*>.

When a `TRIGGER` event occurs, the `"GLOBAL"` node of the `^$EVENT` structured system variable will be populated with the global reference that invoked the trigger event.

If a `SET` or `KILL` trigger was the source of the `TRIGGER` event, the `OLD_VALUE` node of `^$EVENT` will be populated with original value of `^$EVENT("GLOBAL")` prior to the change, and `NEW_VALUE` will be populated with the new value. This allows triggers to contain logic to undo global changes. This functionality can also be used to provide auditing of specific global changes.

The following example shows a trigger implemented for `SET` operations on the `^DD` global.

```
TRIGGER ;
    ;
    ; Set up a SET trigger on ^DD
    ;
    SET ^$JOB($JOB,"EVENT","TRIGGER","SET:^DD")="ONSET^TRIGGER"
    ;
    ; Enable the TRIGGER event class
    ;
    ASTART "TRIGGER"
    ;
    ; Try setting a node in ^DD
    ;
    SET ^DD(1)="Test"
    ;
    ; Quit
    ;
```

```
        QUIT
        ;
        ;
    ONSET ;
        WRITE "The "_^$EVENT("GLOBAL")_" global node was SET.",!
        QUIT
```

You can also set up a trigger that applies to all FreeM processes by setting descendant subscripts of `^$SYSTEM("EVENT","TRIGGER",...)` instead of using `^$JOB($JOB,"EVENT","TRIGGER",...)`.

34 Synchronous Event Handling

35 GUI Programming with MWAPI

36 User-Defined Z Commands

37 User-Defined Z Functions

38 User-Defined SSVNs

39 Language Dialects

40 System Library Routines

40.1 ^%ZCOLUMNS

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

40.2 %SYSINIT

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.

40.3 ^%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`.

40.4 ^%ZROWS

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

41 Interrupt Handling

When FreeM receives the `SIGINT` signal, either by pressing `Ctrl-C` during program execution, or by external signal from the operating system, the FreeM environment daemon, or another external process, one of two things can happen, depending on the state of the `$ZI` special variable:

`$ZI` evaluates *true*
: In this case, the `ZINRPT` error is raised, and normal error handling procedures apply. If neither `$ZTRAP` nor `$ETRAP` are set, FreeM prints an error diagnostic on the home device and will exit the FreeM process in application mode (i.e., the `freem` executable was started with the `--routine` or `-r` flag), or return to the direct mode prompt otherwise.

  This is the default behavior of FreeM.

`$ZI` evaluates *false*
: In this case, no error is raised, but the `$ZCONTROLC` flag is set. In this mode of operation, it is up to program code to check for `$ZCONTROLC` and take appropriate action.

  Checking the value of `$ZCONTROLC` will reset it to *false.*

In either case, if asynchronous event handling is enabled for the `INTERRUPT` event class (i.e., `ASTART "INTERRUPT"` or `ASTART` have been invoked by the current process), an asynchronous event of event class `INTERRUPT` and event identifier `SIGINT` will be enqueued.

42 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.

43 FreeM Error Codes

`ZINRPT` - *interrupt*
: Raised when an interrupt signal is received.

`ZBKERR` - *BREAK point*
: Raised when a `BREAK` point is reached.

`ZNOSTAND` - *non standard syntax*
: Raised when features incompatible with the current value of `$DIALECT` are used.

`ZUNDEF` - *variable not found*
: Raised when an undefined local or global variable is accessed. This error code has been deprecated in favor of standard error codes `M6` and `M7`.

`ZLBLUNDEF` - *label not found*
: Raised when a referenced label is not found.

`ZMISSOPD` - *missing operand*
: Raised when an operand is missing from an expression.

`ZMISSOP` - *missing operator*
: Raised when an operator is missing from an expression.

`ZILLOP` - *unrecognized operator*
: Raised when an unrecognized operator is encountered in an expression.

`ZQUOTER` - *unmatched quotes*
: Raised when unbalanced quotes are encountered.

`ZCOMMAER` - *comma expected*
: Raised when a comma is expected in program syntax but is not found.

`ZASSIGNER` - *equals '=' expected*
: Raised when an equals sign is expected in program syntax but is not found.

`ZARGER` - *argument not permitted*
: Raised when an argument is encountered in a syntactic position where arguments are not permitted.

`ZSPACER` - *blank ' ' expected*
: Raised when a space character is expected in program syntax but is not found.

`ZBRAER` - *unmatched parentheses*
: Raised when unbalanced parentheses are detected in program syntax.

`ZLVLERR` - *level error*
: Raised when a level error occurs.

`ZDIVER` - *divide by zero*
: Raised when program code attempts to divide by zero. Deprecated in favor of standard error code `M9`.

`ZILLFUN` - *function not found*
: Raised when program code attempts to call intrinsic or extrinsic functions that are not defined.

`ZFUNARG` - *wrong number of function arguments*
: Raised when an intrinsic or extrinsic function is called with the wrong number of arguments.

`ZZTERR` - *ZTRAP error*
: Raised when a `$ZTRAP` error occurs.

`ZNEXTERR` - *$NEXT/$ORDER error*
: Raised when an error occurs in `$NEXT` or `$ORDER`.

`ZSELER` - *$SELECT error*
: Raised when an error occurs in `$SELECT`

`ZCMMND` - *illegal command*
: Raised when program code attempts to execute an illegal command.

`ZARGLIST` - *argument list incorrect*
: Raised when the argument list supplied to an M language element does not match that language element's syntactic requirements.

`ZINVEXPR` - *invalid expression*
: Raised when an invalid expression is encountered.

`ZINVREF` - *invalid reference*
: Raised when an invalid variable reference is encountered.

`ZMXSTR` - *string too long*
: Raised when a string is encountered that exceeds `^$SYSTEM("STRING_MAX")`.

`ZTOOPARA` - *too many parameters*
: Raised when too many parameters are passed to a function or subroutine.

`ZNOPEN` - *unit not open*
: Raised when attempting to access an I/O channel that has not been opened.

`ZNODEVICE` - *unit does not exist*
: Raised when attempting to access a device that does not exist.

`ZPROTECT` - *file protection violation*
: Raised when attempting to access a file or device to which you do not have permission.

`ZGLOBER` - *global not permitted*
: Raised when attempting to use a global in a syntactic element where global variables are not permitted.

`ZFILERR` - *file not found*
: Raised when attempting to access a file that does not exist.

`ZPGMOV` - *program overflow*
: Raised when a program overflows the limits of a routine buffer.

`ZSTKOV` - *stack overflow*
: Raised when `DO`, `FOR`, or `XECUTE` nesting levels exceed the value in `^$SYSTEM("NESTLEVLS")`.

`ZSTORE` - *symbol table overflow*
: Raised when program code attempts to store too much data in the local symbol table. Should not occur unless symbol table auto-adjust is disabled.

`ZNOREAD` - *file won't read*
: Raised when program code attempts to read from an unreadable file.

`ZNOWRITE` - *file won't write*
: Raised when program code attempts to write to an unwritable file.

`ZNOPGM` - *routine not found*
: Raised when an attempt is made to load or execute a routine that does not exist in the current namespace.

`ZNAKED` - *illegal naked reference*
: Raised when an attempt is made to use an illegal naked reference.

`ZSBSCR` - *illegal subscript*
: Raised when an illegal subscript access is attempted.

`ZISYNTX` - *insert syntax*
: Raised when illegal insert syntax is used.

`ZDBDGD` - *global data degradation*
: Raised when corruption is detected in global data files.

`ZKILLER` - *job kill signal*
: Raised on a job kill signal.

`ZHUPER` - *hangup signal*
: Raised on a job hangup signal.

`ZMXNUM` - *numeric overflow*
: Raised when an assignment or expression result exceeds `$ZPRECISION`.

`ZNOVAL` - *function returns no value*
: Raised when a function does not return a value. Extrinsic functions must `QUIT` with a value.

`ZTYPEMISMATCH` - *type mismatch*
: Raised when a type mismatch occurs.

`ZMEMOV` - *out of memory*
: Raised when FreeM runs out of heap memory.

`ZNAMERES` - *error in name resolution*
: Raised when an attempted name resolution fails.

`ZSCKCREAT` - *error creating socket*
: Raised when an error occurs creating a socket for network I/O.

`ZSCKIFAM` - *invalid address family (must be IPV4 or IPV6)*
: Raised when the address family specified in an `OPEN` command for a socket I/O channel is not IPV4 or IPV6.

`ZSCKITYP` - *invalid connection type (must be TCP or UDP)*
: Raised when the connection type specified in an `OPEN` command for a socket I/O channel is not `TCP` or `UDP`.

`ZSCKIPRT` - *invalid port number*
: Raised when the port number specified in an `OPEN` command for a socket I/O channel is invalid. Valid TCP and UDP ports are in the range of 1-65535.

`ZSCKCERR` - *connection error*
: Raised when an error occurs on a `USE <channel>:/CONNECT` command.

`ZSCKAERR` - *USE action invalid for connection type (possibly CONNECT on UDP socket?)*
: Raised when an attempt is made to `USE <channel>:/CONNECT` on a UDP socket I/O channel. The UDP protocol is connectionless.

`ZSCKACON` - *attempted to CONNECT an already-connected socket*
: Raised when an attempt is made to `USE <channel>:/CONNECT` on a TCP socket I/O channel that is already connected.

`ZSCKNCON` - *attempted to READ from a disconnected TCP socket*
: Raised when an attempt is made to `READ` a TCP socket that has not yet been connected.

`ZSCKEOPT` - *error setting socket options*
: Raised when an error is encountered while setting socket options.

`ZSCKERCV` - *error in READ from socket*
: Raised when an error occurs in a socket I/O channel `READ`.

`ZSCKESND` - *error in WRITE to socket*
: Raised when an error occurs while attempting to `WRITE` to a socket I/O channel.

`ZNORPI` - *^$ZRPI only supported on Raspberry Pi hardware*
: Raised when an attempt is made to use the `^$ZRPI` structured system variable on a platform other than the Raspberry Pi single-board computer.

`ZCREDEF` - *cannot redefine CONST*
: Raised when attempts are made to redefine a `CONST` after its initial definition.

`ZCMODIFY` - *cannot modify CONST*
: Raised when attempts are made to change the value of a `CONST`.

`ZFILEXWR` - *cannot open existing file for WRITE*
: Raised when an attempt is made to open an existing file in write (but not append) mode.

`INEWMULT` - *initializing NEW with multiple setarguments not supported*
: Raised when you attempt to use multiple setarguments with initializing `NEW`, e.g. `NEW X=2,Y=3`.

`ZECODEINV` - *invalid value for $ECODE*
: Raised when attempts are made to set `$ECODE` to an invalid error code value. Obsolete and replaced by standard error code `M101`.

`ZASSERT` - *programmer assertion failed*
: Raised when an `ZASSERT` expression's result is not true.

`ZUSERERR` - *user-defined error*
: Raised when program code calls `THROW` with an error code argument for which the first character is `U`, or when `$ECODE` is set to an error code for which the first character is `U`.

Custom error messages for `ZUSERERR` may be set in `^$JOB($JOB,"USER_ERRORS",<user_error_code>)`, where `<user_error_code>` represents the custom error code.

For example:

```
DEFAULT.USER> S ^$JOB($JOB,"USER_ERRORS","UBLACKHOLE")="black hole encounter


DEFAULT.USER> THROW UBLACKHOLE


>> Error UBLACKHOLE:  black hole encountered in SYSTEM::^%SYSINIT  [$STACK =
>> THROW UBLACKHOLE
                  ^
```

`ZSYNTERR` - *syntax error*
Raised when a syntax error without a more specific error code is encountered.

`ZCTRLB` - *break*
Pseudo-error used by the FreeM debugger. Not visibly raised in normal program operation.

`ZASYNC` - *asynchronous interruption*
Pseudo-error used by the FreeM asynchronous events subsystem. Not visibly raised in normal program operation.

`M1` - *naked indicator undefined*
Raised when an attempt is made to use a naked reference before the naked indicator is set.

`M2` - *invalid combination with $FNUMBER code atom*
`M3` - *$RANDOM seed less than 1*
`M4` - *no true condition in $SELECT*
`M5` - *line reference less than zero*
`M6` - *undefined local variable*
`M7` - *undefined global variable*
`M8` - *undefined intrinsic special variable*
`M9` - *divide by zero*
`M10` - *invalid pattern match range*
`M11` - *no parameters passed*
`M12` - *invalid line reference (negative offset)*
`M13` - *invalid line reference (line not found)*
`M14` - *line level not 1*
`M15` - *undefined index variable*
`M16` - *argumented QUIT not allowed*
`M17` - *argumented QUIT required*
`M18` - *fixed length READ not greater than zero*
`M19` - *cannot copy a tree or subtree onto itself*
`M20` - *line must have a formal parameter list*
`M21` - *algorithm specification invalid*
`M22` - *SET or KILL to ^$GLOBAL when data in global*
`M23` - *SET or KILL to ^$JOB for non-existent job number*
`M24` - *change to collation algorithm while subscripted local variables defined*
`M26` - *non-existent environment*
`M27` - *attempt to rollback a transaction that is not restartable*
`M28` - *mathematical function, parameter out of range*
`M29` - *SET or KILL on structured system variable not allowed by implementation*
`M30` - *reference to global variable with different collating sequence within a collating algorithm*
`M31` - *control mnemonic used for device without a mnemonic space selected*
`M32` - *control mnemonic used in user-defined mnemonic space which has no associated line*
`M33` - *SET or KILL to ^$ROUTINE when routine exists*
`M35` - *device does not support mnemonic space*
`M36` - *incompatible mnemonic spaces*
`M37` - *READ from device identified by empty string*
`M38` - *invalid structured system variable subscript*
`M39` - *invalid $NAME argument*
`M40` - *call-by-reference in JOB actual parameter*
`M41` - *invalid LOCK argument within a transaction*
`M42` - *invalid QUIT within a transaction*
`M43` - *invalid range value ($X, $Y*
`M44` - *invalid command outside of a transaction*
`M45` - *invalid GOTO reference*
`M56` - *identifier exceeds maximum length*
`M57` - *more than one defining occurrence of label in routine*
`M58` - *too few formal parameters*
`M60` - *illegal attempt to use an undefined SSVN*
`M101` - *invalid value for $ECODE*
`M102` - *synchronous and asynchronous event processing cannot be simultaneously enabled for the same event class*
`M103` - *invalid event identifier*
`M104` - *ETRIGGER event identifier for IPC event class does not match job process identifier*

44 System Configuration

44.1 Installing FreeM

44.1.1 Installation Methods

FreeM allows the following installation methods:

Binary Repository

On recent versions the Ubuntu and Debian distributions of GNU/Linux, we provide package repositories from which FreeM may easily be installed. See the *FreeM Wiki* for more information, and *https://packages.coherent-logic.com* for instructions.

If available, this is the simplest method of installing FreeM.

Binary Packages

We provide binary packages of FreeM for *dpkg* and *rpm*-based distributions of GNU/Linux, and *pkgadd* packages for Solaris 8-10. If you cannot use repositories, this is the easiest option.

See *https://freem.coherent-logic.com/binaries.cfm* for downloads and instructions.

Source Archive

If you prefer installing from source, we recommend that you download the latest *.tar.gz* file from *https://freem.coherent-logic.com/downloads.cfm*, and follow these steps:

```
$ gunzip freem-<version>.tar.gz
$ tar xf freem-<version>.tar
$ cd freem
$ ./configure # see the Build Configuration section for optional flags
$ make
$ sudo make install
```

Once this process has been completed, you may proceed to *Initial Configuration*.

Installation from source archive is the most challenging but flexible supported option for advanced users.

CVS Repository

If you wish to try the bleeding-edge development version of FreeM, you may do so by following these steps:

```
$ cvs -d :pserver:anonymous@cvs.coherent-logic.com:/home/cvsroot co freem
$ cd freem
$ ./autogen.sh
$ ./configure # see the Build Configuration section for optional flags
$ make
$ sudo make install
```

Once this process has been completed, you may proceed to *Initial Configuration*.

This installation method is by far the most complicated, and is intended only for those who wish to contribute to FreeM development. It is not intended for end users, and no technical support will be provided.

See the *Contributor Guide* on the *FreeM Wiki* for more information.

44.1.2 Build Configuration

When configuring FreeM with the supplied `configure` script, there are some FreeM-specific options that may be used to compile in optional features, or exclude default ones:

`--enable-mwapi` (EXPERIMENTAL)
: Enables experimental support for the M Windowing API (ANSI *X11.6-1995*) using the OSF/Motif widget toolkit. Requires that you have the `X11`, `Xt`, `ICE`, and `Xm` libraries, as well as all of their C header files.

  Please consult your operating system's documentation for the correct commands to install the required libraries.

  Example

```
$ ./configure --enable-mwapi
$ make
$ sudo make install
```

44.1.3 Initial Configuration

Once FreeM is installed, you will need to configure it:

1. Create a user and group, each named *freem*, under which FreeM will run
2. Add any user accounts that will need to run FreeM to the *freem* group
3. Have all users added in step 2 sign out and sign in for the new group membership to take effect
4. Run `fmadm configure` with superuser privileges to create the `DEFAULT` environment with `SYSTEM` and `USER` namespaces and default after-image journal settings, and populate the bundled vendor routines
5. Run `fmadm start environment` with superuser privileges to start the `DEFAULT` environment
6. Make sure the environment is ready by running `fmadm status environment` with superuser privileges

44.1.3.1 Creating Additional Environments

To create additional environments, do the following steps:

1. Create a new user and group for the environment *(optional)*
2. Run `fmadm configure -e=<environment> -u=<username> -g=<groupname> [-E=true|false]` *(the* `-E` *flag enables or disables the environment)*
3. Run `fmadm start environment -e=<environment>` to start the environment
4. Run `fmadm status environment` to make sure the environment is healthy

44.1.3.2 Additional Customization

See the FreeM *environment catalog* at *$PREFIX*`/etc/freem/env.conf`, and the *fmadm*(1) **`man`** page for more information.

$PREFIX represents the root location of your FreeM installation. This can be `/usr/local`, `/`, or others, depending on how FreeM was built and installed.

45 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 data access 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`.

45.1 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 SSVN (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 SSVN 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 SSVN.

`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.

45.2 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.

45.3 freem_init()

Initializes `libfreem` in preparation for calling other APIs.

Synopsis

```
pid_t freem_init(char *environment_name, char *namespace_name);
```

Parameters

`environment_name`
Specifies the environment to use.

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

    return 0;
}
```

45.4 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);

}
```

45.5 freem_set()

Sets a FreeM local node, global node, or writable SSVN 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);

}
```

45.6 freem_get()

Retrieves a FreeM local node, global node, or writable SSVN 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 SSVN */
    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);
}
```

45.7 freem_kill()

Deletes a FreeM local node, global node, or killable SSVN 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);
}
```

45.8 freem_data()

45.9 freem_order()

45.10 freem_query()

45.11 freem_lock()

45.12 freem_unlock()

45.13 freem_tstart()

45.14 freem_trestart()

45.15 freem_trollback()

45.16 freem_tlevel()

45.17 freem_tcommit()

45.18 freem_function()

45.19 freem_procedure()

Appendix A FreeM Administrator

The `fmadm` utility is the preferred method of managing a FreeM installation, and will eventually replace all of the existing utilities. In support of FreeM operators, `fmadm` presents a consistent, simple interface for all FreeM management tasks, and is namespace-aware. This appendix will document each `fmadm` facility as it is implemented.

The `fmadm` utility's functions all follow the below, consistent syntax:

```
usage:  fmadm <action> <object> <namespace> [OPTIONS]
```

The *action* keyword can be one of the following:

list	Lists instances of *object*
examine	Examines a single instance of *object*
verify	Verifies the integrity of *object*
compact	Compacts *object*
repair	Repairs integrity problems in *object*
create	Creates an instance of *object*
remove	Removes an instance of *object*
import	Imports an *object*
export	Exports an *object*
backup	Creates a backup of *object*
restore	Restores a backup of *object*
migrate	Migrates an instance of *object* from an older FreeM version to the current version
edit	Edits an instance of *object*

The *object* keyword can be one of the following:

lock

The FreeM `LOCK` table.

Supported actions are `list` and `remove`.

journal

FreeM after-image journaling.

Supported actions are `examine` and `restore`.

The `examine` action will dump the after-image journal entries for the selected namespace in human-readable format.

The `restore` action will play after-image journals forward for the selected namespace.

namespace

FreeM namespaces (collections of M routines and globals).

No actions yet implemented.

global

The data files representing each FreeM *global*.

Supported actions are `list`, `examine`, `remove`, and `verify`.

routine An M routine, stored as a `.m` file.

Supported actions are `list`, `examine`, `remove`, `import`, `export`, `backup`, and `edit`.

job A UNIX process representing an instance of the FreeM runtime.

Supported actions are `list` and `examine`.

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 29: Copy Symbol Table

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

B.14 VIEW 30: Inspect Arguments

Retrieves the arguments passed to the `freem` executable.

B.15 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 (where `Dollar` should be replaced with a dollar sign):

```
/*
 *   DollarIdDollar
 *    Function prototypes, structs, and macros for FreeM
 *    binding library
 *
 *
 *   Author: Serena Willis <snw@coherent-logic.com>
 *    Copyright (C) 1998 MUG Deutschland
 *    Copyright (C) <Year> Coherent Logic Development LLC
 *
 *   This file is part of FreeM.
 *
 *   FreeM is free software: you can redistribute it and/or modify
 *   it under the terms of the GNU Affero Public License as published by
 *   the Free Software Foundation, either version 3 of the License, or
 *   (at your option) any later version.
 *
 *   FreeM 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 Public License for more details.
 *
 *   You should have received a copy of the GNU Affero Public License
 *   along with FreeM.  If not, see <https://www.gnu.org/licenses/>.
 *
 *   DollarLogDollar
 *
 * SPDX-FileCopyrightText:  (C) 2025 Coherent Logic Development LLC
 * SPDX-License-Identifier: AGPL-3.0-or-later
 **/
```

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)`, and also 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.

Index