X11/TG6/WG1/2020-90

Mumps Development Committee

Task Group 6 · Standards Preparation

Work Group 1 · Integration

Mumps Draft Standard 2020

MDC/TG6 Type C Proposal

Kenneth W. McGlothlen · TG6 Chair

Linda M.R. Yaw · WG1 Chair

2020-04-30

Release

X11/TG6/WG1/2020-90: Mumps Draft Standard 2020

The reader is notified that this document neither reflects MDC Standard specifications nor any implied support by members of Task Group 6 of the Mumps Development Committee or their sponsors, but that it is being offered for possible consideration by the MDC Standards Preparation Task Group. It is made available to establish better communication between the Standards Preparation Task Group and that segment of the public interested in MDC Standard development. This proposal is dynamic in nature, and may not correspond to the latest specification available.

Because of the evolutionary nature of MDC specifications, the reader is reminded that changes are likely to occur in the proposal, herein, prior to a final republication.

1 Identification

1.1 Title

Mumps Draft Standard 2020

1.2 MDC proposer, sponsor, & editor

Proposer: Frederick D. S. Marshall
819 North 49th Street, Suite 203
Seattle, Washington 98103 USA
phone: +1 206-632-0166
rick.marshall@vistaexpertise.net
skype: toad42

1.3 Motion

That TG6/WG1 recommend to TG6 that this nontechnical proposal be adopted as a standing document.

1.4 MDC actions

Date	Document	Action
2020-04-30	X11/TG6/WG1/2020-90	WG1 · ⟨current document⟩ · proposer Frederick D. S. Marshall

2 Justification

2.1 Needs

The American National Standards Institute adopted the previous *Mumps Standard* in 1995.

It is time to update Standard Mumps technology. To create a new *Mumps Standard,* we must first create a draft standard that matches the previously accepted standard, then iteratively apply the selected MDC extensions to it to incrementally create the new *Mumps Standard*.

It is also time to update the 1990s-era typography of the *Mumps Standard,* to make it and other MDC documents consistent and optimized for fully digital rather than paper-based MDC operations.

2.2 Existing practice

There is no existing practice. The MDC has been in administrative hibernation since 1999.

Practice at that time was to iteratively apply extensions to a *Revised Mumps Draft Standard* (*RMDS*). The *RMDS* used typography that did not comply with *X11/95-129 Proposal Format* but was unique to the *Mumps Standard* and optimized for the production of paper MDC pre- and post-meeting mailings.

3 Description

3.1 General description

This standing document is the current *Mumps Draft Standard.*

This draft *Standard* typesets the text of the previous *Standard* according to the guidelines of TG19's *Technical Proposal Format 2020,* which modernizes MDC typography with an eye toward a fully digital rather than paper-based operation.

3.2 Annotated examples of use

The changes that differentiate *Mumps Draft Standard 2020* from *Mumps 1995,* including annotated examples of use, are described in detail in the MDC extensions listed in §5.3 Superseded MDC Documents.

The application of those extensions, including any changes introduced during the process of updating the *Standard,* is documented in §5.6 Document History.

3.3 Formalization

This standing document is the only MDC document that contains the complete formal text of the *Mumps Standard,* in §6 Appendix · Draft Standard.

3.4 Definitions

Likewise, all definitions associated with this document are in the text of the *Standard* itself, in §6 Appendix · Draft Standard.

4 Implementation

4.1 Effect on existing user practices & investments

Since the *Mumps Draft Standard 2020* is largely backward compatible with *Mumps 1995,* the main effect is to add new capabilities to users' repertoires, enhancing their practices and investments. Details beyond that are specific to each MDC extension.

4.1.1 *X11/94-5 Initialising intrinsics*

This extension (approved as MDC Type A in June of 1993) notes that the standard had not specified certain initial values when a process is initiated. Thus, this extension cannot be considered fully backwards compatible, and non-conforming applications may require changes. See **`$device`**, **`$io`**, **`$key`**, **`$principal`**, **`$test`**, **`$x`**, and **`$y`** in §7.1.5.10.

4.1.2 *X11/93-39* `$reference`

This extension (approved as MDC Type A in June of 1993) introduces a new intrinsic variable. Because this is a new feature, legacy code will not be affected at all. The proposal notes that it will significantly improve maintaining code that uses naked references. See **`$reference`** in §7.1.5.10.

4.1.3 *X11/SC12/93-33 Effect of* `close $io`

This extension (approved as MDC Type A in June of 1993) standardizes how **`close $io`** is handled; previously, it was implementation-dependent. As such, it is not a backward-compatible change. See §8.2.7 **`close`**.

4.1.4 *X11/94-4 Two character operators*

This proposed extension (approved as MDC Type A in June of 1993) adds four new relational operators. Thus, it has no effect on legacy code but does allow Mumps developers to improve the readability of many conditional expressions. See **`>=`** and **`<=`** in §7.2.2.2 Numeric relations and **`]=`** and **`]]=`** in §7.2.2.3 String relations.

4.1.5 *X11/94-14 Multiple patatoms within alternation*

This extension (approved as MDC Type A in February of 1994) resolves an omission in the intended alternation syntax for pattern match. It has no effect on legacy code, but does allow for improvements in how patterns are specified. See §7.2.3 Pattern match *pattern* and Annex C Metalanguage element dictionary.

4.1.6 *X11/94-28 Portable string length*

This extension (approved as MDC Type A in June of 1994) doubles the maximum length of a portable string (from 255 characters to 510), but preserves the implicit 255 character limit for individual subscript values. It has no effect on legacy code but improves the range of new portable code. See Portability §2.3.3 Values of subscripts and §2.8 Character strings.

4.1.7 *X11/94-47* New svn *addition:* `$test`

This extension (approved as MDC Type A in December of 1994) adds **`$test`** to the list of *svn*s that may be used with the **`new`** command. It has no effect on legacy code but improves the ability to write modular subroutines. See **`$test`** in §7.1.5.10 and §8.2.22 **`New`**.

4.1.8 *X11/98-30* New `$reference`

This extension (approved as MDC Type A in May of 1998) adds **`$reference`** to the list of *svn*s that may be used with the **`new`** command. It has no effect on legacy code but but improves the ability to write modular subroutines. See **`$reference`** in §7.1.5.10 and §8.2.22 **`New`**.

4.1.9 *X11/96-13 Portable length limit of* names

This extension (approved as MDC Type A in October of 1995) nearly quadruples the maximum length limit on names from eight to 31. It has no effect on legacy code except where routines have non-standard names that exceed the previous eight-character limit, but it improves the range of new portable code. See Portability §2.1 Names and Annex B Error code translations.

4.1.10 *X11/96-7 Lower-case characters in names*

This extension (approved as MDC Type A in October of 1995) allows mixed-case and lowercase names to be portable. It should have no effect on legacy code but improves the range of new portable code. See §6.1 Routine head *routinehead* and Portability §2.1 Names.

4.1.11 *X11/97-23 Portable length limit of strings*

This extension (approved as MDC Type A in March of 1997), despite its name, redefines ten portability limits, fixing four of them at 510 and significantly increasing the other six. It should have no effect on legacy code except where users have taken advantage of implementations that allow longer strings. See Portability §2.3.2, §2.4.2, §2.8, §3.2, §4.1, & §8.

4.1.12 *X11/95-63 Naming string length error*

This extension (approved as MDC Type A in February of 1994) clearly defines what should happen when the string length is exceeded, which was unspecified in previous standards. All known implementations at the time returned with some sort of error, but the *ecode* was implementation-dependent. It should have no effect on legacy code except where users have relied on implementation-specific behavior. See Portability §2.8 Character strings and Annex B Error code translations.

4.1.13 *X11/94-46 `^$global` correction*

This extension (approved as MDC Type A in December of 1994) corrects a signficant flaw in the syntactic definition of the **`^$global`** *ssvn*, by removing the unintended possibility of including an *environment* within its first subscript value. Any legacy code that relies on this unintentional feature should be converted to use the intended syntax, by specifying the *environment* as part of the *rssvn* instead of as part of the global name (see *X11/96-43* ssvn *Formalization* introduces *rssvn*). See §7.1.4.3 **`^$global`**.

4.1.14 *X11/SC13/94-33 Kill data and kill subscripts of glvns*

This extension (approved as SC13 Type A in October of 1993) adds **`kvalue`** and **`ksubscripts`** commands to manage *glvn*s more easily. It has no effect on legacy code, but should allow for some code simplification regarding *glvn* management. See §8.2.19 **`Kill`**.

4.1.15 *X11/95-2 Execution environment*

This extension (approved as MDC Type A in January of 1995) introduces the concept of job environments, describes how environments for new jobs are determined, and upgrades the argument to **`job`** to optionally specify a job *environment*. Previously, these capabilities were done in implentation-specific ways, if they were available at all. Legacy code that relied on implementation-specific features should be examined for compliance to this standard. See §5 Metalanguage description, §7.1.4.5.7 *Ssvn*s specifying default *environment*s, §8.2.18 **`Job`**, use of new metalanguage element *vb* passim, and Annex C Metalanguage element dictionary.

4.1.16 *X11/95-31 `Kill` indirection*

This extension (approved as MDC Type A in January of 1995) fixes a defect in the specification of the exclusive **`kill`** to allow nested indirection, which is allowed in the argument indirection for all other commands. Since this error in the *Standard* has until now gone largely unnoticed, programmers have already been using this feature, which implementors were already providing. See §8.2.19 **`Kill`**.

4.1.17 *X11/95-91 `$order` definition*

This extension (approved as MDC Type A in June of 1995) is simply a fix to a definition; it does not alter the language, but serves to patch up a problematic definition involving subscripts. Legacy code will not be affected in the slightest. See §7.1.6.11 **`$order`**.

4.1.18 *X11/95-94 Parameter passing clarification*

This extension (approved as MDC Type A in October of 1993) is another simple definition fix, in this case of how parameter passing scopes variables. It does not alter the language, and legacy code is unaffected. See §8.1.8 Parameter passing.

4.1.19 *X11/95-96 Spaces at end-of-line*

This extension (approved as MDC Type A in February of 1994) adds the ability to have spaces at the end of a line, which was not allowed by the previous *Standard*. Thus, this extension does not affect legacy code; it adds a new capability programmers may use in their applications going forward. See §6.2.5 Line body *linebody* and §8.1.1 Spaces in *command*s.

4.1.20 *X11/95-116 `^$job` device information*

This extension (approved as MDC Type A in June 1995) adds the ability to get device ownership information for any MUMPS job, including a list of which devices are currently open, which was not available in the previous *Standard*. Legacy code is unaffected by this extension; it will permit programmers at their discretion to replace existing implementor-dependent, non-portable applications with portable ones. See §7.1.4.4 **`^$job`**.

4.1.21 *X11/95-117* Ssvn *collation*

This extension (approved as MDC Type A in June 1995) improves the ability to control the collation of *ssvn* subscripts. This extension affects a very small minority of legacy code; it introduces a new *ssvn* that can be set to create the same behavior as before the extension, so even affected legacy code can continue to operate as before without any code changes. See §7.1.4.9 **`^$system`** and §7.1.6.11 **`$order`**.

4.1.22 *X11/95-118 Undefined* ssvns

This extension (approved as MDC Type A in June 1995) defines the error code to use if an undefined *ssvn* is referenced, when that *ssvn*'s semantics do not specify otherwise. As per the ruling of X11/TG13 Backwards incompatibility, replacing implementation-specific error codes with standard ones is considered a fully backwards-compatible change. Non-portable legacy code can continue to use the implementation-specific code or can convert to the new standard code to become implementation independent. See §7.2 Expression tail *exprtail* and Annex B Error code translations.

4.1.23 *X11/95-119 Extended* extids

This extension (approved as MDC Type A in June 1995) redefines *extid*s to make them case-independent and adds support for implementation-specific **Z**-*extid*s; it also redefines *exttext* to allow empty text and an empty *linebody*. These are new capabilities, so legacy code is not affected. See §6.4 Embedded programs.

4.1.24 *X11/95-132 Parameter passing to a routine*

This extension (approved as MDC Type A in June of 1995) adds the ability to create and use user-defined controlmnemonics and mnemonicspaces. Legacy code is entirely unaffected, but it does allow existing Mumps I/O device handling (usually intricate code) to be greatly simplified and/or standardized. See §8.1.7.2 Label reference *labelref*.

4.1.25 *X11/95-95 Portable* controlmnmonics *and* mnemonicspaces

This extension (approved as MDC Type A in June of 1995) adds the ability to create and use user-defined *controlmnemonic*s and *mnemonicspace*s, as well as the ability for Mumps programmers to implement standard *controlmnemonic*s and *mnemonicspace*s. Legacy code is entirely unaffected, but it does allow existing Mumps I/O device handling (usually intricate code) to be greatly simplified and/or standardized. This should encourage the usage of new device types in applications in the knowledge that these are portable. This would dramatically speed up their implementation in applications such as the VA software which mandate the use of portable code. See "User-defined *mnemonicspace*" in §4 Definitions, §8.1.10 User-defined *mnemonicspace*s, §8.2.34 **Use**, Portability §4.4 Labels, Portability §10 Formats, and Annex C Metalanguage element dictionary.

4.1.26 *X11/96-44 Improve* mnemonicspace *handling*

This extension (approved as MDC Type A in June of 1996) adds the ability to set **$key** to status codes based on the results of *controlmnemonic*s, not just the **Read** command itself; this is essential to giving Mumps developers the ability to implement X3.64 and other *mnemonicspace*s that use **$key** in this way. This and the ability to discover which *mnemonicspace*s are currently active for a device are entirely new functionality, so legacy code is entirely unaffected. See **$key** in §7.1.5.10 and 7.1.4.2 **^$device**.

4.1.27 *X11/97-10* Mnemonicspec *cleanup*

This extension (approved as MDC Type A in March of 1997) makes no change to the Mumps language, just clarifies its semantic description to remove possible confusion, so legacy applications are not affected in any way. See §8.2.23 **Open**.

4.1.28 *X11/96-35 Parameter passing cleanup*

This extension (approved as MDC Type A in March of 1996) makes no change to the Mumps language, just clarifies its semantic description to remove possible confusion, so legacy applications are not affected in any way. See §8.1.8 Parameter passing.

4.1.29 *X11/SC12/98-13 User-definable I/O handling*

This extension (approved as MDC Type A in September of 1998) adds the **`read`** and **`write`** commands to the portable, user-definable I/O controls within a *mnemonicspace*. This has no effect on legacy code, but it greatly improves the ability of Mumps programmers to write portable I/O applications. See §8.2.36 **`Write`**, §8.1.10 User-defined *mnemonicspace*s, and Annex C Metalanguage element dictionary.

4.1.30 *X11/96-41 `String` and `M` collation*

This extension (approved as MDC Type A in March of 1996) redefines the existing character set profiles **`ASCII`** and **`M`**, to standardize how collation algorithms are described, in preparation for adding character set **`ISO-8859-1-USA`**. This extension has no effect on legacy code. See Portability §6 Character set profiles and Annex A Character set profiles.

4.1.31 *X11/96-42 `Charset ISO-8859-1-USA`*

This extension (approved as MDC Type A in March of 1996) adds character set profiles ISO-8859-1-USA and ISO-8859-1-USA/M. This has no effect on legacy code, but it introduces the Latin 1 suite of characters to Mumps, which can be used to write more international software than ASCII. See §4 Definitions, and Annex A Character set profiles.

4.1.32 *X11/96-45 `Charset` names*

This extension (approved as MDC Type A in June of 1995) redefines how character set profiles can be named. Since such naming is a standardization activity rather than a user-facing feature, this has no effect on legacy code. See §9 Character set profile *charset* and Annex C Metalanguage element dictionary.

4.1.33 *X11/96-43 `Ssvn` formalization*

This extension (approved as MDC Type A in March of 1996) redefines *ssvn*s to match expectations and to align with how locals and globals are defined, explicitly adding support for name indirection, subscript indirection, environments, the assignment of **`^$z`** *ssvn* names to implementor expansion, and the reservation of all other *ssvn* names for future MDC expansion. None of this should affect legacy code, other than perhaps making existing practice officially recognized by the *Standard.* See §7.1.3.1 Local variable name *lvn*, §7.1.4 Structured system variable *ssvn*, and Annex C Metalanguage element dictionary.

4.1.34 *X11/98-5 Fix `algoref`*

This extension (approved as MDC Type A in August 1997) fixes a defect in the definition of the algorithm references in **`^$character`**, when they take the form of an external reference. Since all implementors understood the intent of this defective metalanguage, this extension has no effect on legacy code, just brings the *Standard* into alignment with it. See §7.1.4 **`^$character`**.

4.1.35 *X11/98-26 Canonic form of* ssvn *name*

This extension (approved as MDC Type A in June of 1998) fixes a defect in the definition of the *namevalue* data type, in which the syntax of *namevalue*s for *ssvn*s is unclear, leading to confusion about what, for example, the result of **`$query(^$c("M"))`** would be. This extension changes the *Standard* to match the intended canonic values, which all implemenations return in these cases, so legacy code should be unaffected. See §7.1.5.12 Name value *namevalue*, step ***e***.

4.1.36 *X11/98-8* Ssvn *for user/group identification*

This extension (approved as MDC Type A in March of 1998) introduces new nodes in **`^$job`** to define a process's user and group for security purposes. Its use is optional, so legacy code is unaffected. See §4 Definitions, §7.1.4.4 **`^$job`**, and Annex B Error code translations.

4.1.37 *X11/98-29 Local variables in* `^$job`

This extension (approved as MDC Type A in June of 1998) introduces new nodes in **`^$job`** to list a process's local variable names. Legacy code is unaffected; any legacy code that relies on the nonstandard unsubscripted **`$order`** function can be converted to this new standard solution. See §7.1.4.4 **`^$job`** and Annex C Metalanguage element dictionary.

4.1.38 *X11/98-19 User-defined* ssvns

This extension (approved as MDC Type A in June of 1998) introduces the capability for Mumps programmers to define and implement their own *ssvn*s. Legacy code is unaffected; however this additional functionality could be used as a means of easy encapsulation of such code, as well as allowing better techniques for data and interface manipulation. See §7.1.4 Structured system variable *ssvn*, §7.1.4.10 **`^$y[`***unspecified***`]`**, and Annex B Error code translations.

4.1.39 *X11/96-51 Device environment*

This extension (approved as MDC Type A in October 1995) introduces the capability to specify device environments. As an entirely new capability, this does not affect legacy code, but it does introduce the ability to replace existing implementation-specific algorithms with portable ones. See §7.1.4.4.5 *ssvn*s specifying default *environment*s, §7.1.5.10 Intrinsic special variable names *svn*, §8.2.7 **`Close`**, §8.2.23 **`Open`**, and §8.2.34 **`Use`**, and Annex C Metalanguage element dictionary.

4.1.40 *X11/97-31 Output time out*

This extension (approved as MDC Type A in September of 1997) introduces the ability to set up a timeout on all output, specified not by syntax changes to the **`read`** or **`write`** commands but with an **`outtimeout`** device parameter, and signaled not by setting **`$test`** but by causing an error condition. This is a new capability; legacy code is not affected, but may be adjusted to take advantage of the new feature to stop output from a blocked device from locking up the process. See §7.1.4.2 **`^$device`**, §8.3.1 Output time out, and Annex B Error code translations.

4.1.41 *X11/SC12/98-11 Output time out initialized*

This extension (approved as MDC Type A in September of 1998) eliminates a potential source of unexpected errors in legacy software, if it was written based on *X11/97-31 Output time out*. If, far more likely, legacy software was not based on it, then *X11/SC12/98-11* has no effect on legacy software. See §7.1.4.2 **`^$device`**, §8.2.7 **`Close`**, and §8.3.1 Output time out.

4.1.42 *X11/98-23 `Open` command clarification (re-`open`)*

This extension (approved as MDC Type A in June of 1998) specifies what happens if Mumps software issues a second **`open`** command on a device already opened by that process but with different device parameters. The extension aligns the standard with the practice of all implementations at the time, so it should have no effect on legacy code. See §8.2.23 **`Open`**.

4.1.43 *X11/96-52 Routine management*

This extension (approved as MDC Type A in October of 1995) introduces a pair of new commands to portably load and save Mumps routines. It has no effect on legacy code, though it will let some platform-specific routine-management software to be rewritten in portable ways. See §8.2.24 **`Quit`**, §8.2.26 **`Rload`**, §8.2.27 **`Rsave`**, Annex B Error code translations, and Annex C Metalanguage element dictionary.

4.1.44 *X11/SC15/98-8 `$mumps` function*

This extension (approved as MDC Type A in June or September of 1998) introduces a powerful new function that can be used to check the syntax of Mumps routine *line*s. It has no effect on legacy code, but it can and will be used to rewrite some fiendish and implementation-dependent code currently used for routine parsers, editors, validators, and the like. Its use will enable more errors to be caught at development time rather than run time, increasing Mumps software safety. See §7.1.6 Intrinsic function *function*, §7.1.6.10 **`$mumps`**, Appendix B Error code transations, and Appendix C Megalanguage element dictionary.

4.1.45 *X11/98-24 Duplicate keywords clarified*

This extension (approved as MDC Type A in June of 1998) clarifies the semantics of command parameters to match the current implementations. It therefore has no effect on legacy code, just making the standard a little more understandable. See §8.2.23 **`Open`**, §8.2.26 **`Rload`**, §8.2.27 **`Rsave`**, §8.2.33 **`Tstart`**, and §8.2.34 **`Use`**.

4.1.46 *X11/SC12/98-14 Undefined* devicekeyword

This extension (approved as MDC Type A in September of 1998) defines the error codes for attempts to refer to an undefined device keyword or attribute. It therefore has no effect on legacy code, except to allow any code that handles such scenarios to become more portable. See §8.2.7 `Close` and Annex B Error code translations.

4.1.47 *X11/98-25 Device parameter issues*

This extension (approved as MDC Type A in June of 1998) introduces an optional / character that can be inserted ahead of keyword or attribute format device parameters to unambiguously distinguish them from the old expression format. It has no effect on legacy code. See §8.2.7 `Close`.

4.1.48 *X11/96-9 Pattern negation*

This extension (approved as MDC Type A in October of 1995) introduces a unary not operator (`'`) that can be applied to pattern codes and string literals within patterns. It has no effect on legacy code, but it can be used to simplify some complicated patterns in existing code. See §7.2.3 Pattern match *pattern* and Annex C Metalanguage element dictionary.

4.1.49 *X11/97-3 Pattern ranges*

This extension (approved as MDC Type A in August of 1996) introduces pattern sets—specified as string literals, `$char` values, and/or ranges—that can be used in pattern match as a kind of ad hoc pattern code for greater flexibility and expressiveness. It has no effect on legacy code, but it can be used to simplify some complicated patterns in existing code. See §5 Metalanguage description, §7.2.3 Pattern match *pattern*, and Annex C Metalanguage element dictionary.

4.1.50 *X11/98-27 Pattern match string extraction*

This extension (approved as MDC Type A in June of 1998) extends pattern matching to allow the matching substrings to be saved off into variables. It has no effect on legacy code, but it can be used to significantly simplify existing code that at present first uses pattern match to validate a string and then uses separate Mumps algorithms to find and extract the matching substrings, presently a clumsy, two-part operation. See §7.2.3 Pattern match *pattern* and Annex C Metalanguage element dictionary.

4.1.51 *X11/96-34 Modulo by zero*

This extension (approved as MDC Type A in June of 1993) defines the result of modulo by zero as the divide-by-zero error, `M9`, the same as any other division by zero. It has no effect on legacy code, since it standardizes what the implementors are already doing. See §7.2.1.2 Arithmetic binary operators.

4.1.52 *X11/96-27 Xor operator*

This extension (approved as MDC Type A in March of 1996) adds `!!` as a new logical xor operator and `'!!` as xnor. It has no effect on legacy code, but may be used to simplify some existing truth-value expressions. See §7.2.2.4 Logical operator *logicalop*.

4.1.53 *X11/SC13/97-9 Mathematics errors*

This extension (approved as MDC Type A in September of 1997) defines four boundary arithmetic situations as specific errors. Programs that contain only valid mathematical calculations will continue to run without change. Programs that contain invalid mathematical calculations that were not trapped by their implementation will now crash. Note that in some cases, a program could be valid overall, yet still contain an invalid computation. See §7.2.1.2 Arithmetic binary operators, Portability §2.6 Number range, and Annex B Error code translations.

4.1.54 *X11/96-10 Reverse* `$query`

This extension (approved as MDC Type A in September of 1995) adds a second argument to `$query` to support reverse `$query`, similarly to reverse `$order`. Many implementations already implement reverse `$query`, and many Mumps developers do not realize it was not previously part of the *Mumps Standard*. This extension does not disrupt legacy code, just adds a new and valuable capability it can use. See §7.1.6.16 `$query`.

4.1.55 *X11/SC13/98-15 Definition of reverse* `$query`

This extension (approved as MDC Type A in September of 1998) fixes a problem with the *Mumps Standard*'s definition of reverse `$query`. It has no effect on legacy code, since all implementations that implement reverse `$query` already do it correctly. See §7.1.6.16 `$query`.

4.1.56 *X11/97-22* `Set $qs[ubscript]` *pseudo function*

This extension (approved as MDC Type A in March of 1997) adds a new pseudo-function to Mumps to let programmers change subscripts in a *namevalue* with a single `set` command. It has no effect on legacy code, but it makes it possible to rewrite some ugly indirection-related code into something shorter and more readable, increasing software safety. See §8.2.28 `Set` and Annex B Error code translations.

4.1.57 *X11/96-68 Negative subscripts in* namevalue

This extension (approved as MDC Type A in September of 1996) fixes the *Standard* to match its intent and to match the practice of all implementations and Mumps developers, to allow negative subscripts in *namevalues*. It has no effect on legacy code. See §7.1.5.12 Name value *namevalue*.

4.1.58 *X11/SC13/98-13 Define variable* `m` *in* `$piece`

This extension (approved as MDC Type A in September of 1998) fixes the Standard to match its intent and to match the practice of all implementations and Mumps developers, to correctly define the semantics of the four-argument `$piece` function. It has no effect on legacy code. See §7.1.6.13 `$piece`.

4.1.59 *X11/SC13/TG6/98-3* `$horolog` system function

This extension (approved as MDC Type A in September of 1998) adds a new intrinsic function to provide timezone, UTC, and fractional-second information. It has no effect on legacy code, but it will permit some implementation-specific code to be rewritten using these new portable features. See §4 Definitions, §7.1.6 Intrinsic function *function*, and §7.1.6.8 `$horolog`.

4.1.60 *X11/SC13/TG3/98-4 Data record functions*

This extension (approved as MDC Type A in September of 1998) adds two new intrinsic functions (`$dextract` and `$dpiece`) and two quasi-functions (`set $dextract` and `set $dpiece`) to add the ability to get and set multiple extracts and pieces in a single operation. It has no effect on legacy code, but it will let Mumps developers replace a great deal of cumbersome existing code (since these are common operations in Mumps) with this new, more streamlined approach. See §4 Definitions, §7.1.6 Intrinsic function *function*, §7.1.6.4 `$dextract`, §7.1.6.5 `$dpiece`, and Annex C Metalanguage element dictionary.

4.1.61 *X11/96-11* Fncode *correction*

This extension (approved as MDC Type A in September of 1995) reserves all fncodatoms not currently defined by the *Standard.* This will impact any sloppy legacy code that violates *Mumps 1995* by including nonstandard *fncodatom*s, which on some implementations may have been ignored in the past but now will generate a syntax error, but standards-compliant legacy code will be unaffected. See §7.1.6.8 `$fnumber`.

4.1.62 *X11/96-32 Sign of zero in* `$fnumber`

This extension (approved as MDC Type A in March of 1996) clarifies that neither the `-` nor the `+` should prefix `0` as a result of using the `$fnumber` function. This ought to have no effect on legacy code, but users should verify the behavior of their implementation. Routines intended to be portable and that depend on the functions specified herein may need to be changed, if the programmers took advantage of idiosyncratic interpretations of `$fnumber`. See §4 Definitions and §7.1.6.8 `$fnumber`.

4.1.63 *X11/96-67 Leading zero in* `$fnumber`

This extension (approved as MDC type A in September of 1996) clarifies that the two- and three-argument forms of `$fnumber` format numbers `>-1` but `<1` differently; the two-argument form returns a Standard Mumps canonic number, with no leading `0` before the decimal point, but the three-argument form does return a value with the leading `0`. This does not change the behavior according to the *Standard* nor the implementations, so it should have no effect on legacy code, just reduce potential confusion. See §7.1.6.8 `$fnumber`.

4.1.64 *X11/96-57 `Goto` rewording*

This extension (approved as MDC Type A in March of 1996) clarifies the conditions under which the **`goto`** command can target any given *line*. It brings the specification into alignment with intent and universal practice, just rewrites a confusing sentence, so it has no effect on legacy code. See §8.2.14 **`Goto`**.

4.1.65 *X11/96-58 Add `job` to routine execution*

This extension (approved as MDC Type A in March of 1996) clarifies the conditions under which blocks are invoked by making clear the **`job`** command can invoke them following the same rules as the **`do`** command and extrinsic functions and variables. It brings the specification into alignment with intent and universal practice, so it has no effect on legacy code. See §6.3 Routine execution.

4.1.66 *X11/SC15/98-42 Subscript indirection and `lock`*

This extension (approved as MDC Type A in March of 1998) extends the specification of the **`lock`** command to support subscript indirection, as originally intended when the **`lock`** command was introduced. It is universally implemented and widely used, and it came as a surprise to the MDC to discover the Mumps Standard did not yet specify it. It has no effect on legacy code, other than perhaps to promote the use of this feature. See §8.2.19 **`Lock`** and Annex C Metalanguage element dictionary.

4.1.67 *X11/96-49 `Quit` with argument in `for` command*

This extension (approved as MDC Type A in March of 1996) clarifies that execution of a **`quit`** with an argument in the scope of a **`for`** should produce an error condition with *ecode*=**`"M16"`**, which is already how this is universally implemented. It therefore has no effect on legacy code. See §8.2.13 **`For`**.

4.1.68 *X11/98-31 `If then & else`*

This extension (approved as MDC Type A in June of 1998) adds a new **`then`** command that can be used with **`if`** and **`else`** and timeouts to protect **`$test`** for the rest of the line, to avoid the classic **`if`**-**`else`** vulnerabilities that have plagued Mumps software since its inception. It has no effect on legacy code, but it will be very useful there, to insert into legacy code to fix existing **`$test`** problems or to prevent them when updating the code, as well as in new Mumps software. It is likely to be a widely used command, taught to all beginning Mumps programmers. See §8.2.30 **`Then`**.

4.1.69 *X11/97-25 First line format*

This extension (approved as MDC Type A in September of 1997) strikes Annex E Routine identification, first line convention in its entirety. Since this annex was only informative, not normative, its removal from the Mumps Standard has no effect on legacy code.

4.1.70 *X11/96-65 Normalize definition of* `tstart`

This extension (approved as MDC Type A in February of 1995) is a refinement of the syntax specification, with no effect on Mumps software, legacy or new. See §8.2.33 **`Tstart`**.

4.1.71 *X11/SC15/98-5 Error handling corrections*

This extension (approved as MDC Type A in June of 1998) is a refinement of the syntax specification, with no effect on Mumps software, legacy or new. See §6.3.2 Error processing, §7.1.5.10 Intrinsic special variable name *svn*, §7.1.6.23 **`$stack`**, and §8.2.28 **`Set`**.

4.1.72 *X11/SC13/TG15/97-3 Local variable storage*

This extension (approved as MDC Type A in June of 1998) removes the symbol table portability limit, rendering **`$storage`** ill-defined, requires implementors to put their symbol table size guarantees in the implementation's conformance clause, and updates the definitions of local and global variables. It has no direct effect on legacy code, but it makes it harder to plan system sizes, allocate system resources fairly, or design software that makes efficient use of storage. See §3.1 Implementations, §7.1.2 Variables, and Portability §8 Storage space restrictions.

4.1.73 *X11/98-14 Sockets binding*

This extension (approved as MDC Type A in March of 1998) adds a new standard **`SOCKETS`** *mnemonicspace* and associated parameters and *controlmnemonic*s, to give Mumps the standard, portable ability to manage socket devices. It has no direct on legacy code, but it will allow a great deal of implementation-specific, non-portable socket-management software in Mumps to be rewritten as Standard, implementation-independent, portable software. See Annex B Error code translations and Annex H Sockets binding.

4.1.74 *X11/98-28 Event processing*

This extension (approved as MDC Type A in June of 1998) adds seven new commands and a new *ssvn* to Mumps to introduce even processing, building on the approach used in *X11.1.6-1995 M Windowing API*. It is one of the largest extensions ever made to the Mumps language, but it consists entirely of new features, so it has no impact on legacy code—though it may allow some existing non-portable and implementation-specific event-oriented applications to be rewritten to become portable and implementation-independent. See §2 Normative references, §3.1 Implementations, §6.3.3 Event processing, §7.1.3.3 Process stack, §7.1.4 Structured system variable name *ssvn*, §7.1.4.3 **`^$event`**, §7.1.4.5 **`^$job`**, §8.2.1 **`Ablock`**, §8.2.3 **`Astart`**, §8.2.4 **`Astop`**, §8.2.5 **`Aunblock`**, §8.2.10 **`Estart`**, §8.2.11 **`Estop`**, §8.2.12 **`Etrigger`**, Portability §12 Event processing, Portability §13 Other portability requirements, Annex B Error code translations, and Annex C Metalanguage element dictionary.

4.1.75 *X11/SC15/98-11 Generic command indirection*

This extension (approved as MDC Type A in September of 1998) adds a new form of indirection, generic command indirection, which allows any part(s) of a command to be evaluated indirectly, from any single character up to and including the entire command itself; it cannot cross command or line boundaries. This new feature has no effect on existing legacy code, but it can be used to rewrite quite a few examples of complicated command-argument indirection, resulting in far more readable and maintainable code, which can increase software safety and decrease development and maintenance costs. See §8.1.4 Generic command indirection.

4.1.76 *X11/SC15/TG2/98-2 Object usage*

This extension (approved as MDC Type A in June of 1998) adds features to permit simple use of objects in Mumps, assuming they have been created through some mechanism not identified by this proposal (such as objects created by other programming languages or the operating system). It has no effect on legacy code, but it may allow some implementation-dependent object usage applications to be rewritten to be portable. Programmers will be able to incorporate use of objects into existing Mumps applications. See §6.3.2 Error processing, §7.1 Espression atom *expratom*, §7.1.1 Values, §7.1.3.2 Local variable handling, §7.1.5.3 Numeric data values, §7.1.6 Intrinsic function *function*, §7.1.6.26 **`$type`**, §7.2.2.1 Relational operator *relation*, §7.2.2.3 String relations, §8.1 General command rules, §8.1.8 Parameter passing, §8.2.2 **`Assign`**, §8.2.8 **`Do`**, §8.2.24 **`Quit`**, §8.2.28 **`Set`**, §8.1.9 Object usage, Annex B Error code translations, and Annex C Metalanguage element dictionary.

4.1.77 *X11/94-23 Library proposal*

This extension (approved as MDC Type A in June of 1994) adds features to permit the definition and use of standard libraries. It has no effect on legacy code, but it can be used to speed up development of new applications, increasing reuse and reducing redundant development. See §4 Definitions, §7.1.4.5 **`^$job`**, §7.1.4.6 **`^$library`**, §7.1.5.8 Extrinsic function exfunc, §7.1.5.9 Extrinsic special variable, §8.1.7.4 Library reference, § 7.1.7 Library, and Annex C Metalanguage element dictionary.

4.1.78 *X11/95-11 Library functions, general mathematics*

This extension (approved as MDC Type A in January of 1995) adds a new **`MATH`** library and a suite of general math functions. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.79 *X11/95-12 Library functions—tringomoetry*

This extension (approved as MDC Type A in January of 1995) adds a new suite of trigonometry functions to the **`MATH`** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.80 *X11/95-13 Library functions—hyperbolic tringomoetry*

This extension (approved as MDC Type A in January of 1995) adds a new suite of trigonometry functions to the **MATH** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.81 *X11/96-26 Library functions—matrix mathematics*

This extension (approved as MDC Type A in March of 1996) adds a new suite of matrix functions to the **MATH** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.82 *X11/95-22 "Standard" in library element definitions*

This extension (approved as MDC Type A in January of 1995) tunes up the definition of the term "standard" when used to describe library elements. It does not change the Mumps language, so legacy code is not affected. See §7.1.7.2 Library element definitions.

4.1.83 *X11/95-14 Library functions—complex mathematics*

This extension (approved as MDC Type A in January of 1995) adds a new suite of complex number functions to the **MATH** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.84 *X11/95-112* `REPLACE` *library function*

This extension (approved as MDC Type A in June of 1995) adds a new find-and-replace function to the **STRING** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.85 *X11/95-111* `PRODUCE` *library function*

This extension (approved as MDC Type A in June of 1995) adds a new string-production function based on find-and-replace to the **STRING** library. It has no effect on legacy code, and users may gain precision and performance by switching from user-written approximations to these standard library functions. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.86 *X11/SC13/TG5/96-5 Corrections to library functions*

This document is not an extension. It is editorial direction to fix the sample code in the previous library functions; those extensions will better achieve their goals for Mumps programmers with these bugs fixed. Otherwise, it has no effect on legacy code. See Annex I

Mumps standard library sample code. Doublecheck MDC vote as to editorial versus substantive when this was presented.

4.1.87 *X11/96-74 Operator overrides*

This extension (approved as MDC Type A in October 1996, what was the editorial amendment?) introduces a new CHARACTER library and two library functions, **$%COLLATE^CHARACTER** and **$%COMPARE^CHARACTER** to assist in evaluating collation and developing character set profiles. It has no effect on legacy code. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.88 *X11/98-21 Miscellaneous character functions*

This extension (approved as MDC Type A in March of 1998) adds two new *ssvn* nodes to **^$character** and two new **STRING** library functions to aid in upper and lower case conversions. It has no effect on legacy code, but converting to the use of these library functions has the advantage that unlike all known nonstandard Mumps solutions, these will work regardless of which character set profile is in effect, since they are tied directly to the current character set profile's definition. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.89 *X11/98-32 Cyclic redundancy code functions*

The extension (approved as MDC Type A in September of 1998) adds three CRC functions to the **STRING** library. It has no effect on legacy code, but permits conversion away from equivalent proprietary **$Z** functions or hand-implemented versions, for greater portability and standardization. See §7.1.7 Library and Annex I Mumps standard library sample code.

4.1.90 *X11/SC13/98-10 FORMAT library function, revised yet again*

The extension (approved as MDC Type A in September of 1998) adds **$%FORMAT^STRING** to provide an extremely flexible number and currency formatting function, rendering **$fnumber** obsolete. It has no effect on legacy code, but will make it possible to retire implementation-specific calls or custom subroutines in favor of this new, powerful, standard function. See Library and Annex I Mumps standard library sample code.

4.2 Effect on existing implementor practices & investments

None yet.

As implementors implement these new MDC extensions, and as new extensions are added to this *Mumps Draft Standard,* this section will be extended with one subsection for each extension, to describe progress toward implementing each extension. The implementations surveyed include:

46 ISM · Cache for UNIX (Ubuntu Server LTS for x86-64) 2018.1.2 (Build 309_5) Wed Jun 12 2019 20:06:03 EDT

47 GTM · YottaDB r1.22 Linux x86_64 [GT.M V6.3-004 Linux x86_64]

49 FM · FreeM 0.1.6
50 MV1 · MUMPS V1.71 for Linux x86_64 Built Feb 16 2020 at 07:42:15

Other implementations to survey include:

48 PSM · M3-Lite
51 M21
52 MiniM
53 MII · Mumps-II

Note: the version information comes from the implementation-specific **$zversion** intrinsic variable, except for the YottaDB info, which comes from **$zyrelease**.

4.2.1 *X11/94-5 Initialising intrinsics*

From the proposal · Some implementors will have to change their implementations, but this should require a small amount of effort. [finish implementation table describing implementor progress toward implementing *Initialising intrinsics*]

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.2 *X11/93-39 $reference*

From the proposal · Minor, many implementers already offer this functionality.

ISM Not implemented.

GTM Partly implemented.

a default and initial values function as specified
b **set $reference** not implemented

FM implemented?

MV1 Partly implemented.

a default and initial values function as specified
b **set $reference** not implemented

4.2.3 *X11/SC12/93-33 Effect of* `close $io`

From the proposal · In a 1989 survey conducted by the MUMPS Users' Group, twelve (40%) of the thirty implementations surveyed conformed to the 1984 standard. Of these, seven implementations performed the **open** *p* **use** *p*, two performed a **use** *p* (the principal device is not **close**able), and three gave **$io** a null value.

Of the remaining eighteen (60%) non-standard implementations. twelve gave **$io** the value of the principal device without first performing the required **open** *p* **use** *p* and could produce an error if the principal device was not owned. One implementation does not perform the **open** *p* **use** *p*, but will allow a write to the device *p* even if it is not owned by the job. Two implementations do not change **$io** (**$io** identifies the device just **close**d). One implementation makes **$io** undefined.

For those implementations that currently [in 1993] adhere to the standard, there are two possibilities:

1. Currently the system performs the `open` *p* `use` *p* (30% of the implementations surveyed). For those cases when the principal device is currently owned by another job, processing waits until ownership of the principal device is obtained.

Under this proposal, processing would continue and could produce an error in those cases where a process performs a `close $io` and assumes `$io` is then set to the principal device. Under these circumstances, this proposal is not backwards compatible. Under this proposal, if the old method is desired (that is, wait until the principal device is obtainable), the following code can be used:

```
close $io open $principal use $principal
```

2. The system currently gives `$io` the empty value (10% of the implementations surveyed). For those cases when the principal device is currently owned by another job, a process must explicitly `open` and `use` some device before performing any further I/O. Under this proposal, processing would continue as before. Therefore, there is no backwards-compatibility issue.

In those cases where the system does not follow the standard to begin with (60% of the implementations surveyed), these implementations would continue to be non-standard.

2020 · [insert implementation table describing implementor progress toward implementing *Initialising intrinsics*]

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.4 *X11/94-4 Two character operators*

From the proposal · Modest change to parsing of relational operators.

ISM Partly implemented.
 a `<=` and `>=` operators function as specified
 b `]=` and `]]=` operators not implemented
GTM Partly implemented.
 a `<=` and `>=` operators function as specified
 b `]=` and `]]=` operators not implemented
FM implemented?
MV1 Not implemented.

4.2.5 *X11/94-14 Multiple patatoms within alternation*

From the proposal · “At least one implementor is known to have already implemented this proposal.”

2020 · as described below:

ISM Fully implemented.
GTM Fully implemented.

FM implemented?

MV1 Fully implemented.

4.2.6 *X11/94-28 Portable string length*

From the proposal · “Small (according to a straw poll among the major implementors).”

2020 · Current implementor string-length limits are as follows (kudos to Implementation Work Group member David L. Wicksell for doing this research):

ISM Fully implemented (Cache distinguishes between normal mode and big string mode, so both limits are listed).
maximum local variable value length: 32 KiB/3,641,144 characters (~3.5 MiB)
maximum local variable key length: 506/506 characters
maximum global variable value length: 32 KiB/3,641,144 characters (~3.5 MiB)
maximum global variable key length: 506/506 characters

GTM Fully implemented.
maximum local variable value length: 1 MiB
maximum local variable key length: 1,019 characters
maximum global variable value length: 1 MiB
maximum global variable key length: 1,019 characters

FM implemented?

MV1 Fully implemented.
maximum local variable value length: 32,767 characters
maximum local variable key length: 32,767 characters
maximum global variable value length: 32,767 characters
maximum global variable key length: 32,767 characters

4.2.7 *X11/94-47* New svn *addition:* `$test`

From the proposal · “None expected.”

2020 · Current implementations

ISM Not implemented.

GTM Not implemented. [it is in YottaDB r1.28!!!]

FM implemented?

MV1 Not implemented.

4.2.8 *X11/98-30* New `$reference`

From the proposal · No effect on existing user practices and investments expected.

ISM Not implemented.

GTM Not implemented.

FM implemented?

MV1 Not implemented.

4.2.9 *X11/96-13 Portable length limit of* names

From the proposal · “Some implementers already use characters beyond the eighth to determine **name** uniqueness. Implementers may have to modify their data structures and/or symbol table algorithms to accommodate longer **name**s."

2020 · current implementation behavior for max local name length (global and label and routine name lengths not yet tested):

ISM Partially implemented. Max local name length is 31, but ISM still ignores longer names instead of generating error **M56**.

GTM Partially implemented. Max local name length is 31, but GTM still ignores longer names instead of generating error **M56**.

FM implemented?

MV1 Not implemented. Max local name length is still 8, and MV1 still ignores longer names instead of generating error **M56**.

4.2.10 *X11/96-7 Lower-case characters in* names

From the proposal · All implementations known to the author treat names as being case-sensitive. Therefore, this extension has no effect on existing implementor practices and investments.

2020 · [only locals and globals tested so far, also test labels and routines]

ISM Fully implemented [locals, globals]

GTM Fully implemented [locals, globals]

FM Partially implemented [locals, globals]

MV1 Partially implemented [locals, globals]

4.2.11 *X11/97-23 Portable length limit of strings*

From the proposal · Implementers may have to modify their data structures to accommodate longer local variable strings. Some implementers already have. This will probably be mitigated by the fact that other language extensions (e.g., multinational or multibyte character sets) will also necessitate rethinking of internal data structures. Two implementors have stated that 32,767 would not be a problem.

Some implementations allow increased string lengths. If an application takes advantage of the increase it becomes non-standard and possibly non-portable.

2020 · [test array ref, string, result, line, routine, symbol table]

ISM implemented? [array ref, string, result, line, routine, symbol table]

GTM implemented? [array ref, string, result, line, routine, symbol table]

FM implemented? [array ref, string, result, line, routine, symbol table]
max array reference length: 256 chars
max string length: 256 chars (same for local, global, & ssvn)
max intermediate/final result length: 256 chars
max routine line length: 256 chars
max routine size: no implementer-imposed limit
max symbol table size: no implementor-imposed limit

MV1 implemented? [array ref, string, result, line, routine, symbol table]

4.2.12 *X11/95-63 Naming string length error*

From the proposal · Section one of the M standard defines the syntax of strings and section two defines the portability requirements. Neither identifies what should occur when the string length is exceeded. Some ludicrous possibilities would be:

- truncate excess characters from the end of the string
- truncate excess characters from the beginning of the string
- randomly remove characters until the length is equal to the limit

No implementation known to the author chooses any of these options; they all return some kind of error. However, it's currently up to the implementor as to what the *ecode* will be. This proposal specifies what the *ecode* will be for a string-length error.

It's not anticipated that this proposal will have any other impact beyond specifying what *ecode* value a string-length error produces.

2020 · [test string length]

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.13 *X11/94-46* `^$global` *correction*

From the proposal · To the author's knowledge one implementor has released the `^$global` *ssvn* with the *gvnexpr V gvn* form of the *ssvn*. Another implementor has implemented `^$global` but has indicated that the final form will depend on the action on this proposal. Note that prior to the June 93 MDC meeling, the `^$global` formalization was significantly different.

The implementors may need to rewrite existing `^$global` definition/code.

2020 · [test `^$global` syntax allowed]

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.14 *X11/SC13/94-33 Kill data and kill subscripts of* glvns

From the proposal · No effect on current implementor practices and investments.

2020 · ISM and GTM (but not MV1) have implemented a `zkill` command equivalent to `kvalue` (investigate more closely) but not yet anything like `ksubscripts`.

ISM	Not implemented
GTM	Not implemented
FM	implemented?
MV1	Not implemented

4.2.15 *X11/95-2 Execution environment*

From the proposal · In the current Mumps standard, the `job` command can start another process. The standard does not address the issue of the value of `$system` for the new process and the default environments for `ROUTINE`, `GLOBAL`, and `LOCK`. This proposal addresses these issues.

Most implementations allow Mumps programmers to specify `SYSTEM` and `UCI` in a non-standardized way that implicitly determines the `ROUTINE`, `GLOBAL`, and `LOCK` environments.

The syntax chosen in this MDC extension is the same as used for `ROUTINE`, `GLOBAL`, and `LOCK`. The vertical bar syntax allows implementors to specify the currently used syntax as the environment specification.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.16 *X11/95-31 `Kill` indirection*

From the proposal · “Unknown, But at least two vcndors allows this now.”

2020 · [ISM & GTM implemented `kill`; MV1 & FM1 have not. `New` not yet tested]

ISM	Partially implemented. [`kill`, test `new`]
GTM	Partially implemented. [`kill`, test `new`]
FM	Not implemented. [`kill`, test `new`]
MV1	Not implemented. [`kill`, test `new`]

4.2.17 *X11/95-91 `$order` definition*

From the proposal · Negligible.

4.2.18 *X11/95-94 Parameter passing clarification*

From the proposal · None. All known implementations conform to the new wording.

4.2.19 *X11/95-96 Spaces at end-of-line*

Many implementations already (intentionally or by accident) incorporate this change. The remaining implementations, those that conform to X11.1-1995, must be adapted.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.20 *X11/95-116 `^$Job` device information*

Implementors already support parsing of *ssvn*s and already track this device ownership information, so implementing this MDC extension is comparatively easy. No implementations currently implement it.

ISM	implemented?
GTM	implemented?
FM	implemented?

MV1 implemented?

4.2.21 *X11/95-117* Ssvn *collation*

For any implementor who has implemented `$order` and *ssvn*s, which is any Mumps 1995 compliant implementor, it is not difficult to add this extension. No implementations currently implement it.

ISM implemented?
GTM Not implemented (no *ssvn*s implemented)
FM Not implemented (*ssvn*s not yet implemented)
MV1 implemented?

4.2.22 *X11/95-118 Undefined* ssvns

None identified; altbough checking the resulting value of `$ecode` after referencing a *ssvn* node which was undefined would seem to be a good first step. Note that there are significant verification issues with *ssvn*s which may be undefined but which are expected to return a value (default values in `^$window`, for example). No implementations currently implement it.

ISM implemented?
GTM Not implemented (no *ssvn*s implemented)
FM Fully implemented
MV1 implemented?

4.2.23 *X11/95-119 Extended* extids

Few, if any, implementors have implemented *extsyntax*; one specifically ignores the case of the *extid* during evaluation of its version of *extsyntax*.

So none expected; there is the issue that existing parsers for *extid* will now need to be case insensitive; at least one implementor had already implemented their parser as case insensitive.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.24 *X11/95-132 Parameter passing to a routine*

As of 1995 DSM for Open VMS, MSM-PC/PLUS, and ISM for NT already were conforming to this extension's new formalization. Therc was a cost to Greystone to work out some accommodation with their customers that used a different interpretation. The status of current implementations needs to be investigated and documented.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.25 *X11/95-95 Portable* controlmnmonics *and* mnemonicspaces

Implementors would need to provide a mechanism for user-callable *mnemonicspace* routines. This could be a considerable overhead for those implementors who do not currently allow for Mumps code to be executed from within their systems.

However it would mean that implementors would not necessarily have to implement every *mnemonicspace* that the user community thought useful. This could be provided by the users or other third-party developers.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.26 *X11/96-44 Improve* mnemonicspace *handling*

This extension will have no effect on implementor practices or investments beyond minor implementation costs. The information this extension makes available is already available to the implementation; this just makes it available for inspection by Mumps developers and sysadmins.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.27 *X11/97-10* Mnemonicspec *cleanup*

This extension makes no change to the Mumps language, just clarifies its semantic description to match how all implementors already interpret the passage in question, so no implementation changes are required.

4.2.28 *X11/96-35 Parameter passing cleanup*

This extension makes no change to the Mumps language, just clarifies its semantic description to match how all implementors already interpret the passage in question, so no implementation changes are required.

4.2.29 *X11/SC12/98-13 User-definable I/O handling*

This extension requires some additions to Mumps implementations, however these are very small compared to the implementation of the portable controlmnemonics proposal.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.30 *X11/96-41* `String` *and* `M` *collation*

This extension makes no change to the Mumps language, just redefines its existing character set profiles, so no implementation changes are required.

4.2.31 *X11/96-42* `Charset ISO-8859-1-USA`

Costs of implementation. Note that this proposed *charset* uses the existing standard structures for using a non-ASCII character set, specifically **`^$character`**.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.32 *X11/96-45* `Charset` *names*

By itself, this extension adds no additional burden to implementors, because it defines the descriptors allowed for future or existing character set names, but does not open up any user-facing feature to create character set profiles with such names. For the moment, therefore, it is more of a redefinition of the *Standard* than a change to the language, so no implementation changes are required. This will be revisited after the user-defined *charset* technical proposal is approved.

4.2.33 *X11/96-43* `Ssvn` *formalization*

The indirection provisions of this extension should already be implemented by all Mumps implementors; this just brings the specification of *ssvn*s into compliance with the intentions and general interpretation of the original *ssvn* extension. Likewise, this makes explicit the intention that **`^$z`** *ssvn*s are set aside for implementor extensions but all other *ssvn* names are reserved for future expansion by the MDC; this is the usual and expected interpretation with names of Mumps language elements, but it needs to be double-checked for violations. Support for the use of environments with *ssvn*s is new, but may also have been interpreted as implied and so already implemented. This all needs to be investigated.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.34 *X11/98-5 Fix* `algoref`

Test to ensure the improper format (**`$&&`***name*) does not work and the new/intended one (**`$&`***name*) does.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.35 *X11/98-26 Canonic form of* ssvn *name*

At the time the extension was approved, all implementions returned the intended canonic form of `^$character`. This should be rechecked.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.36 *X11/98-8* Ssvn *for user/group identification*

There are no known implementation conflicts.

David Marcus of Micronetics commented in September 1996 that some operating system security mechanisms do not use User and Group identifiers. In such cases, it is presumed that the Mumps implementation would either make some form of identifier(s) available and call it (them) by these names for purposes of these *ssvn*s or leave either or both *ssvn*s undefined. In any case, it is presumed that implementors choosing to make these features available would make their definition at process creation time an option.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.37 *X11/98-29 Local variables in* `^$job`

No burden or other negative impact identified beyond the issues of implementation. Almost all implementations include the nonstandard unsubscripted $order, which provides the same features, so the codebase needed to support traversing local variable names has already been written. Note that references to the nodes of `^$job`/`"VAR"` could be considered syntactic sugar for references to the actual local variables (but not for instances where the variable is not referred to as part of an *expr*; such as in `new`, `kill`, or passing by reference).

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.38 *X11/98-19 User-defined* ssvns

This is potentially significant, however many implementations already provide similar facilities for their own implementation of certain *ssvn*s. Thus it is anticipated that this would not be a major burden on implementors. As of 1998, at least one implementor has implemented a very similar scheme.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.39 *X11/96-51 Device environment*

Vendors who permit network node names and directory names in device names or in *deviceparameters* are required to supply an alternate, standard method. The mechanism for global, routine, and lock *environment*s should apply to device *environment*s as well.

At the minimum, an implementor could call all device *environment*s non-existent and generate the specified error if they are used.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.40 *X11/97-31 Output time out*

Implementers must add a timer to the execution of **`write`** arguments and **`read`** arguments that produce output. When the timed interval expires before execution completes, an error condition results.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.41 *X11/SC12/98-11 Output time out initialized*

If incorporated when implementing *X11/97-31 Output time out*, this proposal should incur no additional cost. The sponsor expects the cost of adding it later to be small.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.42 *X11/98-23 `Open` command clarification (re-`open`)*

At the time this extension was approved by the MDC, all implementations complied.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.43 *X11/96-52 Routine management*

The functionality introduced is available in most systems in a slightly different way (e.g., **`zload`** & **`zsave`**), as is the analogous **`merge`** command. It should not be difficult for most implementors to introduce the **`rload`** and **`rsave`** commands.

ISM implemented?
GTM implemented?
FM implemented?

MV1 implemented?

4.2.44 *X11/SC15/98-8* `$mumps` *function*

It is expected that most implementations already have internal entry points that provide, at least, the Boolean functionality required by this proposal. Consequently, this should be implementable with minimal work.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.45 *X11/98-24 Duplicate keywords clarified*

This extension should have no impact on implementions, only making standard what is already current practice.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.46 *X11/SC12/98-14 Undefined* devicekeyword

This is a minor change to insert the standardized error code into a situation where an error is already generated. This proposal will have a larger impact for those implementations that ignore invalid *devicekeyword*s.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.47 *X11/98-25 Device parameter issues*

This extension was edited several times during its proposal phase until it minimized the impact on implementations. It is now trivial to implement.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.48 *X11/96-9 Pattern negation*

Mumps 2020 includes four extensions that update pattern match directly, plus several internationalization extensions that introduce new pattern codes, so all implementations will require updates to their pattern-match modules. In the mid-1990s, Micronetics characterized the work involved for this extension as a “moderate size change.”

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.49 *X11/97-3 Pattern ranges*

One Mumps implementor in the mid 1990s said “While this is fairly easy to do, I would… request: avoid piece-meal solutions.” Best done together with other pattern-match upgrades to save time and effort.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.50 *X11/98-27 Pattern match string extraction*

The disambiguating rules specified in the extension are the result of extensive debate and discussion and should correspond identically with the pattern-matching algorithms used by the implementors, to avoid unnecessarily adding to their effort or significantly impacting the efficiency of pattern-match evaluation. Of course, simply performing the extraction process will doubtless impose some significant impact on the implementors.

The author of this extension was unable to get the implementors to provide any useful feedback about the likely impact of this proposal, despite three years of outreach efforts at the MDC. In the end, just as the proposal was approved as an MDC Type A extension, one implementor indicated that the proposal could not be implemented cost effectively, while offering no further explanation. So the impact is and will remain unknown until a reference implementation implements it and reports back on the experience.

ISM implemented?

GTM implemented?

FM implemented?

MV1 implemented?

4.2.51 *X11/96-34 Modulo by zero*

This is a simple extension, simple to implement, an hour or two of programming at most; many implementations already implement this.

ISM Fully implemented.

GTM Not implemented.

FM Not implemented.

MV1 Fully implemented.

4.2.52 *X11/96-27 Xor operator*

This is a relatively simple extension, simple to test, two or three hours of programming. Run through short list of boundary cases for the new `!!` and `'!!` operators.

ISM Not implemented.

GTM	Not implemented.
FM	Not implemented.
MV1	Not implemented.

4.2.53 *X11/SC13/97-9 Mathematics errors*

For implementations that already return errors for these conditions, this is a relatively simple extension to implement, a few hours of programming per error. For implementations that instead return some value for these cases, it will be a bit more work. Implementors necessarily already special-case these conditions. so triggering an error shouldn’t be hard. One implementor reported “The level of effort is relatively minor (mostly tedious coding).”

ISM	Not implemented.
GTM	Not implemented.
FM	Not implemented.
MV1	Not implemented.

4.2.54 *X11/96-10 Reverse* `$query`

This is already a widely implemented extension.

ISM	Fully implemented
GTM	Not implemented; implemented on YottaDB.
FM	Not implemented; causes uninterruptible infinite loop.
MV1	Fully implemented.

4.2.55 *X11/SC13/98-15 Definition of reverse* `$query`

This involves no change to any implementations, since it just brings the *Mumps Standard* into synch with the way it is and should be implemented.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.56 *X11/97-22* `Set $qs[ubscript]` *pseudo function*

To date, no implementor has identified any undue impact by this proposal. Most implementations store *namevalue*s in a fashion that permits them to efficiently affect the naked indicator and “walk” through a structure with **`$query`**. This proposal is in effect a form of **`set $piece`** of a *namevalue* using a “magic” delimiter to **`set`** the *name* or subscript. As such, it’s not expected to be an onerous task to implement this proposal.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.57 *X11/96-68 Negative subscripts in* namevalue

None. All known implementations permit negative subscripts where namevalue is specified.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.58 *X11/SC13/98-13 Define variable* `m` *in* `$piece`

None: this proposal only corrects an omission in the text of the *Standard*.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.59 *X11/SC13/TG6/98-3* `$horolog` system function

The addition of a new function is not inherently expensive, depending mainly on what it does. Implementations will need to expose higher-resolution seconds, which all of them possess internally, and to expose timezone information and do some math around UTC, which most of them already do. Overall, then, this should be an inexpensive function to implement, especially since some already provide an implementation-specific version.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.60 *X11/SC13/TG3/98-4 Data record functions*

Of the various possible solutions explored over the decade-long history of this proposal, several implementors reported that this approach is preferable, because it does not depend on the left and right sides of the **set** command passing hidden variables for delimiters, etc. If implementations include the capability of user-defined intrinsic functions and setleft quasi-functions—and a new technical proposal might be introduced to support this—then it could cost implementations nothing to support these new functions.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.61 *X11/96-11* Fncode *correction*

This MDC extension only says "Costs of implementation." Some implementations may already reserve invalid fncodatoms and generate some implementation-specific error, while others may currently ignore them and need to be changed. The scope of the change is relatively small, the generation of an error that either reuses an existing generic reserved keyword error or introduces a new one specific to fncodatoms. The upcoming Syntax errors proposal will introduce a standard `S` ecode that must be generated, so implementors may wish to wait for that proposal to be approved before implementing this extension, to save effort.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.62 *X11/96-32 Sign of zero in* `$fnumber`

Vendors should verify their implementations and notify their customers of nonconformance or of changes made to achieve conformance. If changes are needed, they are a small matter of programming.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.63 *X11/96-67 Leading zero in* `$fnumber`

Vendors should verify their implementations and notify their customers of nonconformance or of changes made to achieve conformance. If changes are needed, they are a small matter of programming.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.64 *X11/96-57* `Goto` *rewording*

All known Mumps implementations will be unaffected by this proposal. Three implementors confirm that the change will not affect their implementations.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.65 *X11/96-58 Add* `job` *to routine execution*

All known Mumps implementations will be unaffected by this proposal. Three implementors confirm that the change will not affect their implementations.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.66 *X11/SC15/98-42 Subscript indirection and* `lock`

At the time the extension was approved, all Mumps implementations already implemented this extension, as they had before it was created.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.67 *X11/96-49* `Quit` *with argument in* `for` *command*

Implementations that do not produce an error when executing an argumented **`quit`** command in the scope of a **`for`** command would not be in conformance. All implementations known by the extension author do produce an error in this circumstance.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.68 *X11/98-31* `If then & else`

This extension says “None expected (beyond costs of implementation),” but it introduces two completely new circumstances for restoring a saved variable (in this case **`$test`**)—when leaving a *line* (at *eol* or **`goto`**) and when returning to a *line* from a deeper stack level—which will probably involve some novel coding to call the existing variable-restoration modules and may require changes to those modules. So this is likely to be trickier to implement than the extension author imagined.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.69 *X11/97-25 First line format*

No impact. It removes a convention that was never a requirement nor implemented, except for M-Global, which provided tools that follow the existing MDC Type A. They can continue to do so or not at their pleasure.

4.2.70 *X11/96-65 Normalize definition of* `tstart`

This is a refinement of the syntax specification, with no effect on implementations.

4.2.71 *X11/SC15/98-5 Error handling corrections*

This is a refinement of the syntax specification, with no effect on implementations, other than to make it easier for new implementors to understand the specification.

4.2.72 *X11/SC13/TG15/97-3 Local variable storage*

Minimally, this lets implementors stop worrying about the portability size limit, so long as they document their limits in their implementation's conformance clause. But preferably it requires implementations to provide virtual memory storage for symbol tables, so they are not bound by the restrictions of RAM. But as written, this extension makes it impossible to determine whether an implemtation is compliant with the Mumps Standard in this regard, because it does not provide the language elements needed to do so, while neutering the existing tools (portable symbol-table limit and `$storage`) that would have been used in the past.

4.2.73 *X11/98-14 Sockets binding*

This extension literally includes nothing at all under §4.2 Impact on Existing Vendor Practices and Investments, which under the TG19's new proposal rules would make it ineligible for Subcommittee Type A status, let alone MDC Type A. This is a relatively large, complex, and important extension. Most implementations already include some kind of sockets binding, since sockets are a basic feature of modern computing, but adapting that to this extension's standard syntax and semantics will requre a nontrivial amount of work.

ISM	Not implemented.
GTM	Partially implemented.
FM	implemented?
MV1	implemented?

4.2.74 *X11/98-28 Event processing*

One implementor representative indicated that much of the underlying effects of this proposal already exist in their products to handle the process control inherent in M. They indicated further that they expect that this is the case with all M implementors. In a previous version of this proposal, two implementors were represented in the vote. One voted affirmatively, the other abstained, and no cons were raised. In the version previous to this, three implementors were present, and all three voted in favor of the proposal. No cons concerning vendor impact were raised at that time. Implementors were invited to comment further upon this, but in the final vote, only one implementor was present and they voted against it citing implementation difficulties.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.75 *X11/SC15/98-11 Generic command indirection*

Implementing indirection is always trickier, and this new form is trickier than most, since it can and is intended to cross syntactic-unit boundaries. On the other hand, Mumps implementation code already handles a wide variety of forms of indirection, and this new form is otherwise not different in kind, so there will likely be some implementation savings through reuse of previous indirection-handling algorithms and subroutines.

The current version, which only throws a generic `S0` error if the results are not syntactically valid, which greatly simplifies the implementation challenge of deciding which error to throw. It is the intention of TG19 to produce a followup technical proposal that details the many syntax errors that should be used for different kinds of syntactic problems, but that does not apply to the current extension.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.76 *X11/SC15/TG2/98-2 Object usage*

The extension correctly notes that it will require substantial effort by implementors. However, most Mumps implementors agree that object orientation in some form is an essential extension to Mumps.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.77 *X11/94-23 Library proposal*

The impact on implementations is relatively minor. An interface to some calling table would need to be provided, which would allow either Mumps or non-Mumps code to be called from a function call.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.78 *X11/95-11 Library functions, general mathematics*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.79 *X11/95-12 Library functions—trigonometry*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.80 *X11/95-13 Library functions—hyperbolic trigonometry*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.81 *X11/96-26 Library functions—matrix mathematics*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.82 *X11/95-22 "Standard" in library element definitions*

This extension tunes up the definition of the term "standard" when used to describe library elements. It does not change the Mumps language, so no implementation changes are needed.

4.2.83 *X11/95-14 Library functions—complex mathematics*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM	implemented?
GTM	implemented?
FM	implemented?
MV1	implemented?

4.2.84 *X11/95-112 REPLACE library function*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.85 *X11/95-111 `PRODUCE` library function*

Because the library extension supplies correct sample code for implementing each function, the impact on implementors is small, once the basic library-proposal framework has been implemented. Some implementors already offered the proposed functionality back in 1995.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.86 *X11/SC13/TG5/96-5 Corrections to library functions*

No new implementation work is required by this document. It is editorial direction to fix the sample code in the previous library functions; those extensions will be easier to implement with these bugs fixed.

4.2.87 *X11/96-74 Operator overrides*

The extension says "None for implementors, except for library code installation."

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.88 *X11/98-21 Miscellaneous character functions*

The extension says "A small amount to allow for optional extra nodes in `^$character` for implementors, plus library code installation."

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.2.89 *X11/98-32 Cyclic redundancy code functions*

The extension says "None beyond implementation [sic] it. Note that many vendors already have `$Z`... CRC functions so all they will have to do is call it in a different way."

ISM implemented?
GTM implemented?
FM implemented?

MV1 implemented?

4.2.90 *X11/SC13/98-10* `FORMAT` *library function, revised yet again*

Minimal impact expected. The strategy behind all these library functions to ensure maximal implementation is to supply sample code for a basic implementation with each one, to make implementing them trivial. Implementers may choose to exert themselves to develop more efficient implementations.

ISM implemented?
GTM implemented?
FM implemented?
MV1 implemented?

4.3 Techniques & costs for compliance verification

The *Mumps Validation Test Suite (MVTS) 1995* needs to be updated to the *MVTS 2020.* These costs will be borne by MDC volunteers.

MDC extensions typically recommend compliance steps and offer examples of usage, described below. X11/TG19/WG5 will start by implementing them in the *MVTS 2020,* but will alter and extend the algorithms to create the *MVTS* tests. The results of that work will be documented in this section.

4.3.1 *X11/94-5 Initialising intrinsics*

From the proposal · On initiation of a process if `$io=""` then `$x` and `$y` should be `0` and `$device` and `$key` be `""`. `$Test` should always be `0`. `$io` should either be `""` or `$principal`.

Compliance verification for the initial setting of `$io` or `$principal` cannot be achieved or the values of `$x`, `$y`, `$device` and `$key` if `$io'=""`.

4.3.2 *X11/93-39* `$reference`

From the proposal · Create a file containing the following text:

```
set x=$data(^a(3)),^(3)="data ^a 3 (must be retrieved later)"
set v1=$reference
write !,"$reference should be equal to '^a(3)': ",v1
set x=$data(^(3,4))
set v2=$reference
write !,"$reference should be equal to '^a(3,4)': ",v2
lock ^p(27)
set v3=$reference
write !,"$reference should still be equal to '^a(3,4)': ",v3
set $reference=v1
write !,"Naked reference should work: ",^(3)
write !
quit
```

Also, consider the following piece of Mumps code:

```
        ...
        do savnak
        ...
        do restnak
        ...
        quit
        ;
savnak  set old=$reference quit
restnak set $reference=old quit
```

The code at label **savnak** saves the value of the gvn that was most recently referenced. Note that a reference to a name does not automatically mean that the associated variable exists or has a value. The code at label **restnak** restores the naked indicator to its previous variable. Note that setting **$reference** does not imply that the global variable that it points to is actually referenced, only that the naked indicator is (re)set.

```
 ...
 set ^x=1 write !,$reference
 set ^x(1)=2 write !,$reference
 set $reference="" write !, "Naked indicator is undefined. "
```

should produce:

```
^x
^x(1)
```

Note that a reference to **^x** makes the naked indicator undefined, but does not make **$reference** empty. The final command, **set $reference=""** makes **$reference** empty and also makes the naked indicator undefined.

4.3.3 *X11/SC12/93-33 Effect of* `close $io`

From the proposal · The following code can be used to verify compliance. This test will not work on implementations that do not allow **close**ing the principal device, or on implementations that cannot re-establish ownership of the principal device once it is **close**d.

```
clostest ;test close
         for  read !,"test device: ",device quit:device=""  do
         . write "  "
         . open device:1 if '$test write "can't open" quit
         . use device
         . close device
         . set test=$io=""
         . open $principal use $principal
         . write $select(test:"passed",1:"failed")
         . quit
         quit
```

4.3.4 *X11/94-4 Two character operators*

From the proposal · Annotated examples of use:

```
set a="abc"
set b="xyz"
if a]b write "true" ; should not write anything
if a]"abc" write "true" ; should not write anything
if a]="abc" write "true" ; should write true
```

The three statements:

```
if x>=1
if x'<1
if +x-1!(+x>1)
```

should all produce the same answer. Similar statements for the other operators can be used to verify their compliance.

4.3.5 *X11/94-14 Multiple patatoms within alternation*

From the proposal · Example 1

```
 if x?2N1(3P2A,2U3N).E
```

will now be possible, and is equivalent to

```
 if (x?2N3P2A.E)!(x?2N2U3N.E)
```

Example 2

```
 if x?2N1"-"1(3N1"-"1N,1N1":"4N)
```

can be used to validate data which must be in one of two forms: *nn-nnn-n* or *nn-n:nnnn*.

Example 3

The sample Mumps code below must result in output of **"11"**.

```
test    set x="24,,,ab"
        set y="24,,abc"
        write x?2N1(3P2A,2P3A)
        write y?2N1(3P2A,2P3A)
        quit
```

4.3.6 *X11/94-28 Portable string length*

From the proposal · Create a routine containing the following code:

```
strlen ;this tests whether the longer string length is implemented
       set x=""
       write !,"This should work with the current standard"
       for i=1:1:255 set x=x_$char(i#26+65)
       do show
       write !,"The next code-line is not portable with"
       write !,"the current standard, but would become"
       write !,"importable when this proposal is accepted."
       for i=256:1:510 set x=x_$char(i#26+65)
       do show
       write !,"The following remains non-portable:"
       set x=x_"more"
       do show
       quit
       ;
subs   write !,"Examples with subscripts:"
       write !,"This was already portable:"
       set s="" for i=1:1:100 set s=s_"x"
       set s(s)="Long subscript"
       write i,"This will also become portable:"
       set s="" for i=1:1:255 set s=s_"x"
       set s(s)="Very long subscript"
       write !,"This remains non-portable:"
       set s=s_"This is still portable"
       set s(s)="But this subscript may be too long."
       ;
show   write !,"x now has a length of ",$length(x),"characters."
       quit
```

4.3.7 *X11/94-47* New svn *addition:* `$test`

From the proposal · Create a subroutine which modifies `$test` (i.e. `if '$test`); compare the value of `$test` before and after calling this subroutine, as well as a copy of the subroutine with `new $test` placed as the first command in the subroutine. `$test` should not change in the second version.

This testing subroutine could be written as follows:

```
go      if 1 do test write !,"$test should equal 1, $test="_$test
        else  write !,"This should not print; $test="_$test
        quit
test    new $test ; save the existing value' of $test
        if 0 ; $test should now be equal to 0
        quit  ; this will restore the newed value of $test
```

4.3.8 *X11/98-30* New `$reference`

From the proposal · Annotated example of use:

```
go      if $data(^test(1234)) ;sets $reference
        do test1 ;test which uses new $reference
        write !,"$reference should equal ^test(1234), $reference="_$reference
        do test2 ;test which does not use new $reference
        write !,"$reference should not equal ^test(1234),
$reference="_$reference
        quit
test1   new $reference ;save the existing value of $reference
```

```
        quit
test2   ;entry point where new $reference is not performed
        if $data(^failure(4321)) ;should set $reference
        quit  ;for test1 this should restore the saved value of $reference
```

4.3.9 *X11/96-13 Portable length limit of* names

From the proposal · Set two local variables whose names differ only in the portability *Limit*th character to different values. **Write** them out and ensure the values are correct. Attempt to set a local variable whose name is implementor's *Limit*+1 characters long and ensure that an **"M56"** error occurs.

Set two global variables whose names differ only in the portability *Limit*th character to different values. **Write** them out and ensure the values are correct. Attempt to set a global variable whose name is implementor's *Limit*+1 characters long and ensure that an **"M56"** error occurs.

Create two different routines whose names differ only in the portability limitth character. Ensure that both were created. Attempt to create a routine whose name is implementor's *Limit*+1 characters long and ensure that an **"M56"** error occurs.

Create a routine with two subroutines with labels whose names differ only in the portability *Limit*th character. Have the code in the two subroutines write different values. **Do** the two labels and ensure the values are correct. Create a routine with a label which is implementor's *Limit*+1 characters long. Attempt to **do** or **goto** it and ensure that an **"M56"** error occurs.

2020 · [protect with error processing]

4.3.10 *X11/96-7 Lower-case characters in* names

From the proposal · Create a routine containing the following text:

```
label set X=1,x=2 write !,X,x
```

An implementation that conforms to the standard should be able to execute this routine, and should produce the text **"12"** when executed.

2020 · [need to test locals, globals, labels, and routines, not just locals; need error processing to protect execution of MVTS]

4.3.11 *X11/97-23 Portable length limit of strings*

From the proposal · **kill set x=$justify(" ",2**15-1)** and ensure no **"M75"** error occurs.

```
set x=$justify("",2**15-1)
for i=1:1:$l(x)\500 set ^foo(somenode,i)=$extract(x,500*(i-1)+1,500*i)
set:$length(x)#500 i=$length(x)\500+1,^foo(somenode,i)=$extract(x,500*(i-
   1)+1,$length(x))
```

2020 · [need to test max array reference, string length, results, lines, routine size, symbol table; need error processing to protect execution of MVTS; attempt to create entities of incrementally increasing lengths until provoke an error]

4.3.12 *X11/95-63 Naming string length error*

From the proposal · Attempt to create a string, the length of which would be implementor’s limit`+1` characters, and ensure that an `"M75"` error occurs.

4.3.13 *X11/94-46 `^$global` correction*

From the proposal · Annotated example of use: `^$global("temp")` should represent the global `^temp` in the default environment. `^$|"linus"|global("temp")` should represent the global `^temp` in the environment `"linus"` (provided the *ssvn* formalism is approved, and `^$global` is designated as an *ssvn* which can be used with *environment* syntax).

Create some globals and then check `^$global` to make sure they are presented in the correct format. Specifically:

```
set noglo=1,glo="" for  set glo=$o(^$global(glo)) quit:glo=""  do
. if $data(^$global(glo))#2 write !,"Global ^"_glo_" exists" set noglo=0
if 'noglo quit  ;there are globals -- can't continue
write !,"No Globals are defined, beginning test..."
write !,"^temp is
   "_$select($data(^$global("temp"))#2:"listed??",1:"unlisted")
set ^temp($job)=1 write !,"^temp has been set."
write !,"^temp is
   "_$select($data(^$global("temp"))#2:"listed",1:"unlisted??")
```

4.3.14 *X11/SC13/94-33 Kill data and kill subscripts of* glvns

From the proposal · Annotated examples of use:

Clear any value of `^abc(3)` but does not affect presence of subscripts:

```
kvalue ^abc(3)
```

Deletes any subscripts of `myvar` without affecting its value:

```
ksubscripts myvar
```

This is analogous to the exclustve `kill` command; only the value portion of all local variables other than `a`, `b`, and `c` would be cleared (`kill`ed):

```
kvalue (a,b,c)
```

only the subscripts of all local variables other than `a`, `b`, and `c` would be `kill`ed.

```
ksubscripts (a,b,c)
```

Mumps progam to check `$data` after executing `kvalue` and `ksubscripts`. The following routine is offered as an example of such:

```
testkill ;djm;
         new  d setup            ;start with known 1oca1s
         ks a,b,c                ;ksubscripts inclusive
         i $d(a)'=1 d error(1)   ;a should have remained unchanged
         i $d(b)'=0 d error(2)   ;b(1) should have been deleted
         i $d(c)'=1 d error(3)   ;Only c(1) should have been deleted
         d setup
         kv a,b,c                ;kvalue inclusive
         i $d(a)'=0 d error(4)   ;a should have been deleted
         i $d(e)'=10 d error(5)  ;b should have remained unchanged
         i $d(c)'=10 d error(6)  ;Only c should have been deleted
         d setup
         ks (a,b)                ;ksubscripts exclusive
         i $d(a)'=1 d error(7)   ;a should have remained unchanged
         i $d(b)'=10 d error(8)  ;b(1) should have remained unchanged
         i $d(c)'=1 d error(9)   ;Only c(1) should have been deleted
         d setup
         kv (a,b)                ;kvalue exclusive
         i $d(a)'=1 d error(10)  ;a should have remained unchanged
         i $d(b)'=0 d error(11)  ;b(1) should have remained unchanged
         i $d(c)'=10 d error(12) ;only c should have been deleted
         w !,"End. " q
error(n) w !,"error #",n q
setup    s a=1,b(1)=1,c=1,c(1)=1 ;variables with $d=1, 10, 11
         q
```

2020 · [add **`kvalue`** all and **`ksubscripts`** all]

4.3.15 *X11/95-2 Execution environment*

From the proposal · Annotated examples of use:

```
job |"my-world"|^rou(a,b)::10
job |"system"|label^rou
```

2020 · Surprisingly, this important MDC extension contains no discussion of techniques or costs for compliance verification, consistent with Art Smith's critique of a widespread pattern of inadequacy in MDC members diligence in considering these impacts.

4.3.16 *X11/95-31 `Kill` indirection*

From the proposal · This should write **`001`**:

```
s a="@b",b="x",x=1 k (@a) w $d(a),$d(b),$d(x)
```

2020 · [Similar tests should be written for the **`new`** command.]

4.3.17 *X11/95-91 `$order` definition*

From the proposal · Does not change compliance verification.

4.3.18 *X11/95-94 Parameter passing clarification*

From the proposal · None. The change clarifies the development of a conformance test.

4.3.19 *X11/95-96 Spaces at end-of-line*

A conforming implementation must pass all of the examples listed below. Similar tests can be specified for other argumentless commands.

ex1 *sp* *eol*

ex2 `label` *sp eol*
ex3 *sp sp eol*
ex4 `label` *sp sp eol*
ex5 *sp* `set` *sp* `X=14` *sp eol*
ex6 *sp* `quit` *sp sp eol*
ex7 *sp* `quit` *sp eol*
ex8 *sp* `.` *sp eol*

Examples **`ex1`** and **`ex2`** are requirements of the standard (i.e. a *line* must contain at least the *sp* that represents *ls*. The proposal does not change that.

Examples **`ex3`** and **`ex4`** are already allowed by the current standard (the *sp* is part of *ls*).

Examples **`ex5`** and **`ex6`** are not allowed by the curent standard, but will be allowed by the proposed change.

Example **`ex7`** is allowed by the current syntax definition of the **`quit`** command, but excluded by rule *c* in §8.1.1 (spaces in commands). The proposed change would make this valid syntax.

Example ex8 is allowed by the current syntax definition of *li* (§6.2.1), but excluded by rule *a* in §8.1.1 (spaces in commands). The proposed change would make this valid syntax.

Note that the (argumentless) **`quit`** at the end of the line may be followed by one or more spaces just like any other command. (the rule that it is followed by at least two spaces if it is not immediately followed by *eol* is changed).

4.3.20 *X11/95-116* `^$job` *device information*

Check if **`^$job($job,"$PRINCIPAL")`** is the same value as **`$principal`**.

Check if **`^$job($job,"$IO")`** is the same as **`$io`**, including while switching between devices.

Open a number of devices, and make sure that all the open devices are included under **`^$job($job,"OPEN",*)`**—and make sure they are not there when they are closed.

4.3.21 *X11/95-117* Ssvn *collation*

Testing this extension is involved. It can only be done after *X11/98-19 User defined* ssvn*s* is implemented. A new user-defined *ssvn* must be created that allows the first subscript to be set to any string value, so we can test out the collation of various strings as we change the collation algorithm.

1. Create a routine with three subroutines—**`ascii`**, **`digitm`**, **`digits`**—extrinsic functions, each of which is a one-argument extrinsic function that can be used as an algoref (see §7.1.4.1) to specify a collation algorithm. The collation algorithms should collate differently from each other and differently from the **`M`** character set profile's collation algorithm. They only need to operate upon the values **`1`**, **`2`**, **`11`**, **`12`**, **`"A"`**, **`"B"`**, **`"AA"`**, and **`"AB"`**. Set these values into the first subscript of the user-defined *ssvn*.

Where collation algorithm **`M`** puts these values in this order:

`1`, **`2`**, **`11`**, **`12`**, **`"A"`**, **`"AA"`**, **`"AB"`**, **`"B"`**

the three subroutines will put them in this order:

ascii: **1**, **11**, **12**, **2**, **"A"**, **"AA"**, **"AB"**, **"B"**
digitm: **1**, **2**, **11**, **12**, **"A"**, **"B"**, **"AA"**, **"AB"**
digits: **1**, **2**, **"A"**, **"B"**, **11**, **12**, **"AA"**, **"AB"**

2. Save and then clear the following *ssvn* nodes:

```
^$system($system,"COLLATE")
^$character(^$system($system,"CHARACTER"),"COLLATE"))
^$character(^$job($job,"CHARACTER"),"COLLATE"))
```

Under the definition of **$order** both before and after this extension, with these three *ssvn* nodes missing the **M** collation rules apply. Use a **for** loop with **$order** to traverse the susbcripts in the user-defined *ssvn* and confirm they are traversed in **M** collation order.

3. Set **^$character(^$job($job,"CHARACTER"),"COLLATE"))** to **ASCII** collation, and traverse the subscripts again. Under the Mumps 1995 rules, the nodes should appear in **ASCII** order, but under the new rules introduced by this extension they will still appear in **M** collation order.

4. Set **^$character(^$system($system,"CHARACTER"),"COLLATE"))** to **digitm** collation, and traverse the subscripts again. Under the Mumps 1995 rules, the nodes should still appear in **ASCII** order, but under the new rules introduced by this extension they will now appear in **digitm** collation order.

5. Set **^$system($system,"COLLATE")** to **digits** collation, and traverse the subscripts again. Under the Mumps 1995 rules that is not a legal *ssvn* node, so attempting to set it should cause an error; even if it did succeed, the nodes would still appear in **ASCII** order, but under the new rules introduced by this extension they will now appear in **digits** collation order.

4.3.22 *X11/95-118 Undefined* ssvns

None identified; altbough checking the resulting value of **$ecode** after referencing a *ssvn* node which was undefined would seem to be a good first step. Note that there are significant verification issues with *ssvn*s which may be undefined but which are expected to return a value (default values in **^$window**, for example).

1. Start by creating a user-defined *ssvn* with simple and well-defined characteristics and test its node(s) for defined versus undefined.

2. Then test all the *ssvn* nodes defined in Mumps 1995, if module 4 of the MVTS does not already do this.

3. Then test all the ssvn nodes defined in Mumps 2020.

4. For purposes of the core MVTS, MWAPI and other Mumps standards will not be tested, so **^$window**'s unusual semantics around undefined nodes does not directly apply. However, MVTS modules should eventually be created for them, independent o f the effort to create MVTS 2020.

4.3.23 *X11/95-119 Extended* extids

Compliance testing would need to be developed for specific *extid*s. For existing implementations, try different case versions of *extid*s for which compliance tests have already been developed. Try empty *exttext*, and try *exttext* that includes an empty *linebody*.

4.3.24 *X11/95-132 Parameter passing to a routine*

Build two routines `t1` and `t2`:

```
t1    ;first line not a formal line
en(x) ;
      q

t2(x) ;first line is a formal line
      q
```

`d ^t1(1)` should cause an error.
`d ^t2(1)` should not cause an error.

4.3.25 *X11/95-95 Portable* controlmnmonics *and* mnemonicspaces

This MDC extension omited any discussion of techniques for compliance verification. Two routines should be developed for a user-defined mnemonicspace, the first to hold a variety of test controlmnemonics, the second for controlmnemonics `open`, `use`, and `close` (and perhaps also `?open`, `?use`, and `?close`). A validation routine should be written with tests to use that menmonicspace and exercise the various controlmnemonics, confirming that each generated the specified output, concluding by ceasing to use this mnemonicspace.

This should be validated in concert with *X11/96-44 Improve* mnemonicspace *handling*, which adds the ability to determine the current mnemonicspace in use via `^$device` and to return controlmnemonic status information in `$key`.

4.3.26 *X11/96-44 Improve* mnemonicspace *handling*

Open a device with a list of *mnemonicspace*s—make sure they appear in `^$device(*,"MNEMONICSPEC")` as expected. Use each *mnemonicspace* with the `use` command and make sure the appropriate value is returned in the `^$device(*,"MNEMONICSPACE")` node.

Use this capability to create tests of Mumps 1995's basic *mnemonicspace* capabilities, to confirm that `open` and `use` and `close` are properly activating and deactivating *mnemonicspace*s.

Create a user-defined mnemonicspace that includes controlmnemonics that return status information in `$key`. Use them, and confirm the changes to `$key`.

4.3.27 *X11/97-10* Mnemonicspec *cleanup*

This extension makes no change to the Mumps language, just clarifies its semantic description, so no validation is required.

4.3.28 *X11/96-35 Parameter passing cleanup*

This extension makes no change to the Mumps language, just clarifies its semantic description, so no validation is required.

4.3.29 *X11/SC12/98-13 User-definable I/O handling*

This extension brushes off the validation costs and ignores validation techniques, so the MVTS team will have to develop that from scratch. Generally, the approach will be to define a portable *mnemonicspace* that includes all seven new labels (**`%READ`**, **`%READS`**, **`%WRITE`**, **`%WRITES`**, **`%WRITENL`**, **`%WRITETAB`**, and **`%WRITEFF`**), then use a variety of **`read`**s and **`write`**s and check to ensure the results are as expected. stacking of **`$test`** and **`$reference`** during such **`read`**s and **`write`**s must be tested.

4.3.30 *X11/96-41 `String` and `M` collation*

This extension makes no change to the Mumps language, just redefines its existing character set profiles, so no validation is required.

4.3.31 *X11/96-42 `Charset ISO-8859-1-USA`*

Test the aspects of the *charset*: make sure all the characters match the appropriate patcodes. Confirm the single-character collation sequence is as expected; then generate two- and three-character sequences to confirm the ligature and diacritical character collation sequences as expected. Also test the M collation version using canonic and non-canonic numbers and how they sort compared to non-numeric strings.

4.3.32 *X11/96-45 `Charset` names*

Test *charset* names that include the set of characters defined in *descsep*. The problem at present is that Mumps provides no standard way to do this, so a technical proposal will be developed to allow user-defined *charset*s. That capability will then permit the testing of *charset* names.

4.3.33 *X11/96-43 `Ssvn` formalization*

This extension brushes off the validation costs and ignores validation techniques, so the MVTS team will have to develop that from scratch. All the name indirection and subscript indirection tests in the MVTS for locals and globals should be copied over and adapted for *ssvn*s.

Likewise, tests need to be developed for the use of environments with *ssvn*s (and elsewhere), but this is not possible until basic environment operations are standardized enough for the MVTS to use them to switch, create, and delete environments. [Revisit this after the new environment operations technical proposal is developed and approved.]

Generally, *ssvn*-related tests are lacking in the MVTS, so this whole area needs significant testing work.

4.3.34 *X11/98-5 Fix* `algoref`

Test to ensure the improper format (**`$&&`***`name`*) does not work and the new/intended one (**`$&`***`name`*) does.

4.3.35 *X11/98-26 Canonic form of* ssvn *name*

Possible results of **`$query(^$c("M"))`** could be:

```
"^$c(""M"",xxx)"
"^$character(""M"",xxx)"
"^$C(""M"",xxx)"
"^$CHARACTER(""M"",xxx)"
```

where *xxx* is the first subscript that happens to be defined in **`^$character`** at the time. The effect of implementing this proposal would be that only the last form (**`"^$CHARACTER(""M"",`***`xxx`***`)"`**) would be acceptable.

4.3.36 *X11/98-8* Ssvn *for user/group identification*

Compliance verification would involve using **`$data`** to verify existence/non-existence of implementation pre-defined values in these items. If extant the read-only status could be verified by referencing the values and subsequently attempting to change the values. If not implementation pre-defined, a value could be stored and then treated in a similar fashion.

[This cannot be validated in a standard, portable way because so much of how these nodes are assigned is left to the implementor. Standard ways of defining whether the nodes are written by the application should be proposed, so the language can be self-validating with portable code.]

[This could be handled by a related concern, that the write-once quality should not be intrinsic to these nodes but should be defined somewhere, so the property can be queried. The capability should be generalized to all access control settings, r = read, w = write, c = create, d = delete. write-once would then be reconceptualized as cr access, to allow creating the value but not writing to it thereafter. This should be definable within ssvns to document access restrictions on all ssvn and svn values, and should be settable for any local or global as well. And whether the access control can be adjusted should be defined and definable as well (s = secure). Error conditions should be defined for the remaining access-control violations (and delete access should be split out of **`M96`**). Fileman can be used as a model but should be generalized. Compare to access-control schemas for major security systems to ensure a modern approach.]

4.3.37 *X11/98-29 Local variables in* `^$job`

This is another MDC extension that ignored compliance verification. Use **`for`**, **`kill`**, **`new`**, **`merge`**, **`read`**, and **`set`** to manipulate the symbol table into known states and then **`$order`** and **`$query`** through **`^$job`** to confirm that the correct list of local variables is present, with none missing and no extras; be sure to check the case of an empty symbol table.

4.3.38 *X11/98-19 User-defined* ssvns

This is another MDC extension that brushes off compliance verification. It notes that association of a *routine* with an *ssvn* is an implementation-specific issue, but does not identify that as a problem with this extension; it is, because it makes any MVTS compliance code non-portable. [A followup technical proposal should be developed to standardize the association of user-defined *ssvn*s with *routine*s.]

The tests should define a user-defined *ssvn* multiple times with all the essential combinations of references and labels present and absent to test compliance. The error conditions should be provoked and *ecode* tested. Case sensitivity should be tested.

[This proposal offers fewer protections for **`$test`**, **`$reference`**, and the like than the *Portable* controlmnemonics extension offered. A followup technical proposal should be developed to remedy this, to make user-defined *ssvn*s safer to use.]

4.3.39 *X11/96-51 Device environment*

The value and interpretation of device environment is left to the implementor. If an implementor claims to support them, extend the existing compliance tests of **`$io`**, **`open`**, **`use`**, and **`close`** as shown below:

```
set remdev="remote_device.txt",remenv="RNV"
open |remenv|remdev set curdev=$ioreference
use |remenv|remdev write "Text" set newdev=$ioreference
use @curdev close |remenv|remdev write newdev
```

Add a test for **`$ioreference`**. **`$ioreference`** should return the empty string like **`$io`** if no current device has been identified (if proposal *X11/SC12/92-26* has been accepted it is the empty string). If a current device has been identified it always returns a *devn* where the value of *env* may be the empty-string if the implementation does not support any environments. In that case the default environment is also the empty string.

Add similar tests for **`$pioreference`**.

Ensure tests support implementations that opt to have no *environment*s at all, only the current *environment*, designated by the empty string.

Verify that non-existent device environment evokes the required error code.

[These MVTS tests can be made portable only if two followup technical proposals are developed and passed: the first to introduce an **`^$environment`** *ssvn*, so the existing *environment*s can be identified programmatically; the second to introduce user-defined *environment*s, so they can be created, switched to and from, and deleted programmatically. These would create the basic controls needed to query, set up, use, and clean up *environment*s for testing purposes.]

4.3.40 *X11/97-31 Output time out*

Verification tests must be extended to provide a blocked condition on a device and to test the behavior of **`read`** and **`write`** commands in that condition. This is expected to be a minor extension. Coming up with a reliable, portable way for the MVTS to create a blocked condition on a device is the main challenge.

4.3.41 *X11/SC12/98-11 Output time out initialized*

The compliance tests for output time out need to change. Where they now check that the device parameter is preserved between processes, they must check that **close** resets **OUTTIMEOUT** and **OUTSTALLED** in **^$device** as well as the **OUTTIMEOUT** *deviceparam*.

4.3.42 *X11/98-23 `Open` command clarification (re-`open`)*

This is another MDC extension that brushes off compliance verification, saying only "to be provided." Part of the reason is that Standard Mumps lacks the language features required to verify whether or not any particular I/O parameter was successfully established or discarded by any Mumps I/O command. Therefore, in part, verification of this extension depends upon a successor technical proposal to address this (see ***^$job*** *device parameter information*).

In addition, the MVTS will need to establish and make use of a user-defined *mnemonicspace* that supports *deviceparam*s whose establishment and discarding can be detected by the MVTS.

Finally, the existing Mumps 1995 language about **open** parameters was so painfully terse that it never introduced them and almost implied the **open** command cannot have any effect on device parameters, so a followup technical proposal needs to add introductory language here, to make clear the very topic this extension is further clarifying.

4.3.43 *X11/96-52 Routine management*

This is yet another MDC extension that brushes off compliance verification, which is entirely blank. The basic features of **rload** and **rsave** can be tested by creating sample routines in locals and globals and saving them, by loading them, editing them, and resaving them, and so on. This will require extensive testing, to check all the details of the permitted syntax and semantics, including provoking the errors.

The *routineparameters* cannot be tested until the *Standard* routineparameters proposal is finished and approved as an MDC Type A extension.

We may need to create a *Portable job and routine parameters* technical proposal to introduce the ability to define suites of parameters and associated code to execute, to fully test the syntax and semantics of routine parameters.

4.3.44 *X11/SC15/98-8 `$mumps` function*

This is one of the most difficult possible MDC extensions to test properly, harder to test than all the other Mumps 2020 extensions put together. The extension's guidance for how to do so is vaguely suggestive of the scope of the problem and offers a few examples but is otherwise useless. The extension suggested just passing all the *line*s of all the *routine*s on any old Mumps system through **$mumps**, but this is far too haphazard.

First, one must identify a healthy range that all permutations of 2020 Standard Mumps *routine line*s pass. A catalog of Mumps *routine*s containing the variety of strictly Standard Mumps *line*s to be tested should be created and run through the function. The most convenient source of such *line*s comes from the MVTS itself.

At present the MVTS is structured so that Mumps code samples to be tested and the algorithm code that performs the testing are mixed together. A top-to-bottom overhaul of the MVTS that isolates all the code samples onto *line*s of their own would make it possible to write a traversal over all such legal *line*s, passing them into **`$mumps`** and ensuring the result for each is **`0`**.

Second, one must establish that a healthy range of permutations of Mumps *routine line*s illegal according to both the *2020 Standard* and to the implementation in question all fail when passed to **`$mumps`**. This requires cataloguing what each implementation supports, to be sure the *line*s in question are truly illegal, something the MVTS has never done before. A new technical proposal should be developed to create new a **`^$mumps`** *ssvn* that catalog the supported syntax of Standard Mumps and each implementation's extensions.

The MVTS illegal sample code *line*s can then be tagged with how illegal they are and for which implementations. The collection of sample illegal *routine line*s can be extended to cover the range properly. All of these illegal *line*s should be tagged with which syntax errors they are designed to provoke, even though as this extension makes clear the implementation has a choice of which errors to return. New MVTS code can then be written to traverse all the *line*s fully illegal for the current implementation, passing them into the function and checking their return errors.

Third, one must establish a healthy range of permutations of Mumps *routine line*s illegal according to the *2020 Standard* but legal according to the current version of each implementation. The catalogues of sample *line*s described above needs to be extended to include useful *line*s in that range for the implementation to be tested. The MVTS can then traverse all such *line*s, passing them into the function and checking the results.

The SSVN syntax catalog and the MVTS sample lines catalog must be updated with new subtrees describing each new *Mumps Standard* and each new version release of Mumps implementations, and the MVTS algorithms must be written to use the current version of the implementation to be tested. This will require a new **`$version`** *svn* technical proposal with corresponding ssvns in **`^$mumps`**.

4.3.45 *X11/98-24 Duplicate keywords clarified*

This is another MDC extension that brushes off compliance verification, saying only "None." The MVTS needs to set up uses of the **`open`**, **`use`**, **`close`**, **`rload`**, **`rsave`**, and **`tstart`** commands with duplicate keywords designed to establish the left-to-right evaluation. In the case of **`open`**, **`use`**, and **`close`**, this can be done with user-defined *mnemonicspace*s and duplicate user-defined *keyword*s and *attribute*s. In the case of **`tstart`**, it can be done with duplicate **`TRANSACTIONID`**s. For **`rload`** and **`rsave`**, the absence of anything like a routine *mnemonicspace*—let alone a user-defined one—blocks our ability to reliably test duplicate routine parameters, so a *Routine* mnemonicspace*s* technical proposal should be developed to supply the features needed to test this properly.

4.3.46 *X11/SC12/98-14 Undefined* devicekeyword

This is another MDC extension that brushes off compliance verification, saying only "To be supplied." The MVTS needs to use a user-defined *mnemonicspace*—to control which keywords and attributes do and don't exist—so it can refer to undefined ones and ensure that provokes the correct error.

4.3.47 *X11/98-25 Device parameter issues*

This is, yet again, another MDC extension that brushes off compliance verification, again saying only "To be supplied." The MVTS again needs to use a user-defined *mnemonicspace*, this time to try mixing the different formats of parameters to demonstrate they are resolved correctly.

4.3.48 *X11/96-9 Pattern negation*

Create a series of patterns involving negation both of *patcode*s and *strlit*s, and write subroutines to check the patterns without using pattern negation. Then create a series of strings, some of which match the patterns and some of which don't. Confirm that the subroutine and pattern negation provide the same results.

4.3.49 *X11/97-3 Pattern ranges*

Write a program which uses pattern ranges containing both one and two *strconst*s. Test various strings against the patterns and confirm that the language implementation returns the correct answer. Use the `$char` variant in various contexts, cases, and abbreviations to ensure it is working. If alternate character sets are available, try this test in an environment where the sorts after operator returns results differing from U.S. ASCII and confirm that the pattern match works according to the definition for the character set in use.

The extension makes clear that the intended collation used to define set ranges should be the sorts after operation, but this was left out of the formalization, which uses only such vague language as "between" and "trails." A followup technical proposal should explicitly define trails in terms of sorts after to remove this ambiguity. It should also clarify the result of a pattern set range like `["":"a"]` (probably a set including all characters from `$char(0)` through `"a"`) and define whether or not the empty string itself would count as a match (probably not). Also, *strconst* should perhaps be upgraded to full *gwrite* format.

4.3.50 *X11/98-27 Pattern match string extraction*

This is a complex extension, difficult to test thoroughly. The examples shown in §3.1 and §3.2 of the extension must produce the effects explained in the text. That basic set of tests should then be expanded to cover all the necessary boundary cases, as well as confirming the behavior across different character sets.

4.3.51 *X11/96-34 Modulo by zero*

This is a simple extension, simple to test, a few minutes of programming. Set an error trap, calculate modulo by zero, see whether the proper error condition occurs.

4.3.52 *X11/96-27 Xor operator*

This is a relatively simple extension, simple to test, an hour or two of programming. Run through short list of boundary cases for the new `!!` and `'!!` operators.

4.3.53 *X11/SC13/97-9 Mathematics errors*

This is a relatively simple extension to test, a few hours of programming. Include the following tests, and add additional ones as needed to cover all boundary cases:

```
set n=1 for  set n=n*2 ; will eventually get error M92 (numeric overflow)
set n=0**-1 ; will get error M9 (divide by zero)
set n=0**0 ; will get error M94 (zero to the power of zero)
set n=-3**.5 ; will get error M95 (complex result with nonzero imaginary
   part)
```

4.3.54 *X11/96-10 Reverse `$query`*

This is another MDC extension that brushes off compliance verification, saying only “None,” which could not be more false. This is a new feature, and the MVTS needs to be extended to valdiate whether it is implemented properly. This is maybe a day of programming to test all the boundary conditions properly.

4.3.55 *X11/SC13/98-15 Definition of reverse `$query`*

The error in the previous extension and the fix in this one together create a new boundary case for the MVTS to test, situations in which the previous extension would have returned the incorrect value of `""` instead of the proper value of the array-node name. The extension offers this example (although others should be added as well):

```
kill x set x=1,x(2)=2,x(2,3)=23 set valid=$query(x(2,3),-1)="x(2)"
```

4.3.56 *X11/97-22 `Set $qs[ubscript]` pseudo function*

A basic technique for verifying that **`set $qsubscript`** works properly is to set a variable to a *namevalue*, perform one or more **`set $qsubscript`**s on the variable, and compare the result with what you expect. For example:

```
set (X,Y)=$name(^FOO(1,2,3))
for I=5:-1:1 do
. set $qsubscript(X,I)=I+10
if X'=$name(^FOO(11,12,13,14,15)) do
. write "Ugg 1",!
. set $qsubscript(Y,0)=$name(^BARF("A"),0)
if Y'=$name(^BARF(1,2,3)) do
. write "Ugg 2",!
```

Additionally compliance testing should ensure that an **`M90`** error occurs if either the original or resulting string does not meet the form of a *namevalue*. It should also ensure that no error occurs if the *namevalue* “names” a non-existent *environment* or contains subscript values which do not meet the requirements of Section II Clause 2.3.3 (Values of subscripts).

4.3.57 *X11/96-68 Negative subscripts in* namevalue

The language elements involved in the various place in the *Mumps Standard* changed by this extension need to be checked to prove they allow negative numbers, not just positive ones. The MVTS already probably performs these tests, but it any omissions need to be remedied.

4.3.58 *X11/SC13/98-13 Define variable* `m` *in* `$piece`

Add tests to the MVTS for four-argument **`$piece`** that rule out the old, incorrect definition. We expect all implementations to pass it, but it has to be checked explicitly. Probably only a few hours of programming, most of which involves working out exactly what the erroneous behavior would have been had any implementations literally implemented the *Standard*.

4.3.59 *X11/SC13/TG6/98-3* `$horolog` system function

Compare the values returned by different arguments and see that the formats are correct. Then do similar testing as is done with the **`$horolog`** *svn*.

4.3.60 *X11/SC13/TG3/98-4 Data record functions*

This is another MDC extension that brushes off compliance verification, saying only “None mentioned here.” However, it does provide five well annotated examples of use that can serve as a starting point. The MVTS should adapt these examples and then write new tests to fill in the various boundary cases. It should take no more than a day or two of programming to cover the feature set.

4.3.61 *X11/96-11* Fncode *correction*

This MDC extension says “Not applicable,” but §2.2 Existing practice notes the implementations at the time produced a variety of incompatible responses to using an invalid *fncodatom*, and §7 Issues, pros and cons, and discussion reinforces that, so clearly this does need to be tested by the MVTS to ensure compliance.

Based on the need for MVTS support for this extension that the Syntax errors proposal needs to be completed and should include standard **`S`** errors for each kind of reserved language element, to help pinpoint attempted use of an illegal *command*, intrinsic *function*, *svn*, *ssvn*, *fncodatom*, *patatom*, and the like. That would give the MVTS the ability to try to use reserved *fncodatom*s and confirm the standard syntax error for reserved *fncodatom*s was produced.

4.3.62 *X11/96-32 Sign of zero in* `$fnumber`

This interpretation agrees with the conformance test MVTS V.8.2 from MUMPS Systems Laboratory as quoted in the NIST question (see 2.1) and its accompanying letter to the

MDC. The MVTS tests should be reviewed for opportunities to strengthen these checks, but otherwise no further work is needed.

4.3.63 *X11/96-67 Leading zero in* `$fnumber`

This interpretation agrees with the conformance test MVTS V.8.2 from MUMPS Systems Laboratory as quoted in the NIST question (see §2.1) and its accompanying letter to the MDC. The MVTS tests should be reviewed for opportunities to strengthen these checks, but otherwise no further work is needed.

4.3.64 *X11/96-57* `Goto` *rewording*

Unchanged by this extension. Review MVTS tests for **`goto`** command to ensure they permit or restrict targeting lines correctly.

4.3.65 *X11/96-58 Add* `job` *to routine execution*

Unchanged by this extension. Review MVTS tests for **`job`** command to ensure they the three wording changes are boundary checked in case some implementation decided to implement these accidental omissions.

4.3.66 *X11/SC15/98-42 Subscript indirection and* `lock`

The following example illustrates the intended behavior, supported by all implementations but not specified in the Standard until this extension:

```
>lock +@"^DIC(19)"@(42,0)
>write $data(^$length("^DIC(19,42,0)"))>0 => 1
```

In this example, the programmer uses subscript indirection to lock the *nref* **`^DIC(19,42,0)`**. A **`$data`** check of **`^$lock`** confrms that the **`lock`** command has not only executed without generating an error, it has also been properly interpreted and has established the lock we expect.

The MVTS should build tests around this approach, including checking boundary cases.

4.3.67 *X11/96-49* `Quit` *with argument in* `for` *command*

This should only be about an hour of programming to add the necessary tests to the MVTS (or confirm they are already there)—that an argumented **`quit`** within the scope of a **`for`** command should produce an error with *ecode*=**`"M16"`**. Tests of boundary cases, including nested **`for`** commands and **`quit ""`**, should be added.

4.3.68 *X11/98-31* `If then & else`

MVTS tests can be developed from this example, which should produce a line containing **`"true"`** followed by a line containing **`"false"`** (but only one such line):

```
for a=1,0 do
. if a then  write !,"true" if 0 ;reset $test after write
. else write !,"false"
```

Other MVTS tests should encompass timed **`job`**, **`lock`**, **`open`**, **`read`**, & **`write`** commands.

4.3.69 *X11/97-25 First line format*

NIST never added first-line-format tests to the MVTS because Annex E is only informative, not normative, so removing the annex has no effect on the MVTS.

4.3.70 *X11/96-65 Normalize definition of* `tstart`

This is a refinement of the syntax specification, with no effect on the MVTS tests.

4.3.71 *X11/SC15/98-5 Error handling corrections*

This is a refinement of the syntax specification, with no effect on the MVTS tests.

Except that as yet there are no MVTS tests for error handling. It is the largest omission in the suite that should cover Mumps 1995. It was caused by VA prematurely withdrawing its support for the FIPS Mumps Standard, as part of the VA bureaucratic coup that eventually put an end to the decentralized and user-driven Vista software lifecycle.

So these conditions identified by extension author David Marcus should be used as boundary cases to include in that test suite, when it is written.

4.3.72 *X11/SC13/TG15/97-3 Local variable storage*

This extension takes language features that previously were difficult to test—portable symbol-table limits and **`$storage`**—and makes them impossible to test. It needs to be augmented by two-followup technical proposals, one to make it possible to set size limits for symbol tables, globals, and global environments and to measure their size and growth rate, the other to selectively be able to define certain local arrays as bound not to the symbol-table size limits but those of a global or global environment.

4.3.73 *X11/98-14 Sockets binding*

Despite nine years and significant debate and development that went into the production of this extension, it only proposes validation by “Developing applications which would rely on correct implementation of this proposal.” This is of little specific use to the MVTS. This is a big extension with many detailed features, so developing a complete test suite for it will be time consuming. All the features and all their boundary cases must be tested. To ensure portability, this will probably be best done with a loopback socket, so that multiple MVTS jobs can open and use and write to and read from and close sockets. The example in §3.2 of the extension make a reasonable starting point.

4.3.74 *X11/98-28 Event processing*

Despite seven years and significant debate and development that went into the production of this extension, under validation costs and techniques it says only “None.” This is a big extension with many detailed features, so developing a complete test suite for it will be extremely time consuming, roughly the same scale as transaction processing and error processing, though possibly even harder. All the features and all their boundary cases must be tested. The two examples in §3.2 of the extension make a reasonable starting point, but much, much more most be developed, and some - such as the POWER event - will require manual intervention by the MVTS operator to trigger.

4.3.75 *X11/SC15/98-11 Generic command indirection*

This extension is a small step over a deep chasm. Its §3.2 includes annotated examples of use. These can be used to begin building the extensive suite of tests of boundary cases needed to fully test generic command indirection.

4.3.76 *X11/SC15/TG2/98-2 Object usage*

This will probably be the hardest of the existing MDC Type A extensions to test. The extension itself includes several detailed examples based on the then extant Micronetics Standard Mumps implementation (unfortunately, long since acquired, deprecated, and finally eliminaged by Intersystems), so the MVTS authors will need to start from scratch. This extension probably cannot be tested until the Object definition proposal is completed and integrated, so it can be used to create objects whose use can then be tested.

4.3.77 *X11/94-23 Library proposal*

This extension cannot be tested until after the subsequent library extensions are integrated, so they may be used to test this proposal within the limits of the libraries in question. A complete test of this extension will require the development of a User-defined library extension, to give the MVTS the ability to programmatically create and test libraries and elements that test the bounds of this proposal, to ensure all boundary cases are tested properly.

4.3.78 *X11/95-11 Library functions, general mathematics*

For each function, a test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

4.3.79 *X11/95-12 Library functions—trigonometry*

For each function, a test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

4.3.80 *X11/95-13 Library functions—hyperbolic trigonometry*

For each function, a test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

4.3.81 *X11/96-26 Library functions—matrix mathematics*

For each function, a test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance). Extension §3.2 Annotated examples of use includes a few extended examples that might be adapted to get started.

4.3.82 *X11/95-22 "Standard" in library element definitions*

This extension tunes up the definition of the term "standard" when used to describe library elements. It does not change the Mumps language, so no MVTS tests are needed.

4.3.83 *X11/95-14 Library functions—complex mathematics*

For each function, a test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

4.3.84 *X11/95-112 `REPLACE` library function*

A test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

§4.3 Techniques and costs for compliance verification includes a sample validation test, and *X11/95-111 PRODUCE library function* includes more examples that may be adapted.

4.3.85 *X11/95-111 `PRODUCE` library function*

A test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance).

§4.3 Techniques and costs for compliance verification includes a sample validation test, and the rest of the extension includes more examples that may be adapted.

4.3.86 *X11/SC13/TG5/96-5 Corrections to library functions*

No new tests are required by this document. It is editorial direction to fix the sample code in the previous library functions, so the tests for those extensions will cover this document as well.

4.3.87 *X11/96-74 Operator overrides*

The extension says only "No significant costs are anticipated," abandoning the question of techniques for testing compliance. A test suite will have to be developed to check that the function-value is within the limits derived from mathematically correct value and specified error-range (tolerance). But more strategically, this extension was introduced to help support the definition of character set profiles and their collation algorithms, so in addition to testing the two library functions in isolation, the MVTS needs to test their use within the larger target frameworks. Followup technical proposals to clean up the definition of *ssvn*s, including `^$character`, and to introduce user-defined character set profiles may be needed to test these functions within their intended context.

4.3.88 *X11/98-21 Miscellaneous character functions*

The extension says only "No significant costs are anticipated," abandoning the question of techniques for testing compliance. The MVTS will need to test these functions and *ssvn*s on various strings from various character set profiles to test the boundaries properly. The user-defined *ssvn* and user-defined *charset* extensions will prove helpful here.

4.3.89 *X11/98-32 Cyclic redundancy code functions*

The extension says only "Generate CRC values and compare to expected results," but which values it leaves as an exercise to the MVTS authors.

4.3.90 *X11/SC13/98-10 `FORMAT` library function, revised yet again*

The extension includes extensive examples that can be harvested to help develop the MVTS test suite for this extremely flexible library funciton.

4.4 Legal considerations

None.

5 Relationships

5.1 Dependencies

X11/TG19/WG2/2019-1 Technical proposal format 2020, nontechnical proposal.
X11.1-1995 M, MDC specification.

As MDC extensions are integrated into the *Draft Mumps Standard* and listed in §5.3 Superseded MDC Documents, this document becomes dependent on them.

5.2 Dependents

This is the MDC's most important document, with the greatest number of dependents.

Every new technical proposal is dependent on this one, because this document defines the MDC specification they propose to modify.

5.3 Superseded MDC documents

1. *X11/94-5 Initialising intrinsics*
2. *X11/93-39 `$reference`*
3. *X11/SC12/93-33 Effect of `close $io`*
4. *X11/94-4 Two character operators*
5. *X11/94-14 Multiple* patatom*s within* alternation
6. *X11/94-28 Portable string length*
7. *X11/94-47 `New`* svn *addition: `$test`*
8. *X11/98-30 `New $reference`*
9. *X11/96-13 Portable length limit of* name*s*
10. *X11/96-7 Lower-case characters in* name*s*
11. *X11/97-23 Portable length limit of strings*
12. *X11/95-63 Naming string length error*

13. *X11/94-46* `^$global` *correction*
14. *X11/SC13/94-33 Kill data and kill subscripts of* glvn*s*
15. *X11/95-2 Execution environment*
16. *X11/95-31* `Kill` *indirection*
17. *X11/95-91* `$order` *definition*
18. *X11/95-94 Parameter passing clarification*
19. *X11/95-96 Spaces at end-of-line*
20. *X11/95-116* `^$job` *device information*
21. *X11/95-117* `Ssvn` *collation*
22. *X11/95-118 Undefined* ssvn*s*
23. *X11/95-119 Extended* extid*s*
24. *X11/95-132 Parameter passing to a routine*
25. *X11/95-95 Portable* controlmnemonic*s and* mnemonicspace*s*
26. *X11/96-44 Improve* mnemonicspace *handling*
27. *X11/97-10* Mnemonicspec *cleanup*
28. *X11/96-35 Parameter passing cleanup*
29. *X11/SC12/98-13 User-definable I/O handling*
30. *X11/96-41* **`String`** *and* **`M`** *Collation*
31. *X11/96-42* Charset *ISO-8859-1-USA*
32. *X11/96-45* Charset *names*
33. *X11/96-43* Ssvn *formalization*
34. *X11/98-5 Fix* algoref
35. *X11/98-26 Canonic form of* ssvn *name*
36. *X11/98-8* Ssvn *for user/group identification*
37. *X11/98-29 Local variables in* `^$job`
38. *X11/98-19 User-defined* ssvn*s*
39. *X11/96-51 Device environment*
40. *X11/97-31 Output time out*
41. *X11/SC12/98-11 Output time out initialized*
42. *X11/98-23* `Open` *command clarification (re-*`open`*)*
43. *X11/96-52 Routine management*
44. *X11/SC15/98-8* `$mumps` *function*
45. *X11/98-24 Duplicate keywords clarified*
46. *X11/SC12/98-14 Undefined* devicekeyword
47. *X11/98-25 Device parameter issues*
48. *X11/96-9 Pattern negation*
49. *X11/97-3 Pattern ranges*
50. *X11/98-27 Pattern match string extraction*
51. *X11/96-34 Modulo by zero*
52. *X11/96-27 Xor operator*
53. *X11/SC13/97-9 Mathematics errors*
54. *X11/96-10 Reverse* `$query`

55. *X11/SC13/98-15 Definition of reverse* `$query`
56. *X11/97-22* `Set $qs[ubscript]` *pseudo function*
57. *X11/96-68 Negative subscripts in* namevalue
58. *X11/SC13/98-13 Define variable* `m` *in* `$piece`
59. *X11/SC13/TG6/98-3* `$horolog` *system function*
60. *X11/SC13/TG3/98-4 Data record functions*
61. *X11/96-11* Fncode *correction*
62. *X11/96-32 Sign of zero in* `$fnumber`
63. *X11/96-67 Leading zero in* `$fnumber`
64. *X11/96-57* `Goto` *rewording*
65. *X11/96-58 Add* `job` *to routine execution*
66. *X11/Sc15/98-42 Subscript indirection and* `Lock`
67. *X11/96-49* `Quit` *with argument in* `for` *command*
68. *X11/98-31* `If then` *&* `else`
69. *X11/97-25 First line format*
70. *X11/96-65 Normalize definition of* `tstart`
71. *X11/SC15/98-5 Error handling corrections*
72. *X11/SC13/TG15/97-2 Local variable storage*
73. *X11/98-14 Sockets binding*
74. *X11/98-28 Event processing*
75. *X11/SC15/98-11 Generic indirection*
76. *X11/SC15/TG2/98-2 Object usage*
77. *X11/94-23 Library proposal*
78. *X11/95-11 Library functions, general mathematics*
79. *X11/95-12 Library functions—trigonometry*
80. *X11/95-13 Library functions—hyperbolic trigonometry*
81. *X11/96-26 Library functions—matrix mathematics*
82. *X11/95-22 “Standard” in library element definitions*
83. *X11/95-14 Library functions—complex mathematics*
84. *X11/95-112* `REPLACE` *library function*
85. *X11/95-111* `PRODUCE` *library function*
86. *X11/SC13/TG5/96-5 Corrections to library functions*
87. *X11/96-74 Operator overrides*
88. *X11/98-21 Miscellaneous character functions*
89. *X11/98-32 Cyclic redundancy code functions*
90. *X11/SC13/98-10 FORMAT library function, revised yet again*

5.4 Related MDC documents

[As current MDC Type A extensions are added to §5.3 above, remove them from this list.]

X11.1-1995 M Language Standard

X11.6-1995 M Windowing API

X11/89-5 `$principal` (MDC Type A)

X11/90-51 Logical OR capability in pattern match operator (MDC Type A)
X11/91-5 (demoted & rescinded)
X11/91-??? Event management (Alfred Garcia, 1991-10-22)
X11/92-8 Structured system variables (MDC Type A)
X11/92-48 Structured system variables (MDC Type A)
X11/92-96 NIST request for clarification
X11/93-6 NIST problem statements 35–42
X11/93-9 Responses to NIST issues 1–34
X11/93-45 Responses to NIST issues 35–42
X11/94-50 Charles Sorenson letter to MDC
X11/95-52 MDC reply to Charles Sorenson
X11/95-60 NIST issues 1–45 revisited
X11/95-72 OLE/2 objects in M
X11/99-7 Notes about draft ANSI/MDC X11.1-2001 Standard
X11/TG3/92-4 Mumps directions
X11/TG8/93-15 Library proposal format
X11/SC1/88-20 Natural language handling
X11/SC1/88-46 `$HoroLog` *function*
X11/SC1/89-44 `$HoroLog` *function ver 2*
X11/SC1/89-54 SSVN
X11/SC1/90-1 `HoroLog`/SSVN
X11/SC1/90-22 Clock
X11/SC1/90-63 Clock version 2
(——— early 1990s document ID unknown) Otherwise/Alternative (failed after discussion)
X11/SC1/91-4 Proposal for event processing in Mumps
X11/SC1/91-81 Error processing
X11/SC1/91-82 Summary of differences between X11/SC1/91-43 & X11/SC1/91-81
X11/SC1/TG19/91-3 Synchronous event processing
X11/SC1/TG19/91-3A Asynchronous event processing
X11/SC12/92-26 Value of `$io`
X11/SC12/93-20 Character set profiles (MDC Type A)
X11/SC12/93-21 ASCII character usage
X11/SC12/93-23 Alternate collation sequences
X11/SC12/93-32 Null device
X11/SC13/92-17 `Set` *incremental*
X11/SC13/92-19 `Set` *positional*
X11/SC13/94-16 ssvn *for local variables*
X11/SC13/TG2/WG1/93-1 Regular expressions (discussion document)
X11/SC13/TG2/WG1/94-1 A comparison of regular expressions and pattern match in Mumps (discussion document)
X11/SC13/TG2/98-5 CRC-12 cyclic redundancy code function
X11/SC13/TG2/99-1 Corrections to the format library function

X11/SC13/TG13/94-2 `$order` definition and mathematical style (discussion document)
X11/SC13/TG13/95-5 Unsubscripted `$order`
X11/SC14/TG6/95-7 TCP-MUMPS mnemonic binding
X11/SC15/96-12 M object extensions
X11/SC15/91-13 First line format
X11/SC15/94-23 Process specific globals v2
X11/SC15/96-5 Library functions, editorial corrections
X11/SC15/TG1/91-1 Error processing
X11/SC15/TG2/92-6 OO framework & architecture
X11/SC15/TG2/92-7 Object environment, basic architecture
X11/SC15/TG2/92-8 The “class” object
X11/SC15/TG2/92-9 Communications with objects
X11/SC15/TG2/92-10 Methods & messages and object behavior
X11/SC15/TG2/92-11 Inheritance engines
X11/SC15/TG2/92-12 Regions of the object environment
X11/SC15/TG2/WG1/93-1 A first try at an M OOP specification
X11/SC15/TG2/WG1/93-2 A second try at an M OOP spec
X11/SC15/TG2/WG1/93-3 Sample M OOP code
X11/SC15/TG2/WG1/93-4 Document register
X11/SC15/TG2/WG1/93-7 Declaring & distinguishing variables
X11/SC15/TG2/WG1/93-8 The class 'class'
X11/SC15/TG2/WG1/93-9 Accessing the public members of an object
X11/SC15/TG2/WG1/93-10 Possible elements for a variable definition
X11/SC15/TG2/WG1/93-21 M object-oriented programming open/pending issues
X11/SC15/TG2/WG1/95-20 Object definition
X11/SC15/TG4/92-3 Synchronous event processing
X11/SC15/TG4/WG1/92-4 Unified event processing proposal
X11/SC15/TG4/WG1/92-6 Notes on event processing for Mumps
X11/SC15/TG4/WG1/92-7 Interprocess communication using event queue
X11/SC15/TG9/94-8 Standard routineparameters
X11/SC15/TG9/95-1 Parameter passing to a routine
X11/SC15/TG9/95-5 Naming syntax errors
X11/SC15/TG16/98-2 Indirect everything (superseded, but contains valuable material)
X11/SC15/TG17/97-2 `Then` command (rescinded)
X11/SC16/97-2 Taking M beyond the year 2000

5.5 Related standards activities & liaisons

ANSI X3.51-1986 Representation of universal time, local time differential, and United Stares time zone references for infomation interchange

ANSI X3J

Common Object Request Broker Architecture (*CORBA,* a specification for cross-system object interactions).

CSC-STD-002-85 Department of Defense Password Management Guideline (the "Green Book")
Department of Defence, *RFC # 793, Transmission Control Protoco; DARPA Internet program Protocol Specification; September 1981*
ISBN 0-201-03809-0 *The Art of Computer Programming, Fundamental Algorithms* (Knuth)
ISBN 0-486-61272-4 *Handbook of mathematical functions* (Abramowitz, Stegun)
ISBN 0-06-461019-5 *Dictionary of Mathematics* (Borowski, Borwein)
ISBN 7204-2033-4 *Statistical and Computational Methods in Data Analysts* (Brandt)
ISO/SC22/WG20 Internationalization
ISO-6937 Diacritical mark ordering
ISO 7498-2-1988(E) Security Architecture
ISO 8601 Data elements and interchange formats—Information interchange—Representation of dates and times
ISO 8601:1988/Cor.1:1911 (E) Data elements and interchange formats—Information interchange—Representation of dates and times. Technical Corrigendum 1
ISO 8859-1:1987 8 Bit single-byte coded graphic character sets - part 1: Latin alphabet No. 1
ISO/IEC 9899 C programming language
ISO/IEC 10646 Universal Coded Character Set (UCS)
ISO/IEC DIS 11430 - Generic package of elementary functions for ADA
ISO/SC11/WG20 N221 European multilingual ordering
A Painless Guide to CRC Error Detection Algorithms, Williams, Ross N.
UTS #10 Unicode Collation Algorithm (UCA)
Working Implementation Agreements for Open Systems Interconnection Protocols: Part 12- OS Security; Output from the December 1991 OSI Implementors' Workshop
MDC/TG13 Backwards incompatibility (dissolved 1993-10)
MDC/TG17 Interpretations
MDC/TG18 *ssvn* coordination
MDC/TG19 Reconstitution
  MDC/TG19/WG1 Transcription
  MDC/TG19/WG2 Governance
  MDC/TG19/WG3 Language Syntax
  MDC/TG19/WG4 Implementation
  MDC/TG19/WG5 Validation
  MDC/TG19/WG6 Shell
MDC/SC11 User interface (merged into SC12 1995-10)
  MDC/SC11/TG4 Windowing API (moved to SC12/TG12 1995-10)
MDC/SC12 Evironment
  MDC/SC12/TG2 Internationalization (dissolved 1998-06)
  MDC/SC12/TG7 Security
  MDC/SC12/TG9 General device issues
  MDC/SC12/TG14 Networking (dissolved 1998-06)
MDC/SC13 Data management & manipulation
  MDC/SC13/TG2 String handling

MDC/SC13/TG3 Record manipulation

MDC/SC13/TG5 Mathematics (dissolved 1997-03)

Recommendation: explore whether to set a flag on overflow instead of generate an error. Look at documents from other standards bodies about this and other mathematics errors.

MDC/SC13/TG6 Date and time

MDC/SC13/TG9 Data structures (dissolved 1995-06)

MDC/SC13/TG10 **$reference** & naked indicator (dissolved 1996-09)

MDC/SC13/TG11 **Kill** data & descendants (dissolved 1995-01)

MDC/SC13/TG13 Mumps data structure traversal

MDC/SC13/TG15 Local variable storage (dissolved 1998-06)

MDC/SC14 Networking & communications (merged into SC12/TG14 1996-09)

MDC/SC14/TG2 Network directions (dissolved 1995-06)

MDC/SC14/TG3 Network syntax (merged into SC12/TG14 1996-09)

MDC/SC15 Programming structures

MDC/SC15/TG2 Object oriented programming

MDC/SC15/TG4 Event processing

MDC/SC15/TG7 SQL (dissolved 1995-06)

MDC/SC15/TG9 Routine management & manipulation

MDC/SC15/TG10 Routine management (dissolved 1994-06, merged into TG9)

MDC/SC15/TG11 Portability size issues (dissolved 1997-09)

MDC/SC15/TG13 *ssvn* syntax (dissolved 1997-03)

MDC/SC15/TG16 Indirection

MDC/SC15/TG17 Process Control

MDC/SC16 Object oriented language

5.6 Document history

[As each extension is integrated into this *Mumps Draft Standard,* it is assigned a new document ID (2020-1, 2020-2, etc.), and a new subentry is added in this section with its own draft # (v1, v2, etc.), describing how it was changed when it was integrated.]

2020-04-30 · *X11/TG6/WG1/2020-90 Mumps Draft Standard 2020* [v90]

<current document> TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/98-10 FORMAT library function, revised yet again,* with two changes:

1 in §7.1.7.6.4 **$%FORMAT^STRING** repoint references to *graphic & expr*; and

2 in §7.1.7.6 **STRING** library and Annex I Mumps standard library sample code, renumber existing library elements.

The definition and sample code for this library function depend upon ssvns that this extension does not define, which existed in earlier versions of this proposal. This needs to be fixed with a followup technical proposal, if *X11/SC13/TG2/99-1 Corrections to the FORMAT library function* does not already fix it.

2020-04-30 · *X11/TG6/WG1/2020-89 Mumps Draft Standard 2020* [v89]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-32 Cyclic redundancy code functions,* with one change:

1 in §7.1.7.6 **STRING** library and Annex I Mumps standard library sample code, renumber existing library elements.

2020-04-30 · *X11/TG6/WG1/2020-88 Mumps Draft Standard 2020* [v88]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-21 Miscellaneous character functions,* with one change:

1 in §7.1.7.6 **STRING** library and Annex I Mumps standard library sample code, renumber existing library elements.

2020-04-30 · *X11/TG6/WG1/2020-87 Mumps Draft Standard 2020* [v87]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-74 Operator overrides,* with three changes:

1 in §7.1.7.1.1 add CHARACTER to list of mandatory libraries;

2 insert §7.1.7.4 CHARACTER library elements, renumber subsequent libraries, and repoint references to them; and

3 in Annex I Mumps standard library sample code insert §1 CHARACTER library sample code, renumber subsequent libraries, and repoint references to them.

2020-04-29 · *X11/TG6/WG1/2020-86 Mumps Draft Standard 2020* [v86]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/TG5/96-5 Corrections to library functions,* with no changes.

2020-04-29 · *X11/TG6/WG1/2020-85 Mumps Draft Standard 2020* [v85]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-111 PRODUCE library function,* with no changes.

2020-04-29 · *X11/TG6/WG1/2020-84 Mumps Draft Standard 2020* [v84]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-112 REPLACE library function,* with three changes:

1 in §7.1.7.1.1 add STRING to list of mandatory libraries;

2 add §7.1.7.6 STRING library elements; and

3 in Annex I Mumps standard library sample code insert §2 STRING library sample code.

2020-04-28 · *X11/TG6/WG1/2020-83 Mumps Draft Standard 2020* [v83]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-14 Library functions—complex mathematics,* with one change:

1 in §7.1.7.5 **MATH** library and Annex I Mumps standard library sample code, renumber existing library elements.

2020-04-28 · *X11/TG6/WG1/2020-82 Mumps Draft Standard 2020* [v82]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-22 "Standard" in library element definitions,* with no changes.

Also passim finished standardizing capitalization and added missing references to annexes B and C to early entries in §4.1.

2020-04-27 · *X11/TG6/WG1/2020-81 Mumps Draft Standard 2020* [v81]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-26 Library functions—matrix mathematics,* with three changes:

1 in §7.1.7.2 Library element definitions add MATRIX to list of data types & point to new §7.1.7.2.1 Matrix data values;
2 insert the extension's formalization in §3.3.1 General concepts, notation, and terminology as new §7.1.7.2.1; and
3 in §7.1.7.5 **`MATH`** library and Annex I Mumps standard library sample code, renumber existing library elements.

Passim: apply initial standard mathematics typography.

2020-04-27 · *X11/TG6/WG1/2020-80 Mumps Draft Standard 2020* [v80]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-13 Library functions—hyperbolic trigonometry,* with one change:

1 in §7.1.7.5 **`MATH`** library and Annex I Mumps standard library sample code, renumber existing library elements.

2020-04-26 · *X11/TG6/WG1/2020-79 Mumps Draft Standard 2020* [v79]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-12 Library functions—trigonometry,* with one change:

1 in §7.1.7.5 **`MATH`** library and Annex I Mumps standard library sample code, renumber existing library elements.

2020-04-25 · *X11/TG6/WG1/2020-78 Mumps Draft Standard 2020* [v78]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-11 Library functions, general mathematics,* with one change:

1 put main function listings under §7.1.7.5 **`MATH`** library and sample code in Annex I Mumps standard library sample code.

Move §11 Library to §7.1.7 Library and §10 Object usage to §8.1.9 Object usage, and renumber User-defined mnemonicspaces as §8.1.10.

2020-04-24 · *X11/TG6/WG1/2020-77 Mumps Draft Standard 2020* [v77]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-23 Library proposal,* with three changes:

1 in after inserting §7.1.4.5.4 Characteristic: libraries, renumber subsequent **`^$job`** *ssvn*s and repoint references to them.

2 in §7.1.4.6 **`^$library`** insert a line introducing its top-level syntax and abbreviation **`^$li`**, following the pattern established for all other *ssvn*s; and

3 after inserting §7.1.4.6, renumber subsequent ssvns and repoint references to them.

2020-04-22/23 · *X11/TG6/WG1/2020-76 Mumps Draft Standard 2020* [v76]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC15/TG2/98-2 Object usage,* with five changes:

1 in §6.3.2 reword inserted text to incorporate wording changes from *X11/SC15/98-5 Error handling corrections*;

2 after inserting new §7.1.1 Values, renumber remaining clauses in §7.1 & repoint references to them;

3 after inserting §7.1.6.26 **`$type`**, renumber remaining functions & repoint references to them;

4 in §7.2.2.3 String relations the new text is inserted after the paragraph about **`=`** to keep the semantic text in the same order as the list of values for *relation*; and

5 in §8.2 Command definitions after inserting §8.2.2 **`Assign`**, renumber remaining functions & repoint references to them.

2020-04-22 · *X11/TG6/WG1/2020-75 Mumps Draft Standard 2020* [v75]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC15/98-11 Generic indirection,* with two changes:

1 extension proposes inserting the new clause as §6.3.1, but this dates back to the Indirect everything version; this approved extension is more of a Generic command indirection, so it is inserted as §8.1.4;

2 subsequent clauses in §8.1 are renumbered and references to them repointed.

2020-04-19 · *X11/TG6/WG1/2020-74 Mumps Draft Standard 2020* [v74]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-28 Event processing,* with eight changes:

1 in §6.3.3 Event processing replace "It is to be understood that use" with "Use", to bring text in line with wording about portability throughout the *Standard*;

2 in §6.3.3 replace "The requirement that **`Z`** be used permits the unused names to be reserved for future enhancement of the standard without altering the execution of existing routines which observe the rules of the standard" with "Event class names other than those starting with the letter **`"Z"`** are reserved for future enhancement of the standard", to bring text in line with wording about portability throughout the *Standard*;

3 in §7.1.4 Structured system variable *ssvn* add "**`e[vent]`**" to definition of *ssvname* instead of "syntax of **`^$event`** structured system variable";

4 in §7.1.4 renumber the subsequent *ssvn*s & repoint references to them;

5 in §7.1.4.5 **`^$job`**, in keeping with the earlier extension that changed how **`^$job`** is

specified, collect the new **^$job** nodes in §7.1.4.5.3 Characteristic: events, renumber the subsequent **^$job** node subsections, & repoint references to them;

6 in §7.1.4.5, following the pattern used throughout the *Standard* for values of variable nodes, replace "the value of the node is zero" with "*intlit*=**0**" and replace "the value is greater than zero" with "*intlit*>**0**".

7 in §8 Commands add **ab[lock]**, **asta[rt]**, **asto[p]**, **aunb[lock]**, **esta[rt]**, **esto[p]**, and **et[rigger]** to definition of *commandword*; and

8 in §8.2 Command definitions renumber the subsequent *commands* & repoint references to them.

2020-04-18 · *X11/TG6/WG1/2020-73 Mumps Draft Standard 2020* [v73]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-14 Sockets binding,* with four changes:

1 in Annex H Socket binding add the subtitle "(Informative)" to match the pattern of the other annexes;

2 in all headings involving partial metalanguage definitions, abbreviate the heading and move the partial definitions to follow the heading, to match the pattern used throughout the *Mumps Standard*;

3 passim replace the idiosyncratic "M[UMPS] device" with "device"; and

4 passim fix punctuation errors.

2020-04-18 · *X11/TG6/WG1/2020-72 Mumps Draft Standard 2020* [v72]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/TG15/97-3 Local variable storage,* with no changes.

2020-04-17 · *X11/TG6/WG1/2020-71 Mumps Draft Standard 2020* [v71]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC15/98-5 Error handling corrections,* with one change:

1 in §6.3.2 Error processing replace "indirection's" with "indirection", replace "text of the first list" with "the first *linebody*", replace "second line" with "second *linebody*", and add "as follows"; and

2 in §7.1.6.23 replace " then return" with ", returns" to match previous case.

2020-04-17 · *X11/TG6/WG1/2020-70 Mumps Draft Standard 2020* [v70]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-65 Normalize definition of* ***TSTART****,* with no changes.

2020-04-17 · *X11/TG6/WG1/2020-69 Mumps Draft Standard 2020* [v69]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-25 First line format,* with one change:

1 reletter remaining annexes & repoint references to them.

Repoint references to commands after §8.2.30 **Then**.

2020-04-16 · *X11/TG6/WG1/2020-68 Mumps Draft Standard 2020* [v68]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-31 `If` `then` & `else`*, with three changes:

1. in §8.2.30 **`Then`** replace "as per '**`new`** *svn*'" with "equivalent to **`new`** *svn*" to follow the pattern used elsewhere for such equivalencies (and to avoid potentially confusing extraneous single quotes);
2. in §8.2.30 "&" replaced with "and" to follow the pattern used throughout the *Standard,* which avoids confusion with the **`&`** operator; and
3. renumber following commands.

Note: There is an error in the semantic language about the effect of a **`quit`** command encountered at the current execution level, because it does not allow for any **`quit`** within the scope of a **`for`** command that follows the **`then`** command. A followup technical proposal should refine this language.

2020-04-16 · *X11/TG6/WG1/2020-67 Mumps Draft Standard 2020* [v67]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-49 `Quit` with argument in `for` command,* with no changes.

2020-04-16 · *X11/TG6/WG1/2020-66 Mumps Draft Standard 2020* [v66]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC15/98-42 Subscript indirection and `LOCK`,* with no changes.

2020-04-16 · *X11/TG6/WG1/2020-65 Mumps Draft Standard 2020* [v65]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-58 Add `job` to routine execution,* with no changes.

2020-04-16 · *X11/TG6/WG1/2020-64 Mumps Draft Standard 2020* [v64]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-57 `Goto` rewording,* with no changes.

2020-04-15 · *X11/TG6/WG1/2020-63 Mumps Draft Standard 2020* [v63]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-67 Leading zero in `$fnumber`,* with no changes.

2020-04-15 · *X11/TG6/WG1/2020-62 Mumps Draft Standard 2020* [v62]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-32 Sign of zero in `$fnumber`,* with no changes.

2020-04-15 · *X11/TG6/WG1/2020-61 Mumps Draft Standard 2020* [v61]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-11* fncode *correction,* with no changes.

2020-04-14 · *X11/TG6/WG1/2020-60 Mumps Draft Standard 2020* [v60]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/TG3/98-4 Data record functions,* with two changes:

1 in §4 Definitions update new term (from "Packed global" to "Data-record") to match final draft of proposal, & renumber remaining terms.
2 in §7.1.6 Intrinsic function *function* renumber remaining functions; and
3 in §8.2.28 step *c* replace "five operations" with "seven operations".

In §4 took this opportunity to standardize capitalization of first letter of each definition; sentences begin with a capital letter, non-sentences with lowercase, unless first word is a proper noun.

2020-04-14 · *X11/TG6/WG1/2020-59 Mumps Draft Standard 2020* [v59]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/TG6/98-3 `$horolog` system function,* with four changes:

1 in §4 Definitions renumber remaining definitions;
2 §7.1.6.8 **`$horolog`** replace "used by **`$h`**" with "used by the **`$horolog`** intrinsic variable" to follow pattern of spelling out language elements in semantic text and to avoid implying a recursive definition;
3 in §7.1.6.8 adjust periods, spaces, and capitalization to better match format of other functions; and
4 in §7.1.6 Intrinsic function *function* renumber remaining functions.

2020-04-14 · *X11/TG6/WG1/2020-58 Mumps Draft Standard 2020* [v58]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/98-13 Define variable* **`m`** *in `$piece`,* with no changes.

2020-04-14 · *X11/TG6/WG1/2020-57 Mumps Draft Standard 2020* [v57]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-68 Negative subscripts in* namevalue*,* with no changes.

Consistently italicize the term *Standard* when referring to *X11.1.*

2020-04-13 · *X11/TG6/WG1/2020-56 Mumps Draft Standard 2020* [v56]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-22 `Set $qs[ubscript]` pseudo function,* with one change:

1 in §8.2.28 **`Set`** replace substeps *1–5* with *a–e* to match the pattern followed by the other examples in this section.

2020-04-13 · *X11/TG6/WG1/2020-55 Mumps Draft Standard 2020* [v55]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/98-15 Definition of reverse `$query`,* with no changes.

2020-04-13 · *X11/TG6/WG1/2020-54 Mumps Draft Standard 2020* [v54]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-10 Reverse `$query`,* with no changes.

The character styles of mathematical subscripts throughout the MDS were updated to make them consistent (use of roman for most, italic only for subscripts that are variables).

2020-04-11 · *X11/TG6/WG1/2020-53 Mumps Draft Standard 2020* [v53]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/97-9 Mathematics errors,* with no changes.

2020-04-11 · *X11/TG6/WG1/2020-52 Mumps Draft Standard 2020* [v52]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-27 Xor operator,* with one change:

1 in §7.2.2.4 Logical operators *logicalop* replace “dual operators **`'&`** and **`'!`** are” with “dual operators **`'&`**, **`'!`**, and **`'!!`** are”.

2020-04-11 · *X11/TG6/WG1/2020-51 Mumps Draft Standard 2020* [v51]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-34 Modulo by zero,* with no changes.

2020-04-10 · *X11/TG6/WG1/2020-50 Mumps Draft Standard 2020* [v50]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-27 Pattern match string extraction,* with no changes.

2020-04-10 · *X11/TG6/WG1/2020-49 Mumps Draft Standard 2020* [v49]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-3 Pattern ranges,* with no changes.

2020-04-10 · *X11/TG6/WG1/2020-48 Mumps Draft Standard 2020* [v48]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-9 Pattern negation,* with no changes.

2020-04-09 · *X11/TG6/WG1/2020-47 Mumps Draft Standard 2020* [v47]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-25 Device parameter issues,* with no changes.

2020-04-09 · *X11/TG6/WG1/2020-46 Mumps Draft Standard 2020* [v46]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC12/98-14 Undefined* devicekeyword, with two changes:

1 in §8.2.7 **`Close`** change the location where the text is inserted, because of the previous reorg of this section;

2 instead of inserting two nearly identical paragraphs, different only by a single word,

insert a single paragraph with the phrase "contains a *deviceattribute* or *devicekeyword*"; and

3 replace the text "cause an error to happen. If an error occurs, the *ecode* will be **M109**" with "cause an error condition with *ecode*=**"M109"**" to match the pattern used throughout the *Standard* for describing error conditions.

2020-04-09 · *X11/TG6/WG1/2020-45 Mumps Draft Standard 2020* [v45]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-24 Duplicate keywords clarified,* with one change:

1 in §8.2.27 **Rsave** the new text is inserted as a new paragraph immediately before the paragraph starting "Assume that *glvn* …".

2020-04-09 · *X11/TG6/WG1/2020-44 Mumps Draft Standard 2020* [v44]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC15/98-8* ***$mumps*** *function,* with five changes:

1 in §7.1.6 Intrinsic function *function*, add **m[umps]** to list of functions in definition of *function*;

2 replace "the function returns" with "**$mumps** returns" and replace "the number **0**" with "the value 0" to match the patterns used throughout §7.1.6 [note: a new technical proposal will overhaul how *function* return values and *svn* and *ssvn* values are defined throughout the *Mumps Standard*];

3 after inserting 7.1.6.10 **$Mumps**, renumber following subsections of §7.1.6 & update all references to them;

4 add new error code **S0** to Annex B Error code translations; and

5 add new metalanguage elements to Annex C Metalanguage element dictionary.

2020-04-08 · *X11/TG6/WG1/2020-43 Mumps Draft Standard 2020* [v43]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-52 Routine management,* with four changes:

1 in §8.2.24 **Quit** add the text to the end of step *g* instead of step *f*;

2 renumber following subsections of §8.2 Command definitions & update all references to them;

3 in §8.1 add **rl[oad]** and **rs[ave]** to definition of *commandword* metalanguage element; and

4 in §8.2.27 **Rsave** replace every reference to "a/the **rsave** command" with "**rsave**" to match the pattern used throughout §8.2.

Also, in support of *X11/SC13/94-33 Kill data and subscripts*, in §8.1 add **kv[alue]** and **ks[ubscripts]** to definition of *commandword* metalanguage element, an essential modification that extension omited.

2020-04-08 · *X11/TG6/WG1/2020-42 Mumps Draft Standard 2020* [v42]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-23 `Open` command clarification (re-`open`),* with one change:

1 text in the paragraphs before and after the newly inserted one has been reorganized to improve flow.

2020-04-01 · *X11/TG6/WG1/2020-41 Mumps Draft Standard 2020* [v41]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC12/98-11 Output time out initialized,* with no changes.

2020-04-01 · *X11/TG6/WG1/2020-40 Mumps Draft Standard 2020* [v40]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-31 Output time out,* with no changes.

2020-04-01 · *X11/TG6/WG1/2020-39 Mumps Draft Standard 2020* [v39]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-51 Device environment,* with four changes:

1 default device environment text in §7.1.4.4.5 instead of §7.1.4.9, reference to “the following four *ssvn*s” changed to “five”, and the two lists of entities reordered to match those five *ssvn*s;

2 in §7.1.5.10 **`$ioreference`** slight edits (adding commas, etc.) to distinguish main clauses from supporting clauses to clarify conditions;

3 in §7.1.5.10 **`$principal`** add the missing periods to *a*, *b*, and *c*; and

4 in §8.2.7 replace the vertical bars with *vb*, and reorganize the existing semantic text to put it into a coherent narrative order.

2020-03-31 · *X11/TG6/WG1/2020-38 Mumps Draft Standard 2020* [v38]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-19 User-defined* ssvn*s,* with two changes:

1 inserted as §7.1.4.10 **`^$y[`***unspecified***`]`**, and following § was renumbered; and

2 the first two lines are reversed to put the metalanguage first, so this section follows the same pattern used for the other *ssvn*s.

2020-03-31 · *X11/TG6/WG1/2020-37 Mumps Draft Standard 2020* [v37]
TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-29 Local variables in `^$job`,* with three changes:

1 mathematical subscript variables *S* replaced with *s* (lowercase) to follow naming convention used throughout *Standard*;

2 implementation-specific *ecode* value changed from beginning with **`"Z"`** to just **`Z`**, to follow pattern used throughout *Standard* to avoid confusion about whether quotes are part of the value; and

3 instead of adding text to end of §7.1.4.4 `^$job`, it was inserted as §7.1.4.4.3 Characteristic: local variables, and following §§ were renumbered.

Moved §7.1.4.4.4 Characteristic: user & group identification ahead of the § listing *ssvn*s specifying default environments, so the `^$job` subsections are in order alphabetically by header wording.

2020-03-31 · *X11/TG6/WG1/2020-36 Mumps Draft Standard 2020* [v36]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-8* ssvn *for user/group identification,* with one change:

1 instead of adding text to end of §7.1.4.4 `^$job`, it was inserted as §7.1.4.4.3 Characteristic: user & group identification, and following §§ were renumbered.

Proposal §5.5 Related standards activities & liaisons expanded to list MDC subcommittees and task groups associated with all integrated Mumps 2020 extensions to date; this list will be kept up to date to help ensure that related coordination and standardization activities, such as the development of followup proposals, takes place.

2020-03-29 · *X11/TG6/WG1/2020-35 Mumps Draft Standard 2020* [v35]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-26 Canonic form of* ssvn *name,* with no changes.

2020-03-29 · *X11/TG6/WG1/2020-34 Mumps Draft Standard 2020* [v34]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-5 Fix* `algoref`*,* with no changes.

2020-03-29 · *X11/TG6/WG1/2020-33 Mumps Draft Standard 2020* [v33]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-43* `ssvn` *formalization,* with no changes.

2020-03-25/29 · *X11/TG6/WG1/2020-32 Mumps Draft Standard 2020* [v32]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-45* `charset` *names,* with no changes. [Note: this extension contains a serious metalanguage error with its ... operator, which should apply to the previous two items, but since they are not properly grouped together it does not. This needs to be fixed by a follow-up technical proposal, since it is a substantive change to the *Mumps Standard.*]

In §5 Metalanguage description, remove extraneous final period from the semi-metalanguage/semi-text description of the *L* metalanguage operator (trying to make this into a proper sentence ending with a period risks confusing the reader into thinking the final period is actually part of the syntax that *L name* is equivalent to).

2020-03-21/23 · *X11/TG6/WG1/2020-31 Mumps Draft Standard 2020* [v31]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-42* `charset` `ISO-8859-1-USA`, with three changes:

1 fix various typography issues in §4 Definitions;
2 apply ISO-8859-1-USA table's practice of italicizing control-code names to ASCII table;
3 apply ASCII table's practice of annotating ambiguous characters to same characters in ISO-8859-1-USA table; and
4 replace the symbol for character 173 with SHY, since it is a soft-hyphen control character, which is not at all apparent from the forced-in dash of the table in the extension (since a true soft hyphen would be invisible here). [Note: this extension contains mistakes in many of its assignments of pattern codes to characters. These were reasonable choices at the time, before the full impact of Unicode was felt, but with the advantage of hindsight, since ISO-8859-1-USA essentially became the initial characters of Unicode, these pattern codes should be assigned to fit the larger categories of Unicode and not based solely on the limited samples present in ISO-8859-1 itself. This needs to be fixed with a followup technical proposal, since it is a substantive change to the *Mumps Standard.*]

2020-03-20 · *X11/TG6/WG1/2020-30 Mumps Draft Standard 2020* [v30]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-41* `String` *and* `M` *collation,* with three changes:

1 underline several metalanguage elements;
2 format formal definitions; and
3 modernize ASCII table format (eliminate lines, adjust alignment & spacing).

Also, in Annex A ensure character set profile names appear in monospaced typeface, since they are values that appear in the Mumps language.

2020-03-19 · *X11/TG6/WG1/2020-29 Mumps Draft Standard 2020* [v29]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC12/98-13 User-definable I/O handling,* with two changes:

1 add new metalanguage elements to Annex C Metalanguage element dictionary; and
2 underline several metalanguage elements.

Make capitalization of headings consistent (European style).

2020-03-18 · *X11/TG6/WG1/2020-28 Mumps Draft Standard 2020* [v28]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-35 Parameter passing cleanup,* with no changes.

2020-03-17 · *X11/TG6/WG1/2020-27 Mumps Draft Standard 2020* [v27]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-10* mnemonicspec *cleanup,* with one change:

1 what the extension calls the 2nd paragraph was moved yesterday to a later position; the edit was applied to the correct paragraph.

2020-03-16 · *X11/TG6/WG1/2020-26 Mumps Draft Standard 2020* [v26]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-44 Improve* mnemonicspace *handling,* with one change:

1 under **`$key`** in §7.1.5.10, final paragraph, insert § pointers 8.2.25 **`Read`** and 8.2.36 **`Write`**

Simplify text in proposal §4.1.24 *X11/95-132 Parameter passing to a routine.*

2020-03-16 · *X11/TG6/WG1/2020-25 Mumps Draft Standard 2020* [v25]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-95 Portable* controlmnemonic*s and* mnemonicspace*s,* with five changes:

1 instead of §8.2.34 **`Use`**, apply those changes to §8.2.23 **`Open`**, where *mnemonicspace* is actually defined;

2 before appending the new text to the end of §8.2.23, move the three paragraphs from "*Mnemonicspace* specifies the set" to "in the *ssvn* **`^$device`**." so all the semantic text about *mnemonicspace*s and *controlmnemonic*s are grouped together, and so **`open`**'s semantic text begins with its regular cases instead of with *mnemonicspace*s;

3 in §10.1, §10.2, & §10.3 underline "mnemonicspace";

4 in the new text to insert at the end of §10, replace "However," with "Note:" to avoid the awkwardness of two paragraphs back to back opening with "However"; and

5 cut the text ", with the exception that the first 31 characters are uniquely distinguished" because other MDC extensions have already raised the maximum portable name length limit to 31 characters and have standardized how intrpretaiton of uniqueness is handled. (Note that this raises questions about having the maximum portable length limit for labels of the form intlit also be 31 characters, as currently written, because of the usual limits elsewhere upon maximum numeric range.)

Minor updates to table of contents.

2020-03-11/14 · *X11/TG6/WG1/2020-24 Mumps Draft Standard 2020* [v24]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-132 Parameter passing to a routine,* with one change:

1 instead of adding the new text "to the beginning of the 3rd paragraph" insert it before the 3rd paragraph, for greater parallelism with §8.1.6.1's equivalent text.

Reorder documents in §5.4 above.

In §4 standardize capitalization of recently added document titles (European style, as with all documents integrated earlier). Standardize capitalization in §5 to match.

Complete rewrite of §4.1 by TG6 Chair Kenneth W. McGlothlen (2020-02-26/03-04) to simplify, clarify, and standardize how we describe the user impact of MDC extensions; folded into current document by Mr. Marshall.

2020-03-10 · *X11/TG6/WG1/2020-23 Mumps Draft Standard 2020* [v23]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-119 Extended* extid*s,* with no changes.

Passim: replace every use of the word vendor with implementor.

Fix OCR typos in §4.1.8.

2020-03-03 · *X11/TG6/WG1/2020-22 Mumps Draft Standard 2020* [v22]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-118 Undefined* ssvn*s,* with one change:

1 add *ecode* `M60` to Annex B

2020-03-02 · *X11/TG6/WG1/2020-21 Mumps Draft Standard 2020* [v21]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-117* ssvn *collation,* with no changes.

Applied followup change for *X11/95-116* `^$job` *device information*:

1 in Portability §12 second paragraph second sentence, r/7.1.4.9 w/7.1.4.4.3

2020-03-02 · *X11/TG6/WG1/2020-20 Mumps Draft Standard 2020* [v20]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-116* `^$job` *device information,* with one change:

1 list `$io` subscript ahead of `$principal` (alphabetical)

Backfill proposal §5.4 & §5.5.

2020-02-29 · *X11/TG6/WG1/2020-19 Mumps Draft Standard 2020* [v19]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-96 Spaces at end-of-line,* with no change.

2020-02-29 · *X11/TG6/WG1/2020-18 Mumps Draft Standard 2020* [v18]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-94 Parameter passing clarification,* with no change.

2020-02-27 · *X11/TG6/WG1/2020-17 Mumps Draft Standard 2020* [v17]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-91* `$order` *definition,* with no change.

2020-02-26 · *X11/TG6/WG1/2020-16 Mumps Draft Standard 2020* [v16]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-31* `Kill` *indirection,* with no change.

Revise §4 above's display of code examples & quotes from proposals.

Add *jobenv* from *X11/95-2 Execution environment* to Annex C Metalanguage element dictionary.

2020-02-25 · *X11/TG6/WG1/2020-15 Mumps Draft Standard 2020* [v15]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-2 Execution environment,* with four changes:

1 §8.2.18 instead of §5.2.10
2 §7.1.4.5.6 instead of §4.1.xx.9
3 in Section 2 §2.4.2 don't replace `|` with *vb* because it's being used descriptively rather than a metalanguage definition, and it's much clearer in this case to use the `|`.
4 in §7.1.4.5.6 r/`job` *environment* w/*job environment* to match the other three *ssvn*s.

TG19 will need to create a followup technical proposal to clarify the normative text's compliance with the spirit of this extension.

2020-02-24 · *X11/TG6/WG1/2020-14 Mumps Draft Standard 2020* [v14]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/94-33 Kill data and kill subscripts of* glvn*s,* with two changes:

1 §8.2.19 instead of §5.2.11
2 reword final sentence to better match rest of §8.2.19

2020-02-24 · *X11/TG6/WG1/2020-13 Mumps Draft Standard 2020* [v13]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-46* `^$global` *correction,* with no changes.

Also, made corrections identified by Ken McGlothlen:

1 in proposal §2.1 r/1990s era w/1990s-era
2 in proposal §4.1.5 ensure *patatom & alternation* formatted as MEs; r/3.2 w/4.3.5
3 in proposal §4.1.8 r/$Reference w/`$reference`
4 in proposal §4.1.10 r/The portability section w/Section 2 M Portability Requirements; r/interprete w/interpret; r/difffferent w/different; r/case sensitive w/case-sensitive; r/case insensitive w/case-insensitive; r/lower case characters w/lowercase characters.

2020-02-22 · *X11/TG6/WG1/2020-12 Mumps Draft Standard 2020* [v12]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/95-63 Naming string length error,* with one change:

1 add new error to Annex B, with translation "maximum string length exceeded"

Also, go back to *X11/96-13 Portable length limit of* name*s* and make a change:

1 add new error to Annex B, with translation "maximum name length exceeded"

2020-02-21 · *X11/TG6/WG1/2020-11 Mumps Draft Standard 2020* [v11]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/97-23 Portable length limit of strings,* with no changes.

2020-02-20 · *X11/TG6/WG1/2020-10 Mumps Draft Standard 2020* [v10]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-7 Lower-case characters in* names*,* with no changes.

2020-02-19 · *X11/TG6/WG1/2020-9 Mumps Draft Standard 2020* [v9]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/96-13 Portable length limit of* names*,* with the following change:

1. hyphenate thirty-one.

2020-02-18 · *X11/TG6/WG1/2020-8 Mumps Draft Standard 2020* [v8]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/98-30* `New $reference`, with the following change:

1. after the paragraph 3 inserted in §8.2.22 subclause *d*, replaced the original `3` for `new $test` with `4`.

2020-02-18 · *X11/TG6/WG1/2020-7 Mumps Draft Standard 2020* [v7]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-47* `New` svn *addition:* `$test`, with no changes.

2020-02-16/17 · *X11/TG6/WG1/2020-6 Mumps Draft Standard 2020* [v6]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-28 Portable string length,* with the following change:

1. , replaced with ;

Added MV1 to implementation tables in §4.2; fill in details for ISM, GTM, and MV1.

2020-02-15 · *X11/TG6/WG1/2020-5 Mumps Draft Standard 2020* [v5]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-14 Multiple* patatom*s Within* alternation*,* with no additional change required.

2020-02-14 · *X11/TG6/WG1/2020-4 Mumps Draft Standard 2020* [v4]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-4 Two character operators,* with the following changes:

1. § numbers updated from 4.2.2.1, 4.2.2.2, & 4.2.2.3 to 7.2.2.1, 7.2.2.2, & 7.2.2.3.
2. adjusted ordering of lists of operators throughout these sections to be consistent.
3. fixed underlines, strikethroughs, & italics to be consistent with the standard and the formalization itself
4. in 7.2.2.1 add `<=` `>=` `]=` and `]]=` to list of relational operators in first sentence after metalanguage.

2020-02-13 · *X11/TG6/WG1/2020-3 Mumps Draft Standard 2020* [v3]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/SC13/93-33 Effect of* `close $io`, with the following change:

1 § number updated from 5.2.2 to 8.2.7.

2020-02-12 · *X11/TG6/WG1/2020-2 Mumps Draft Standard 2020* [v2]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/93-39 `$reference`,* with the following change:

1 § numbers were updated from 4.1.3.10 to 7.1.5.10, from 4.1.4.13 to 7.1.6.15, and from 5.2.17 to 8.2.28

Upgraded proposal §4 above.

2020-02-02 · *X11/TG6/WG1/2020-1 Mumps Draft Standard 2020* [v1]

TG19/WG2 Chair Frederick D. S. Marshall integrated *X11/94-5 Initialising intrinsics,* with the following changes:

1 § numbers were updated from old 4.1.3.10 to new 7.1.5.10
2 commas were added after "When the process is initiated" for **`$device`**, **`$key`**, **`$io`**, **`$test`**, **`$x`**, and **`$y`**
3 In clause *a* of **`$principal`**, the spelling of *overridden* was corrected

2019-12-26/2020-01-31 · *X11/TG6/WG1/2019-1 Mumps Draft Standard 2020* [v0]

TG19/WG2 Chair Frederick D. S. Marshall systematized document styles and re-typeset according to *X11/TG19/WG2/2019-1 Technical proposal format 2020* to standardize the format and create the baseline text into which MDC extensions may be integrated to create *Mumps 2020.*

2014-01-09/2014-01-28: X11.1 Editor Linda M.R. Yaw did a dry run applying first six MDC extensions to draft standard; interrupted to rethink TG19 authority and processes and to modernize typography.

2008-11-17/2013-10-30: X11.1 Editor Frederick D. S. Marshall explored conversion to modern digital typefaces, replace manual formatting with document styles.

2007-08-07/2008-10-13: X11.1 Editor Frederick D. S. Marshall re-typeset the *Canvass Document for ANSI/MDC X11.1-1994 M,* replace metalanguage text-tables with true tables, apply index and contents markup throughout, ensure PDF navigation table is generated, etc.; typeset as closely as possible to original *Canvass Document.*

6 Appendix · Draft standard

The remainder of this document consists of the *Mumps Draft Standard 2020.*

Abstract

This standard contains a three-section description of the M computer programming language. Section 1, the M Language Specification, consists of a stylized English narrative definition of the M language. Section 2, the M Portability Requirements, identifies constraints on the implementation and use of the language for the benefit of parties interested in achieving M application-code portability. Section 3 is a binding to *ANSI X3.64* (*Terminal Device Control Mnemonics*).

Foreword

M is a high-level interactive computer programming language developed for use in complex data handling operations. It is also known as MUMPS, an acronym for Massachusetts General Hospital Utility Multi-Programming System. The MUMPS Development Committee has accepted responsibility for creation and maintenance of the language since early 1973. The first ANSI approved standard was approved Sept. 15, 1977 via the canvass method. The standard was revised and approved again on November 15, 1984, again on November 11, 1990. Subsequently, the MUMPS Development Committee has met several times annually to consider revisions to the standard.

Document preparation was performed by the MUMPS Development Committee. Suggestions for improvement of this standard are welcome. They should be submitted to the MUMPS Development Committee, c/o MDC Secretariat, 819 North 49th Street, Suite 203, Seattle, Washington 98103.

Contents

Section 1
M language specification

Introduction

Section 1 consists of nine clauses that describe the MUMPS language. Clause 1 describes the metalanguage used in the remainder of Section 1 for the static syntax. The remaining clauses describe the static syntax and overall semantics of the language. The distinction between “static” and “dynamic” syntax is as follows. The static syntax describes the sequence of characters in a routine as it appears on a tape in routine interchange or on a listing. The dynamic syntax describes the sequence of characters that would be encountered by an interpreter during execution of the routine. (There is no requirement that MUMPS actually be interpreted). The dynamic syntax takes into account transfers of control and values produced by indirection.

1 Scope

This standard describes the M programming language.

2 Normative references

The following standard(s) contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standard(s) indicated below. Members of ANSI maintain registers of the currently valid standards.

ANSI X3.135-1992 (Information Systems—Database Language—SQL)
ANSI X3.4-1990 (ASCII Character Set)
ANSI X3.64-1979 R1990 (ANSI Terminal Device Control Mnemonics)
ANSI X11.6-1995 M Windowing API

3 Conformance

3.1 Implementations

A *conforming implementation* shall

a correctly execute all programs conforming to both the *Standard* and the implementation defined features of the implementation

b reject all code that contains errors, where such error detection is required by the *Standard*

c be accompanied by a document which provides a definition of all implementation-defined features and a conformance statement of the form:

"*xxx* version *v* conforms to **X11.1-*yyyy*** with the following exceptions:
. . .
Supported Character Set Profiles are ...
Uniqueness of the values of **$system** is guaranteed by ...
The minimum amount of local variable storage for a job is guaranteed to be ...
The depth of event queues is ...
The number of timer events is ...
The resolution of timers is ..."

where the exceptions are those components of the implementation which violate this *Standard* or for which minimum values are given that are less than those defined in Section 2.

An *MDC conforming implementation* shall be a conforming implementation except that the conforming document shall be this *Standard* together with any such current MDC documents that the implementor chooses to implement. The conformance statement shall be of the form:

"*xxx* version *v* conforms to **X11.1-*yyyy***, as modified by the following MDC documents:
ddd (MDC status *m*)
with the following exceptions:
. . .
Supported Character Set Profiles are ...
Uniqueness of the values of **$system** is guaranteed by ...
The minimum amount of local variable storage for a job is guaranteed to be ...
The depth of event queues is ...
The number of timer events is ...
The resolution of timers is ..."

An *MDC strictly conforming implementation* is an MDC conforming implementation whose MDC modification documents only have MDC Type A status and which has no exceptions.

A *<National Body>. . . implementation* is an implementation conforming to one of the above options in which the requirements of Section 2 are replaced by the **<National Body>** requirements and other extensions required by the **<National Body>** are implemented.

An implementation may claim more than one level of conformance if it provides a switch by which the user is able to select the conformance level.

3.2 Programs

A *strictly conforming program* shall use only the constructs specified in Section 1 of this standard, shall not exceed the limits and restrictions specified in Section 2 of the *Standard* and shall not depend on extensions of an implementation or implementation-dependent features.

A *strictly conforming non-ASCII program* is a strictly conforming program, except that the restrictions to the ASCII character set in Section 2 are removed.

A *strictly conforming `<National Body>` program* is a strictly conforming program, except that the restrictions in Section 2 are replaced by those specified by the **`<National Body>`** and any extensions specified by the **`<National Body>`** may be used.

A *conforming program* is one that is acceptable to a conforming implementation.

4 Definitions

For the purposes of this standard, the following definitions apply.

4.1 argument (of a command): M command words are verbs. Their arguments are the objects on which they act.
4.2 array: M arrays, unlike those of most other computer languages, are trees of unlimited depth and breadth. Every node may optionally contain a value and may also have zero or more descendant nodes. The name of a subscripted variable refers to the root, and the *n*th subscript refers to a node on the nth level. Arrays vary in size as their nodes are set and killed. See *scalar, subscript.*
4.3 atom: a singular, most-basic element of a construction. For example, some atoms in an expression are names of variables and functions, numbers, and string literals.
4.4 block: one or more lines of code within a routine that execute in line as a unit. The argumentless **`do`** command introduces a block, and each of its lines begins with one or more periods. Blocks may be nested. See *level.*
4.5 call by reference: A calling program passes a reference to its actual parameter. If the called subroutine or function changes its formal parameter, the change affects the actual parameter as well. Limited to unsubscripted names of local variables, either scalar or array. See also *call by value.*
4.6 call by value: A calling program passes the value of its actual parameter to a subroutine or function. Limited to a single value, that is, the value of a scalar variable or of one node in an array. See also *call by reference.*
4.7 call: a procedural process of transferring execution control to a *callee* by a *caller.*
4.8 callee: the recipient of a *call.*
4.9 caller: the originator of a *call.*
4.10 character: 1. a member of a set of elements used for the organisation, control, or representation of data. 2. A character is a simple or composite graphic symbol belonging to a conventional set of symbols. There are alphabetic characters, numerical characters (arabic and roman), diacritic characters (for example **^ ˚ˋ ʼ**), punctuation characters (for example **.,;:!?**) and specific other characters (for example **§$%&{#**).

Synonyms which should be avoided: {graphic, phonetic} symbol, sign, mark, note, cipher.

4.11 combining character: a member of an identified subset of the coded character set of *ISO/IEC 10646* intended for combination with the preceding non-combining graphic character, or with a sequence of combining characters preceded by a non-combining character (see also *composite sequence*). Note: This part of *ISO/IEC 10646* specifies several subset collections which include combining characters.

4.12 command: a command word (a verb), an optional conditional expression, and zero or more arguments. Commands initiate all actions in M.

4.13 composite sequence: a sequence of graphic characters consisting of a non-combining character followed by one or more combining characters (see also *combining characters*). Notes: 1. A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence. 2. A composite sequence is not a character and therefore is not a member of the repertoire of *ISO/IEC 10646*.

4.14 computationally equivalent: The result of a procedure is the same as if the code provided were executed by a M program without error. However, there is no implication that executing the code provided is the method by which the result is achieved.

4.15 concatenation: the act or result of joining two strings together to make one string.

4.16 conditional expression: guards a command (sometimes an argument of a command). Only if the expression's value is true does the command execute (on the argument). See *truthvalue.*

4.17 contains: a logical operator that tests whether one string is a substring of another.

4.18 data-cell: in the formal model of M execution. It contains the value and subscripts (if any) of a variable, but not the name of the variable. Many variable names may point to a data-cell due to parameters passed by reference. See also *name-table, value-table.*

4.19 data-record: using a single packed string to store multiple values, either by using a delimiter to separate the individual values into pieces or by using positional definitions to store padded values into extracts.

4.20 default state: the state that is assumed when no state has been explicitly specified.

4.21 descriptor: uniquely defines an element. It comprises various characteristics of the element that distinguish the element from all other similar elements.

4.22 device-dependent: that which depends on the device in question.

4.23 diacritic: character which is not a [letter] of the Latin alphabet and which is placed over, under, or through a letter or a combination of letters indicating a semantic or phonetic value different from that given the unmarked or otherwise marked letter. A letter with a diacritic is a composite character. Note: The point of german "umlaut"-character should be regarded as diacritic [also called a *diacritical mark*].

4.24 diacritical mark: an attribute used of a character applied to denote a variation of a letter.

4.25 digit: a graphic character used to represent the numeric value, or part thereof, of a number. Examples: decimal digits, hexadecimal digits.

4.26 empty: an entity that contains nothing. For example, an empty string contains no characters; it exists but has zero length. See also *null string, NULL character.*

4.27 environment: a set of distinct names. For example, in one global environment all global variables have distinct names. Similar to a directory in many operating systems.

4.28 evaluate: to derive a value.

4.29 execute: to perform the operations specified by the commands of the language.

4.30 extract: to retrieve part of a value, typically contiguous characters from a string.

4.31 extrinsic: a function or variable defined and created by M code, distinct from the primitive functions or special variables of the language. See *intrinsic.*

4.32 follow: to come after according to some ordering sequence. See also *sorts after.*

4.33 function: a value-producing subroutine whose value is determined by its arguments. Intrinsic functions are defined elements of the language, while extrinsic functions are programmed in M.

4.34 global variable: a scalar or array variable that is public, available to more than one job, and persistent, outliving the job. See *local variable.*

4.35 GMT: Greenwich Mean Time.

4.36 graphic character: a character, other than a control character, that has a visual representation normally handwritten, printed, or displayed.

4.37 hidden: unseen. The `NEW` command hides local variables. Also pertains to unseen elements invoked to define the operation of some commands and functions.

4.38 intrinsic: a primitive function or variable defined by the language standard as opposed to one defined by M code. See *extrinsic.*

4.39 job: a single operating system process running a M program.

4.40 job environment: the value of `$SYSTEM` for the process to be initiated by the `JOB` command.

4.41 label: identifies a line of code.

4.42 letter: 1. A letter (or alphabetic character) is a character which is an individual unspecific basic unit of the Latin alphabet, irrespective of the shape and any graphical realization on a medium. A letter can be specified as a *small letter* or *capital letter.* 2. a graphic character used for writing natural language, normally representing a sound of the language.

4.43 level: the depth of nesting of a block of code lines. The first line of a routine is at level 1 and successively nested blocks are at levels 2, 3, … . Formally, the level of a *line* is one plus *li*. Visually, *li* periods follow the label (if any) and precede the body of the line. See *block.*

4.44 ligature: 1. a composite character joining two or more letters. There are ligatures which are conventionalized units of a national variant of the Latin alphabet, and ligatures which are caused by the font used in a document. Maybe the first ones should be named *ligature characters,* the last ones *ligature font elements.* [Language dependent. Only ligature characters are taken into consideration.] 2. two or more letters written together. The resulting symbol is in some cases considered equivalent with the originating letters, in some cases it is considered a separate entity.

4.45 library: A library is a collection of library elements, with unique names, which are referenced using a single library name. A library is defined as being either mandatory or optional.

4.46 library element: A library element is an individual function which is separately defined and accessible from a Mumps process using the library reference syntax.

4.47 local variable: a scalar or array variable that is private to one job, not available to other jobs, and disappears when the job terminates. See *global variable.*

4.48 lock: to claim or obtain exclusive access to a resource.

4.49 mapping: the logical association or substitution of one element for another.

4.50 map: the act of mapping.

4.51 metalanguage: underlined terms used in the formal description of the M language.

4.52 modulo: an arithmetic operator that produces the remainder after division of one operand by another. There are many interpretations of how this operation is performed in the general computing field. M explicitly defines the result of this computation.

4.53 multidimensional: used in reference to arrays to indicate that the array can have more than one dimension.

4.54 Mumps Standard Library: The Mumps Standard Library consists of all libraries and library elements defined within the Mumps Standard, whether mandatory or optional.

4.55 naked: a shorthand reference to one level of the tree forming a global array variable. The full reference is defined dynamically.

4.56 name-table: in the formal model of M execution, a set of variable names and their pointers to data-cells.

4.57 negative: a numeric value less than zero. Zero is not negative.

4.58 node: one element of the tree forming an array. It may have a value and it may have descendants.

4.59 NULL character: the character that is internally coded as code number `0` (zero). A string may contain any number of occurrences of this character (up to the maximum string length). A string consisting of one NULL character has a length of `1` (one).

4.60 null string: 1. a string consisting of `1` (one) NULL character; 2. a string consisting of `0` (zero) characters.

4.61 object: an entity considered as a whole in relation to other entities.

4.62 ordering: bringing strings of characters into a well-defined sequence using a string comparison specification.

4.63 own: to have exclusive access to a resource. In M this pertains to devices.

4.64 parameter: A qualifier of a command modifies its behavior (for example by imposing a time out), or augments its argument (for example by setting characteristics of a device). Some parameters are expressions, and some have the form `keyword=value`. See *argument.*

4.65 parameter (of a function or subroutine): The calling program provides actual parameters. In the called function or subroutine, formal parameters relate by position to the caller's actual arguments. See also *call by reference, call by value, parameter passing.*

4.66 parameter passing: This alliterative phrase refers to the association of actual parameters with formal parameters when calling a subroutine or function.

4.67 partition: the random access memory in which a job runs.

4.68 piece: a part of a string, a sub-string delimited by chosen characters.

4.69 pointer: Indirection allows one M variable to refer, or point to, another variable or the argument of a command.

4.70 portable: M code that conforms to the portability section of the standard.

4.71 positive: a numeric value greater than zero. Zero is not positive.

4.72 post-conditional: see *conditional expression.*

4.73 primitives: the basic elements of the language.

4.74 process-stack: in the formal model of M execution, a push-down stack that controls the execution flow and scope of variables.

4.75 relational: pertaining to operators that compare the values of their operands.

4.76 scalar: single-valued, without descendants. See *array.*

4.77 scope (of a command): the range of other commands affected by the command, as in loop control, block structure, and conditional execution.

4.78 scope (of a local variable): the range of commands for which the variable is visible, from its creation to its deletion, or from its appearance in a `NEW` command to the end of the subroutine, function, or block. Scope is not textual, but dynamic, controlled by the flow of execution.

4.79 sorts after: to come after according to an ordering sequence that is based on a collating algorithm. See also *follows.*

4.80 subscript: an expression whose value specifies one node of an array. Its value may be an integer, a floating point number, or any string. Subscripts are sparse, that is, only those that have been defined appear in the array. See *array, scalar.*

4.81 truthvalue: the value of an expression considered as a number. Non-zero is true, and zero is false.

4.82 tuple: a sequence of a predetermined number of descriptors (usually a name and a series of subscripts) that identifies a member of a set.

4.83 type: M recognizes only one data type, the string of variable length. Arithmetic operations interpret strings as numbers, and logical operations further interpret the numbers as true or false. See also *truthvalue*.

4.84 UCT: Universal Coordinated Time.

4.85 unbound: in the formal model of M execution, the disassociation of a variable's name from its value.

4.86 undefined: pertaining to a variable that is not visible to a command.

4.87 unsubscripted: see *scalar.*

4.88 user-defined mnemonicspace: a mnemonicspace whose controlmnemonics and deviceparameters are defined and implemented by the user in one or more routines.

4.89 value-denoting: representing or having a value.

4.90 value-table: in the formal model of M execution, a set of data-cells.

4.91 variable: M variables may be local or global, scalar or array.

4.92 write-once: a property of an *ssvn* descriptive of the ability of M code to assign a value to it if and only if it does not currently have a `$data` value of `1` or `11`.

5 Metalanguage description

The primitives of the metalanguage are the ASCII characters. The metalanguage operators are defined as follows:

Operator	Meaning
`::=`	definition
[ ]	option
| |	grouping
...	optional indefinite repetition
list{ }	list (comma-delimited list)
∈{ → }	value (datatype/dynamic syntax constraint)
sp	space
cr	carriage-return
lf	line-feed
ff	form-feed
vb	vertical bar
ob	open bracket
cb	close bracket

The following visible representations of ASCII characters required in the defined syntactic objects are used: *sp* (space), *cr* (carriage-return), *lf* (line-feed), *ff* (form-feed), and *vb* (vertical bar). Also, where necessary to avoid confusion with the "option" metalanguage operator, *ob* is used to represent the open bracket character (`[`) and *cb* is used to represent the close bracket character (`]`).

In general, defined syntactic objects will have designators which are italicized names spelled with lower case letters, e.g., *name*, *expr*, etc. Metalanguage names differing only in the use of corresponding upper and lowercase letters are equivalent. Concatenation of syntactic objects is expressed by horizontal juxtaposition; choice is expressed by vertical juxtaposition. The `::=` symbol denotes a syntactic definition. An optional element is enclosed in square brackets `[ ]`, and three dots ... denote that the previous element is optionally repeated any number of times. The definition of *name*, for example, is written:

$$\mathit{name} \ ::= \left| \begin{array}{l} \texttt{\%} \\ \mathit{ident} \end{array} \right| \left[\begin{array}{l} \mathit{digit} \\ \mathit{ident} \end{array} \right] \ldots$$

The vertical bars are used to group elements or to make a choice of elements more readable.

Special care is taken to avoid any danger of confusing the square brackets in the metalanguage with the ASCII graphics `]` and `[`. Normally, the square brackets will stand for the metalanguage symbols.

The unary metalanguage operator *list*{ } denotes a list of one or more occurrences of the syntactic object in the curly braces, with one comma between each pair of occurrences. Thus,

$$\mathit{list}\{\mathit{name}\} \text{ is equivalent to } \mathit{name}[\texttt{,}\mathit{name}]\ldots$$

The binary metalanguage operator ∈{ → } places the constraint on the syntactic object to the left of the → that it must have a value which satisfies the syntax of the syntactic object to the right of the →. For example, one might define the syntax of a hypothetical **`example`** command with its argument list by

$$\mathit{examplecommand} \ ::= \ \texttt{example}\ \mathit{sp}\ \mathit{list}\{\mathit{exampleargument}\}$$

where

$$\mathit{exampleargument} \ ::= \left| \begin{array}{l} \mathit{expr} \\ \texttt{@}\in\{\mathit{expratom} \rightarrow \mathit{list}\{\mathit{exampleargument}\}\} \end{array} \right|$$

This example states: after evaluation of indirection, the command argument list consists of any number of *expr*s separated by commas. In the static syntax (i.e., prior to evaluation of indirection), occurrences of **`@`***expratom* may stand in place of nonoverlapping sublists of command arguments. Usually, the text accompanying a syntax description incorporating indirection will describe the syntax after all occurrences of indirection have been evaluated.

6 Routine *routine*

The *routine* is a string made up of the following symbols:
The *graphic*, including the space character represented as *sp*, and also,
the carriage-return character represented as *cr*,
the line-feed character represented as *lf*,
the form-feed character represented as *ff*.
Each *routine* begins with its *routinehead*, which contains the identifying *routinename*. The *routinehead* is followed by the *routinebody*, which contains the code to be executed. The *routinehead* is not part of the executed code.

routine `::=` *routinehead* *routinebody*

6.1 Routine head *routinehead*

routinehead `::=` *routinename* *eol*
routinename `::=` *name*
name `::=` $\left| \begin{array}{l} \texttt{\%} \\ \textit{ident} \end{array} \right| \left[\begin{array}{l} \textit{digit} \\ \textit{ident} \end{array} \right] \ldots$
control `::=` The `ASCII/M` codes `0-31` and `127` (see Annex A for the definition of `ASCII/M`)
digit `::=` The `ASCII/M` codes `48-57` (characters **`"0"`**-**`"9"`**)
graphic `::=` Those characters in the current charset which are not *control* characters
ident `::=` The `ASCII/M` codes `65-90` and `97-122` (**`"A"`**-**`"Z"`** and **`"a"`**-**`"z"`**) are *ident* characters, all other characters in the range `0-127` are not *ident* characters. Additional characters, with codes greater than `127`, may be defined as *ident* through the algorithm specified in **`^$character(`***charsetexpr***`,"IDENT")`**.
eol `::=` *cr lf*

*Name*s differing only in the use of corresponding upper and lower case letters are not equivalent.

6.2 Routine body *routinebody*

The *routinebody* is a sequence of *line*s terminated by an *eor*. Each *line* starts with one *ls* which may be preceded by an optional *label* and *formallist*. The *ls* is followed by zero or more *li* (level-indicator) which are followed by zero or more *commands* and a terminating *eol*. If there is a *comment* it is separated from the last *command* of a *line* by one or more spaces.

$$\begin{array}{rcl} \mathit{routinebody} & ::= & \mathit{line}\ldots\ \mathit{eor} \\ \mathit{line} & ::= & \left| \begin{array}{l} \mathit{levelline} \\ \mathit{formalline} \end{array} \right| \\ \mathit{eor} & ::= & \mathit{cr}\ \mathit{ff} \end{array}$$

6.2.1 Level line *levelline*

A *levelline* is a *line* that does not contain a *formallist*. A *levelline* may have a *level* greater than one. The *level* of a *line* is the number plus one of *li*. Subclause 6.3 (Routine Execution) describes the effect a *line*'s *level* has on execution.

$$\begin{array}{rcl} \mathit{levelline} & ::= & [\mathit{label}]\ \mathit{ls}\ [\mathit{li}]\ldots\ \mathit{linebody} \\ \mathit{li} & ::= & \mathtt{.}\ [\mathit{sp}]\ldots \end{array}$$

6.2.2 Formal line *formalline*

A *formalline* contains both a *label* and a *formallist* which is a (possibly empty) list of variable *names*. These *names* may contain data passed to this subroutine (see 8.1.8 Parameter passing). A *formallist* shall only be present on a *line* whose *level* is one, i.e., does not contain an *li*.

$$\begin{array}{rcl} \mathit{formalline} & ::= & \mathit{label}\ \mathit{formallist}\ \mathit{ls}\ \mathit{linebody} \\ \mathit{formallist} & ::= & \mathtt{(}[\mathit{list}\{\mathit{name}\}]\mathtt{)} \end{array}$$

If any *name* is present more than once in the same *formallist* an error condition occurs with *ecode*="**M21**".

6.2.3 Label *label*

Each occurrence of a *label* to the left of *ls* in a *line* is called a *defining occurrence* of *label*. An error occurs with *ecode*="**M57**" if there are two or more defining occurrences of *label* with the same spelling in one *routinebody*.

$$\mathit{label} ::= \left| \begin{array}{l} \mathit{name} \\ \mathit{intlit} \end{array} \right|$$

6.2.4 Label separator *ls*

A label separator (*ls*) precedes the *linebody* of each *line*. A *ls* consists of one or more spaces. The flexible number of spaces allows programmers to enhance the readability of their programs.

$$\mathit{ls} ::= \mathit{sp}\ldots$$

6.2.5 Line body *linebody*

The *linebody* consists of an optional sequence of *commands* and an optional *comment*. Note that the *comment* always comes after any *commands* in the *line* (see 8.1.2 for more about *comments*). Individual *commands* are separated by one or more spaces (see 8.1.1 for more about spaces in *commands*). The end of the *line* is terminated by a *cr lf* character sequence.

```
                ┌                              ┐
                │ commands [cs[comment]]       │
linebody  ::=   │ [commands cs] extsyntax      │ eol
                │ comment                      │
                └                              ┘
commands  ::=  command [cs command]...
      cs  ::=  sp...
 comment  ::=  ; [graphic]...
```

The use of the *extsyntax* form is allowed only within the context of an embedded M program (see 6.4 Embedded programs).

6.3 Routine execution

Routines are executed in a sequence of blocks. Each block is dynamically defined and is invoked by the instance of an argumentless **do** command, a **job** command (in the new process), a *doargument*, an *exfunc*, or an *exvar*. Each block consists of a set of *lines* that all have the same *level*; the block begins with the line reference implied by the **do**, **job**, *exfunc*, or *exvar* and ends with an implicit or explicit **quit** command. If no *label* is specified in the *doargument*, *jobargument*, *exfunc*, or *exvar*, the first *line* of the *routinebody* is used. The *execution level* is defined as the level of the *line* currently being executed. *Lines* which have a level greater than the current execution level are ignored, i.e., not executed. An implicit **quit** command is executed when a *line* with a level less than the current execution level or the *eor* is encountered, thus terminating this block (see 8.2.24 for a description of the actions of **quit**). The initial level for a process is one. The argumentless **do** command increases the execution level by one. (See also the **do** command and **goto** command).

Within a given block execution proceeds sequentially from *line* to *line* in top to bottom order. Within a *line*, execution begins at the leftmost *command* and proceeds left to right from *command* to *command*. Routine flow commands **do**, **else**, **for**, **goto**, **if**, **quit**, **trestart**, **xecute**, *exfunc* and *exvar* extrinsic functions and special variables, provide exception to this execution flow. (See also 6.3.2 Error Processing.) In general, each *command*'s argument is evaluated in a left-to-right order, except as explicitly noted elsewhere in this document.

6.3.1 Transaction processing

A *transaction* is the execution of a sequence of *commands* that begins with a **tstart** and ends with either a **tcommit** or a **trollback**, and that is not within the scope of any other transaction. A transaction may be restartable, serializable, or both, depending on parameters specified in the **`tstart`** that initiates the transaction. (See 8.2.33 **`Tstart`**.) These properties affect execution of the transaction as described below.

`Tstart` adds one to the intrinsic special variable **`$tlevel`**, which is initialized to zero when a process begins execution. **`Tcommit`** subtracts one from **`$tlevel`** if **`$tlevel`** is greater than zero. **`Trollback`** sets **`$tlevel`** to zero. A process is within a transaction whenever its **`$tlevel`** value is greater than zero. A process is not within a transaction whenever its **`$tlevel`** value is zero.

If, as a result of a **`tcommit`**, **`$tlevel`** would become zero, an attempt is made to *commit* the transaction. A commit causes the global variable modifications made within the transaction to become durable and accessible to other processes.

A *rollback* is performed if, within a transaction, either a **`trollback`** or a **`halt`** *command* is executed. A rollback rescinds all global variable modifications performed within the scope of the transaction, removes any *nref*s from the *lock-list* that were not included in the *lock-list* when the transaction started (i.e. when **`$tlevel`** changed from zero to one), and removes any *restart context-structure*s for both the transaction linked list and the *process-stack* linked list, discarding the *context-structure*s. M errors do not cause an implicit rollback. (See the **`lock`** *command* for definitions of *nref* and *lock-list*.)

Global variable modifications carried out by *command*s executed within a transaction are subject to the following rules:

a A process that is outside of a transaction cannot access the global variable modifications made within a transaction until that transaction has been committed.

b A process that is inside a transaction is not explicitly excluded from accessing modifications made by other processes. However, a process cannot commit a transaction that has accessed the global variable modifications of any other uncommitted transaction before that other transaction has been committed.

c If the *transparameter*s within the argument to the **`tstart`** initiating the transaction specifies serializability, then all global modifications performed by the transaction and all other concurrently executing transactions must be equivalent to some serial, non-overlapping execution of those transactions.

If it has been determined that a transaction in progress either cannot or is unlikely to conform to the above-stated rules, then the transaction implicitly *restart*s. In addition, the **`trestart`** *command* explicitly causes the transaction to restart.

The actions of a restart depend on whether it is restartable. A transaction is restartable if the initiating **`tstart`** specifies a *restartargument*. (See 8.2.33 **`Tstart`**.) A restart of a restartable transaction causes execution to resume with the initial **`tstart`**. A restart of a non-restartable transaction ends in an error (*ecode*=**`"M27"`**).

The following discussion uses terms defined in the Variable Handling (see 7.1.3.2) and Process-Stack (see 7.1.3.3) models and, like those subclauses, does not imply a required implementation technique. Execution of a restart occurs as follows:

a The frame at the top of the process-stack is examined. If the frame's linked list of *context-structure*s contains entries, they are processed in last-in-first-out order from their creation. If the *context-structure* is exclusive, all entries in the currently active local variable *name-table* are pointed to empty *data-cell*s. In all cases, the *context-structure name-table*s are copied to the currently active *name-table*s. For each *restart context-structure*, **$tlevel** is decremented by one until **$tlevel** reaches `0` (zero) or the list is exhausted. If **$tlevel** does not reach `0` (zero), then:

1 if the frame contains *formallist_*information, it is processed as described by step *d* in the description of the **quit** command (see 8.2.24).

2 the frame is removed and step *a* repeats.

b **$test** and the naked indicator are restored from the *context-structure* that triggered **$tlevel** to reach `0` (zero).

c a rollback is performed. If the transaction is not restartable, restart terminates and an error condition occurs with *ecode*=**"M27"**.

d **$trestart** is incremented by `1`. restart terminates and execution continues with the initial **tstart**, which includes re-evaluating *postcond*, if any, and *tstartargument*, if any.

6.3.2 Error processing

Error trapping provides a mechanism by which a process can execute specifiable commands in the event that **$ecode** becomes non-empty. The following facilities are provided:

The **$etrap** special variable may be set to either the empty string or to code to be invoked when **$ecode** becomes non-empty. Stacking of the contents of **$etrap** is performed via the **new** command.

$ecode provides information describing existing error conditions. **$ecode** is a comma-surrounded list of conditions.

The **$stack** function and **$stack** variable provide stack related information.

$estack counts stack levels since **$estack** was last **new**ed.

An Error Processing transfer of control consists of terminating the current command and processing in the scope of any active **for** commands and indirection. Execution explicitly resumes at the same *level* with two lines where the first *linebody* is the value of **$etrap** and the second *linebody* is `quit:$quit "" quit` as follows:

ls [*li*] *x* *eol*
ls [*li*] `quit:$quit "" quit` *eol*

Where *li* represent the line level at the time of the transfer of control and *x* represents the value of **$etrap**.

For purposes of this transfer each *command* argument is considered to have its own *commandword* (see 8.1 General command rules).

When an error condition is detected, the information about the error is appended to the current value of **$ecode** and to **$stack($stack,"ECODE")**. If appending to **$ecode** or **$stack($stack,"ECODE")** would exceed an implementations maximum string length, the implementation may choose which older information in **$ecode** or

$stack($stack,"ECODE") to discard. The value of **$ecode** may also be replaced via the **set** command.

An Error Processing transfer of control is performed when:

- a The value of **$ecode** is updated to a non-empty value. This occurs when an error condition occurs or may be forced via the **assign** or **set** commands.
- b **$ecode** is not the empty string and a **quit** command removes a *process-stack* level at which **$stack($stack,"ECODE")** would return a non-empty string, and, at the new *process-stack* level, **$stack($stack,"ECODE")** would return an empty string (in other words, when a **quit** takes the process from a frame in which an error occurred to a frame where no error has occurred).

When in the context of error processing (i.e. **$stack($stack,"ECODE")** returns a non-empty string) and a new error occurs (i.e. the value of **$ecode** changes to a different non-empty string), the following actions are performed:

- a It associates the **$stack** information about the failure as if it were associated with the frame identified by **$stack**+1.
- b The following commands are implicitly incorporated into the current execution environment immediately preceding the next *command* in the normal execution sequence:

  ls `trollback:$tlevel  quit:$quit "" quit  ;`

6.3.3 Event processing

Event processing provides a mechanism by which a process can execute specifiable commands in response to some occurrence outside the normal program flow. Event processing can be done using either a synchronous model or an asynchronous model. Synchronous event processing is enabled by issuing the **estart** command, and disabled by issuing the **estop** command. Asynchronous event processing is enabled by issuing the **astart** command, and disabled by issuing the **astop** command. It is possible to temporarily block asynchronous events from being processed using the **ablock** command. This temporary block is released using the **aunblock** command. Events can be generated by running processes using the **etrigger** command.

Asynchronous event processing and synchronous event processing cannot both be enabled at the same time for any event class.

Events are divided into event classes, and those classes are further divided into event IDs. Each event class may be independently enabled, disabled, blocked, and unblocked (except that individual event classes may not be disabled in the synchronous model).

The event classes are:

COMM These are events associated with devices. *evid* is always a *devicexpr* for this class of event. Not all devices necessarily generate events. What devices generate **COMM** events and under what circumstances is determined by the implementation. Use of **COMM** events may not be portable.

HALT **HALT** events are generated when a process terminates. *evid* is **1** for processes which halt by an explicit `halt` command. Other values may be specified by the implementation to correspond to vendor-specific job termination utilities. Use of these other values may not be portable.

`IPC` These are events generated by other processes using the `etrigger` command. The *evid* values are restricted to valid *processid*s. The *evid* value will always be the *processid* of the process which issued the `etrigger` command.

`INTERRUPT` These are events generated by the interruption of a running job in some implementation-specific manner (typically by implementation-specific keyboard commands or job control utilities). Different forms of interrupts may be possible in some implementations, and these may possibly be differentiated by *evid* values. The valid *evid* value(s) is determined by the implementor. Use of `INTERRUPT` events may not be portable.

`POWER` These are events generated when an imminent loss of power can be anticipated (typically because of a signal from the power source). Different types of warnings may be possible in some implementations, and these may possibly be differentiated by *evid* values. The *evid* value(s) is determined by the implementor. Use of `POWER` events may not be portable.

`TIMER` Timer events are generated when a specified interval has elapsed after the timer was set (see `^$event`). *evid* values are *name*s. The implementor may limit the number of concurrent timers available, either by a single process or by the entire M system, or both.

`USER` User events are always generated by `etrigger` commands in the current process. *evid* values are *names*.

`Z`[unspecified] `Z` is the initial letter reserved for defining non-standard event classes. Event class names other than those starting with the letter `Z` are reserved for future enhancement of the *Standard*.

Only those events which have been registered by creating a node in `^$job(`*processid*`,"EVENT",`*evclass*`,`*evid*`)` generate action. In those cases the value of the node is an *entryref* which specifies the event handler. Asynchronous processing of an event (described below) occurs immediately following the event unless the event is blocked.

Blocked events are saved on one of two per-process event queues (one each for synchronous and asynchronous event classes). Each queue is only guaranteed to hold one event, though they may hold more. Events occurring when the queue is full are lost. Queued events are processed in the order they occurred once they are unblocked. It is possible that blocked events will not execute in the order they occurred if the events are of different event classes, and the event classes are separately unblocked in an order different from the order of occurrence of the events. Disabling an event class via `astop` or by killing the appropriate node(s) in `^$event` or `^$job` removes all entries of that class from the event queue.

When a registered event is processed in the asynchronous model, the current value of `$test`, the current execution level, and the current execution location are saved in an extrinsic frame on the *process-stack*. The process then increments the block count on all event classes, and implicitly executes the command

```
do handler
```

where *handler* is the registered event handler. Note that neither **$reference** nor any other shared resource is stacked by this action. If the event handler changes the naked indicator, it may be advisable for it to first **new $reference**. When the process control returns from the handler, the process decrements the block count on all event classes. The value of **$test** and the execution level are restored, the process returns to the stacked execution location and the extrinsic frame is removed from the *process-stack*.

Synchronous event processing is enabled by the **estart** command, which leaves the process in a wait-for-event state. Events are processed sequentially in the order in which they occur. Each event is added to the per-process synchronous event queue. This queue is only guaranteed to hold one event, though it may hold more. Events occurring when the queue is full are lost. When the process is in the wait-for-event state and there is an event in the queue, the event is processed in the synchronous model.

```
do handler
```

where *handler* is the registered event handler. When process control returns from the handler, the process returns to the waiting-for-event state. If the handler executes an **estop** command, the control implictly performs the number of M **quit** commands necessary to return to the execution level of the most recently executed **estart** command, and then terminates that **estart** command.

When a process is initiated, no event processing is enabled, and no nodes in **^$job(***processid***,"EVENT")** are defined. When a process terminates, event processing is implicitly terminated and **^$job(***processid***,"EVENT")** is implicitly killed. Any queued events (synchronous or asynchronous event queues) for that process are discarded.

6.4 Embedded programs

An embedded *xxx* M program is a program which consists of M text and text written to the specifications of the *xxx* programming language or standard. Although it is not a *routine*, an embedded M program conforms to the syntax of a M *routinebody*.

extsyntax ::= **&***extid***(***exttext***)**

exttext ::=
| [*graphic*]...
| [*graphic*]... [*eol* **&** *ls* [*graphic*]...]...

extid ::=
| **SQL**
| **Z**[unspecified]

In *exttext* each *eol* **&** *ls* sequence is either ignored or, if required by the other programming language or standard, replaced by one or more *graphic* characters. *Exttext* is then treated as if the *graphic* characters following the *ls* were part of the previous line (a continuation line).

The exact syntax of the remainder of *exttext* is defined by the external programming language or standard. In the case of *extid* being **SQL** this standard is *X3.135* (see also Annex D). *Extid*s differing only in the use of corresponding upper and lower case letters are equivalent. *Extid*s beginning with the letter **Z** are reserved for futute extension of the language.

Note: An embedded program implies that one or more M routines may be created by some compilation process, replacing any external syntax with appropriate M command lines, function calls etc. An embedded program or embedded program pre-processor does not, therefore, need to adhere to the portability requirements of Section 2 although the equivalent M routines and M implementation should.

7 Expression expr

The expression, *expr*, is the syntactic element which denotes the execution of a value-producing calculation. Expressions are made up of expression atoms separated by binary, string, arithmetic, or truth-valued operators.

expr ::= *expratom* [*exprtail*]...

7.1 Expression atom *expratom*

The expression atom, *expratom*, is the basic value-denoting object of which expressions are built.

$$expratom \; ::= \left| \begin{array}{l} glvn \\ expritem \\ owservice \end{array} \right|$$

7.1.1 Values

All expressions and defined variables have values. These values, whether intermediate or final, may be thought of and operated upon as one of two data types, *mval* or *oref*.

7.1.1.1 Values of data type *mval*

*Mval*s may always be thought of and operated upon as strings. They may be literally represented as *strlits* or *numlits*.

7.1.1.2 Values of data type *oref*

Values of data type *oref* are values that have no canonic representation. Instead, each *oref* uniquely identifies a specific *object*, in an implementation-specific manner.

When a value of data type *oref* is assigned to an *lvn*, this value assignment is the only action that will result. In particular, no copy is made of the *object* that is identified by this *oref*.

All copies of a value of data type *oref* are completely equivalent for accessing the *object*.

After an *lvn* has been assigned a value of data type *oref*, any new value may be assigned to that *lvn*. Such new values may be of data type *mval* as well as *oref*. Such assignments never have any impact on the *object* that is identified by the original *oref*.

Values of data type *oref* have no literal representation. Except in certain special situations, values of data type *oref* are coerced into values of data type *mval* according to the following rules, based on the value of the default *property*, if any, of the *object* that is identified by the *oref*:

a Let `ref` be the value of data type *oref* that is being considered.

b Let `obj` be the *object* identified by `ref`.

c If `obj` has no default *property*, an error will occur with *ecode*=`"M107"` (No Default Value).

d If the value of the default *property* of `obj` has the data type *oref*, then replace `ref` by the value of this default *property* and go back to step `b`.

e Otherwise, the value of the default *property* of `obj` must be of data type *mval*, and this value will be returned as the coerced string value of the original *object*.

The special situations under which values of data type *oref* are not coerced into values of data type *mval* are:

1 The final values of the *expr*s that are operands of `==` operators.

2 The values of parameters that are passed in *actuallist*s or *namedactuallist*s that are not part of *externref*s or **`job`** command arguments.

3 The final values of the *expr*s on the right-hand side of the `=` in arguments of **`assign`** commands.

4 Values that are returned as function-values through **`quit`** commands.

6 Values that are used as the *object* portion of an *owservice*.

7.1.2 Variables

The *M Standard* uses the terms *local variables* and *global variables* somewhat differently from their connotation in certain other computer languages. This subclause provides a definition of these terms as used in the M environment.

An M *routine*, or set of *routine*s, runs in the context of an operating system process. During its execution, the *routine* will create and modify variables that are restricted to its process. These process-specific variables may be stored in primary memory or on secondary peripheral devices such as disks. It can also access (or create) variables that can be shared with other processes. These shared variables will normally be stored on secondary peripheral devices such as disks. At the termination of the process, the process-specific variables cease to exist. The variables created for long term (shared) use remain on auxiliary storage devices where they may be accessed by subsequent processes.

M uses the term *local variable* to denote variables that are created for use during a single process activation. These variables are not available to other processes. However, they are generally available to all routines executed within the process's lifetime. M does include certain constructs, the **new** command and parameter passing, which limit the availability of certain variables to specific routines or parts of routines.

A *global variable* is one that is created by a process, but is permanent and shared. As soon as a process creates, modifies or deletes a global variable outside of a ***transaction***, other processes accessing that global variable outside of a transaction receive its modified form. (See 6.3.1 Transaction processing for a definition of transaction and information on how transactions affect global modifications.) Global variables do not disappear when a process terminates. Like local variables, global variables are available to all routines executed within a process.

M has no explicit declaration or definition statements. Local and global variables, both non-subscripted and subscripted, are automatically created as data is stored into them, and their data contents can be referred to once information has been stored. Since the language has only one data type—string—there is no need for type declarations or explicit data type conversions. Array structures can be multidimensional with data simultaneously stored at all levels including the variable name level. Subscripts can be positive, negative, or zero; they can be integer or noninteger numbers as well as nonnumeric strings (other than empty strings).

7.1.3 Variable name *glvn*

The metalanguage element *glvn* is defined so as to be satisfied by the syntax of *gvn*, *lvn*, or *ssvn*.

$$\mathit{glvn} \; ::= \left| \begin{array}{l} \mathit{lvn} \\ \mathit{gvn} \\ \mathit{ssvn} \end{array} \right|$$

7.1.3.1 Local variable name *lvn*

$$\mathit{lvn} \; ::= \left| \begin{array}{l} \mathit{rlvn} \\ \texttt{@}\in\{\mathit{expratom} \rightarrow \mathit{lvn}\} \end{array} \right|$$

$$\begin{array}{rl} \mathit{rlvn} \; ::= & \left| \begin{array}{l} \mathit{name}\,[\,\texttt{(}\mathit{list}\{\mathit{expr}\}\texttt{)}\,] \\ \texttt{@}\mathit{lnamind}\texttt{@(}\mathit{list}\{\mathit{expr}\}\texttt{)} \end{array} \right| \\ \mathit{lnamind} \; ::= & \in\{\mathit{rexpratom} \rightarrow \mathit{lvn}\} \\ \mathit{rexpratom} \; ::= & \left| \begin{array}{l} \mathit{rlvn} \\ \mathit{rgvn} \\ \mathit{rssvn} \\ \mathit{expritem} \end{array} \right| \end{array}$$

See 7.1.3.4 for the definition of *rgvn*. See 7.1.5 for the definition of *expritem*.

A local variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An unsubscripted occurrence of *lvn* may carry a different value from any subscripted occurrence of *lvn*.

When *lnamind* is present it is always a component of an *rlvn*. If the value of the *rlvn* is a subscripted form of *lvn*, then some of its subscripts may have originated in the *lnamind*. In this case, the subscripts contributed by the *lnamind* appear as the first subscripts in the value of the resulting *rlvn*, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the *rlvn*.

7.1.3.2 Local variable handling

In general, the operation of the local variable symbol table can be viewed as follows. Prior to the initial setting of information into a variable, the data value of that variable is said to be undefined. Data is stored into a variable with commands such as **assign**, **for**, **job**, **merge**, **read**, **set**, **tcommit**, **trestart**, and **trollback**. Subsequent references to that variable return the data value that was most recently stored. When a variable is killed, as with the **kill** command, that variable and all of its array descendants (if any) are deleted, and their data values become undefined.

No explicit syntax is needed for a routine or subroutine to have access to the local variables of its caller. Except when the **new** command or parameter passing is being used, a subroutine or called routine (the callee) has the same set of variable values as its caller and, upon completion of the called routine or subroutine, the caller resumes execution with the same set of variable values as the callee had at its completion.

The **new** command provides scoping of local variables. It causes the current values of a specified set of variables to be saved. The variables are then set to undefined data values. Upon returning to the caller of the current routine or subroutine, the saved values, including any undefined states, are restored to those variables. Parameter passing, including the **do** command, extrinsic functions, and extrinsic variables, allows parameters to be passed into a subroutine or routine without the callee being concerned with the variable names used by the caller for the data being passed or returned.

The formal association of local variables with their values can best be described by a conceptual model. This model is *not* meant to imply an implementation technique for a M implementation.

The value of a variable may be described by a relationship between two structures: the *name-table* and the *value-table*. (In reality, at least two such table sets are required, one pair per executing process for process-specific local variables and one pair for system-wide global variables.) Since the value association process is the same for both types of variables, and since issues of scoping due to parameter passing or nested environments apply only to local variables, the discussion that follows will address only local variable value association. It should be noted, however, that while the overall structures of the table sets are the same, there are two major differences in the way the sets are used. First, the global variable tables are shared. This means that any operations on the global tables, e.g., **set** or **kill**, by one process, affect the tables for all processes. Second, since scoping issues of parameter passing and the **new** command are not applicable to global variables, there is always a one-to-one relationship between entries in the global *name-table* (variable names) and entries in the global *value-table* (values).

The *name-table* consists of a set of entries, each of which contains a *name* and a pointer. This pointer represents a correspondence between that *name* and exactly one *data-cell* from the *value-table*. The *value-table* consists of a set of *data-cell*s, each of which contains zero or more tuples of varying degrees. The degree of a tuple is the number (possibly 0) of elements or subscripts in the tuple list. Each tuple present in the *data-cell* has an associated data value.

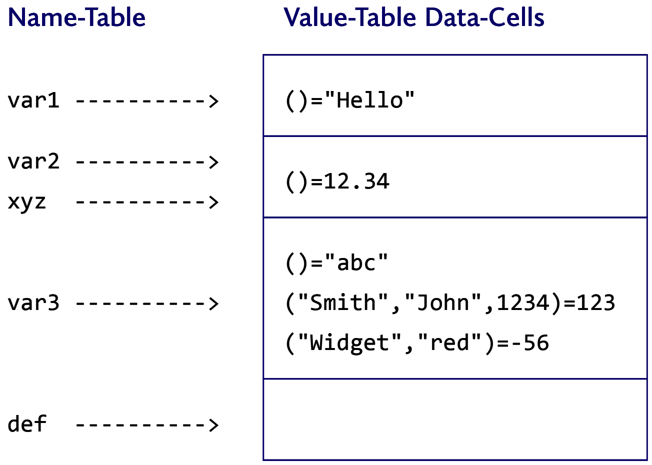
The *name-table* entries contain every non-subscripted variable or array name (*name*) known, or accessible, by the process in the current environment. The *value-table data-cell*s contain the set of tuples that represent all variables currently having data-values for the process. Every *name* (entry) in the *name-table* refers (points) to exactly one *data-cell*, and every entry contains a unique name. Several *name-table* entries (*name*s) can refer to the same *data-cell*, however, and thus there is a many-to-one relationship between (all) *name-table* entries and *data-cell*s. A *name* is said to be *bound* to its corresponding *data-cell* through the pointer in the *name-table* entry. Thus the pointer is used to represent the correspondence and the phrase *change the pointer* is the equivalent to saying *change the correspondence so that a* name *now corresponds to a possible different data-cell (value). Name-table* entries are also placed in the *process-stack* (see 7.1.3.3 Process-Stack).

The value of an unsubscripted *lvn* corresponds to the tuple of degree `0` found in the *data-cell* that is bound to the *name-table* entry containing the *name* of the *lvn*. The value of a subscripted *lvn* (array node) of degree *n* also corresponds to a tuple in the *data-cell* that is bound to the *name-table* entry containing the *name* of the *lvn*. The specific tuple in that *data-cell* is the tuple of degree *n* such that each subscript of the *lvn* has the same value as the corresponding element of the tuple. If the designated tuple doesn't exist in the *data-cell* then the corresponding *lvn* is said to be *undefined.*

In the following figure, the variables and array nodes have the designated data values.

```
var1="Hello"

var2=12.34

var3="abc"

var3("Smith","John",1234)=123

var3("Widget","red")=-56
```

Also, the variable ***def*** existed at one time but no longer has any data or array value, and the variable *xyz* has been bound through parameter passing to the same data and array information as the variable *var2*.



```
Name-Table            Value-Table Data-Cells

var1 ---------->      ()="Hello"

var2 ---------->      ()=12.34
xyz  ---------->

                      ()="abc"
var3 ---------->      ("Smith","John",1234)=123
                      ("Widget","red")=-56

def  ---------->
```

The initial state of a process prior to execution of any M code consists of an empty *name-table* and *value-table*. When information is to be stored (set, given, or assigned) into a variable (*lvn*):

a If the *name* of the *lvn* does not already appear in an entry in the *name-table*, an entry is added to the *name-table* which contains the *name* and a pointer to a new (empty) *data-cell*. The corresponding *data-cell* is added to the *value-table* without any initial tuples.

b Otherwise, the pointer in the *name-table* entry which contained the *name* of the *lvn* is extracted. The operations in steps *c* and *d* refer to tuples in that *data-cell* referred to by this pointer.

c If the *lvn* is unsubscripted, then the tuple of degree `0` in the *data-cell* has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

d If the *lvn* is subscripted, then the tuple of subscripts in the *data-cell* (i.e., the tuple created by dropping the *name* of the *lvn*; the degree of the tuple equals the number of subscripts) has its data value replaced by the new data value. If that tuple did not already exist, it is created with the new data value.

When information is to be retrieved, if the *name* of the *lvn* is not found in the *name-table*, or if its corresponding *data-cell* tuple does not exist, then the data value is said to be undefined. Otherwise, the data value exists and is retrieved. A data value of the empty string (a string of zero length) is not the same as an undefined data value.

When a variable is deleted (killed):

a If the *name* of the *lvn* is not found in the *name-table*, no further action is taken.

b If the *lvn* is unsubscripted, all of the tuples in the corresponding *data-cell* are deleted.

c If the *lvn* is subscripted, let `N` be the degree of the subscript tuple formed by removing the *name* from the *lvn*. All tuples that satisfy the following two conditions are deleted from the corresponding *data-cell*:

1 The degree of the tuple must be greater than or equal to `N`, and
2 The first `N` arguments of the tuple must equal the corresponding subscripts of the *lvn*.

In this formal language model, even if all of the tuples in a *data-cell* are deleted, neither the *data-cell* nor the corresponding *name*s in the *name-table* are ever deleted. Their continued existence is frequently required as a result of parameter passing and the **`new`** command.

7.1.3.3 Process-Stack

The *process-stack* is a virtual last-in-first-out (LIFO) list (a simple push-down stack) used to describe the behavior of M. It is used as an aid in describing how M appears to work and does not imply that an implementation is required to use such a stack to achieve the specified behavior. Three types of items, or frames, will be placed on the *process-stack*, **`do`** frames (including **`xecute`**s), extrinsic frames (including *exfunc*, *exvar*, and asynchronous events) and error frames (for errors that occur during error processing):

a **`Do`** frames contain the execution level and the execution location of the *doargument* or *xargument*. In the case of the argumentless **`do`**, the execution level, the execution location of the **`do`** command and a saved value of **`$test`** are saved. The execution location of a process is a descriptor of the location of the command and possible argument currently being executed. This descriptor includes, at minimum, the *routinename* and the character position following the current command or argument.

b Extrinsic frames contain saved values of **`$test`**, the execution level, and the execution location.

c Error frames contain information about error conditions during error processing (see 6.3.2 Error processing).

The term *context-structure* is used to refer to a set of information related to the maintenance of the process context.

7.1.3.4 Global variable name *gvn*

$$\mathit{gvn}\ \texttt{::=}\ \left|\begin{array}{l} \mathit{rgvn} \\ \texttt{@}\in\{\mathit{expratom}\rightarrow\mathit{gvn}\} \end{array}\right|$$

$$\mathit{rgvn}\ \texttt{::=}\ \left|\begin{array}{l} \texttt{^(}\mathit{list}\{\mathit{expr}\}\texttt{)} \\ \texttt{^}[\mathit{vb}\ \ \mathit{environment}\ \ \mathit{vb}]\ \ \mathit{name}[\texttt{(}\mathit{list}\{\mathit{expr}\}\texttt{)}] \\ \texttt{@}\mathit{gnamind}\texttt{@(}\mathit{list}\{\mathit{expr}\}\texttt{)} \end{array}\right|$$

$$\mathit{gnamind}\ \texttt{::=}\ \in\{\mathit{rexpratom}\rightarrow\mathit{gvn}\}$$

$$\mathit{environment}\ \texttt{::=}\ \mathit{expr}$$

The prefix `^` uniquely denotes a global variable name. A global variable name is either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted. An abbreviated form of subscripted *gvn* is permitted, called the *naked reference,* in which the prefix is present but the *environment*, *name* and an initial (possibly empty) sequence of subscripts is absent but implied by the value of the *naked indicator.* An unsubscripted occurrence of *gvn* may carry a different value from any subscripted occurrence of *gvn*.

When *environment* is present it identifies a specific set of all possible *name*s.

When *gnamind* is present it is always a component of an *rgvn*. If the value of the *rgvn* is a subscripted form of *gvn*, then some of its subscripts may have originated in the *gnamind*. In this case, the subscripts contributed by the *gnamind* appear as the first subscripts in the value of the resulting *rgvn*, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the *rgvn*.

Every executed occurrence of *gvn* affects the naked indicator as follows. If, for any positive integer *m,* the *gvn* has the nonnaked form

$N(v_1,v_2,\ldots,v_m)$

then the *m*-tuple $N,v_1,v_2,\ldots,v_{m-1}$, is placed into the naked indicator when the *gvn* reference is made. A subsequent naked reference of the form

$\hat{}(s_1,s_2,\ldots,s_i)$ (i positive)

results in a global reference of the form

$N(v_1,v_2,\ldots,v_{m-1},s_1,s_2,\ldots,s_i)$

after which the $m+i-1$-tuple $N,v_1,v_2,\ldots,s_{i-1}$ is placed into the naked indicator. Prior to the first executed occurrence of a nonnaked form of *gvn*, the value of the naked indicator is undefined. A nonnaked reference without subscripts or a *rollback*, or a change of the default global *environment* leaves the naked indicator undefined. When a *gvn* is encountered in the form of a naked reference and the naked indicator is undefined, an error condition occurs with *ecode*=**"M1"**.

The effect on the naked indicator described above occurs regardless of the context in which *gvn* is found; in particular, an assignment of a value to a global variable with the command `set` *gvn*=*expr* does not affect the value of the naked indicator until after the right-side *expr* has been evaluated. The effect on the naked indicator of any *gvn* within the right-side *expr* will precede the effect on the naked indicator of the left-side *gvn*.

7.1.4 Structured system variable *ssvn*

ssvn ::= | *rssvn*
| **@**∈{*expratom* → *ssvn*} |

rssvn ::= | **^$**[*vb* *environment* *vb*] *ssvname*[**(***list*{*expr*}**)**]
| **@***ssvnamind***@(***list*{*expr*}**)** |

ssvnamind ::= ∈{*rexpratom* → *ssvn*}

The prefix **^$** denotes a structured system variable name. The parenthesized list of *expr*s following the *ssvname* are called subscripts; a *ssvn* may be either subscripted or unsubscripted; if it is subscripted, any number of subscripts separated by commas is permitted (the allowed values and/or interpretation of each subscript is defined for each individual *ssvname*). Structured system variable names (*ssvname*s) differing only in the use of corresponding upper and lowercase letters are equivalent.

When *ssvnamind* is present it is always the component of a *rssvn*. If the value of the *rssvn* is a subscripted form of *ssvn*, then some of its subscripts may have originated in the *ssvnamind*. In this case, the subscripts contributed by the *ssvnamind* appear as the first subscripts in the value of the resulting *rssvn*, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the *rssvn*.

Values may not be assigned to *ssvn*s and *ssvn*s may not be **kill**ed unless the semantics of these operations are explicitly defined. The *environment* form of the *ssvn* syntax may only refer to the default *environment* unless the *ssvn* is explicitly defined to permit use of *environment*s other than the default. A reference to such an *ssvn* which refers to an *environment* which is not explicitly permitted is erroneous and causes an error condition with *ecode*=**"M59"**. Other references to *ssvn*s using the *environment* syntax however, due to technical reasons or security concerns, may be restricted by implementors to a restricted set of possible *environment*s. An attempt to violate this restriction causes an error condition with an implementor-specified *ecode* beginning with **"Z"**.

The meaning of the individual subscripts of a *ssvn* is explicitly defined for each *ssvn*. The standard contains the following *ssvname*s:

```
            | c[haracter]   |
            | d[evice]      |
            | e[vent]       |
            | g[lobal]      |
            | j[ob]         |
ssvname ::= | l[ock]        |
            | r[outine]     |
            | s[ystem]      |
            | y[unspecified]|
            | z[unspecified]|
```

Unused structured system variable names beginning with an initial letter other than **Z** are reserved for future enhancement of the *Standard.*

7.1.4.1 ^$character

```
^$c[haracter](charsetexpr)
charsetexpr ::= ∈{expr → charset}
```

^$character provides information regarding the available Character Set Profiles on a system, such as collation order and pattern code definitions.

When and only when a Character Set Profile identified by *charset* exist,

^$character(*charset***)** has a value; all nonempty string values are reserved for future extension of the standard.

Data manipulation and the execution of commands within a process are performed in the context of the process *charset*. (See 7.1.4.5 **^$job**)

Input-Transformation

^$character(***charsetexpr*$_1$**,**∈{*expr* → **"INPUT"**}**,***charsetexpr*$_2$)**=∈{*expr* → *algoref*}

algoref ::=
| *emptystring* |
| **$$** *labelref* |
| **$** *externref* |
| **$** *functionname* |

emptystring ::= a string of zero length

This node specifies the input-transformation algorithm which is performed on a string in the process Character Set Profile *charset*$_1$ when it is retrieved from a global or routine which uses *charset*$_2$ or transmitted from a device using *charset*$_2$. The *algoref* specifies the algorithm by which this translation is accomplished, if no input-transformation algorithm is defined, an empty-string value is used. The conversion of the string *`old`* to the string *`new`* using the input-transformation algorithm *`transform`* may be evaluated by executing: `"set new="_transform_"(old)"`.

Output-Transformation

^$character(***charsetexpr*$_1$**,**∈{*expr* → **"OUTPUT"**}**,***charsetexpr*$_2$)=**∈{*expr* → *algoref*}

This node specifies the output-transformation algorithm which is performed on a string in the process Character Set Profile *charset*$_1$ when it is stored in a global or routine which uses *charset*$_2$ or transmitted to a device using *charset*$_2$. The *algoref* specifies the algorithm by which this translation is accomplished, if no output-transformation algorithm is defined, an empty-string value is used. The conversion of the string *`old`* to the string *`new`* using the output-transformation algorithm *`transform`* may be evaluated by executing: `"set new="_transform_"(old)"`.

Valid *name* characters

^$character(*charsetexpr***,**∈{*expr* → **"IDENT"**}**)**=∈{*expr* → *algoref*}

This node specifies the identification algorithm used to determine which characters in a *charset* are valid for use in *names* (i.e. is a character in the set *ident*).

The *ident* truth-value `truth`, of a character `char` using an identification algorithm `ident`, may be evaluated by executing the expression:

```
"set truth="_ident_"($ascii(char))"
```

When `truth` is “true”, `char` is an *ident*; when `truth` is “false”, `char` is not an *ident*. Note that for `$ascii(char)` values less than `128`, `65-90`, and `97-122` are required to be “true” and all other values less than `128` are required to be “false”. If the identification algorithm node is undefined, or is the empty string, then it will return “false” for all `$ascii(char)` greater than `127`; values less than `128` will be returned as indicated.

Patcode definition

^$character(*charsetexpr***,**∈{*expr* → **"PATCODE"**}**,**∈{*expr* → *patcode*}**)**=∈{*expr* → *algoref*}

This node identifies the pattern testing algorithm that determines which characters of *charset* match the specified *patcode*; if this node is not defined, or is the empty string, then no characters in the *charset* will match that *patcode*. The *patcode* truth-value `truth` of a character `char` using a nonempty-string pattern testing algorithm `pattest` may be evaluated by executing the expression:

```
"set truth="_pattest_"($ascii(char))"
```

When `truth` is “true”, `char` belongs to the specified *patcode*; when `truth` is “false”, `char` does not belong to that *patcode*.

Collation Algorithm

^$character(*charsetexpr***,**∈{*expr* → **"COLLATE"**}**)**=∈{*expr* → *algoref*}

This node identifies the collation algorithm for the specified Character Set Profile (*charset*).

Case-conversion Algorithms

^$character(*charsetexpr***,**∈{*expr* → **"LOWER"**}**)**=*algoref*

TThis optional node identifies the algorithm for the conversion of character strings whereby upper-case characters are converted to lower-case ones.

^$character(*charsetexpr***,**∈{*expr* → **"UPPER"**}**)**=*algoref*

TThis optional node identifies the algorithm for the conversion of character strings whereby lower-case characters are converted to upper-case ones.

7.1.4.2 ^$device

^$d[**evice**]**(***devicexpr***)**
devicexpr ::= ∈{*expr* → *device*}
device ::= devicespecifier; an implementation specific device identifier.

^$device provides information about the existence, operational characteristics and availability of devices.

Note: The holding of information about a device when it is not open may be transitory. There are also likely to be more devices in a system which could be opened by a M process than will have information stored in **^$device**.

Device characteristic information for a *device* is stored beneath the **^$device(***devicexpr***)** node:

^$device(*devicexpr***,**∈{*expr* → **"CHARACTER"**}**)=***charsetexpr*

This node identifies the current Character Set Profile of the specified device. The Character Set Profile is assigned to the device in an implementation-specific manner.

^$device(*devicexpr***,**∈{*expr* → *deviceattribute*})

This contains the primary value or values associated with this *deviceattribute*. Additional values may be stored in descendants of this node.

When a device is opened then values for the *deviceattribute*s are created in **^$device**. These may be retained after the device is closed. The range of *deviceattribute* names and the format of the values is defined by the *mnemonicspace* in use for the device.

^$device(*devicexpr***,**∈{*expr* → **"MNEMONICSPACE"**}**)=***mnemonicspace*

This node identifies the *mnemonicspace* currently in effect for the device. If there is no *mnemonicspace* in effect then this node has the value of the empty string.

^$device(*devicexpr***,**∈{*expr* → **"MNEMONICSPEC"**}**,**∈{*expr* → **"MNEMONICSPACE"**}**)=***emptystring*

This node identifies a *mnemonicspace* that has been associated with the device through the **open** and **use** commands. All nonempty string values are reserved for future extension of the standard.

When the *mnemonicspace* in use for the device defines an output time out as described in 8.3.1 it shall also define the following two members of **^$device**:

a the value of **^$device(***devicexpr***,**∈{*expr* → **"OUTTIMEOUT"**}**)** shall equal the value of the most recently executed **OUTTIMEOUT** *deviceparam* for the device. It shall equal `0` when no **OUTTIMEOUT** *deviceparam* has executed for the device.

b the value of **^$device(***devicexpr***,**∈{*expr* → **"OUTSTALLED"**}**)** shall indicate the output time out status of the device. It shall assume the value `0` when the execution of any output-producing argument of a **read** or **write** command begins and it shall assume the value `1` when that argument times out.

7.1.4.3 ^$event

^$e[vent](*eventexpr***)**

eventexpr `::=` *expr* → ∈{ | *einfoattribute* / **EVENTDEF** | }

Note that *einfoattribute* is defined in *X11.6,* the MWAPI standard, along with its semantics.

Nodes under **^$event("EVENTDEF")** are used to identify specific behavior of the named events. Node **^$event("EVENTDEF","TIMER",***timerid***,"INTERVAL")**, where *timerid* is a valid *evid* value for a **TIMER** event, identifies (if positive) the running time remaining before the timer event (in seconds). This value counts down continuously at the rate of `1`/second the corresponding **^$event("EVENTDEF","TIMER",***timerid***,"ACTIVE")** node (see below) evaluates as a *tvexpr* to `1`.

Node **^$event("EVENTDEF","TIMER",***timerid***,"AUTO")**, where *timerid* is a valid *evid* value for a **TIMER** event, is the value set into **^$event("EVENTDEF","TIMER",***timerid***,"INTERVAL")** when it is decremented from a positive value to a non-positive value.

Node **^$event("EVENTDEF","TIMER",***timerid***,"ACTIVE")**, where *timerid* is a valid *evid* value for a **TIMER** event, identifies the state of the timer. If the node evaluates as a *tvexpr* to `1`, the timer is active (running). If the node evaluates as a *tvexpr* to `0`, the timer is inactive.

All of these nodes must be set to establish the timer. If any of the nodes are killed, no timer event occurs.

7.1.4.4 ^$global

^$g[lobal](*gvnexpr***)**
gvnexpr `::=` ∈{*expr* → *name*}

^$global provides information about the existence and characteristics of globals.

When and only when a global identified by *gvnexpr* exists, **^$global(***gvnexpr***)** has a value; all nonempty string values are reserved for future extension of the standard. Global characteristic information is stored beneath the **^$global(***gvnexpr***)** node:

^$global(*gvnexpr***,**∈{*expr* → **"CHARACTER"**}**)=***charsetexpr*

This node identifies the Character Set Profile of the specified global. When the first node in a global is created, and the node **^$global(***gvnexpr***,"CHARACTER")** has a **$data** value of zero, the value assigned is that of **`^$job($job,"CHARACTER")`**. The result of killing a *gvn* does not alter the characteristics stored in **`^$global`** for that *gvn*.

Collation Algorithm

`^$global(`*gvnexpr***`,`**∈{*expr* → **`"COLLATE"`**}**`)`**=∈{*expr* → *algoref*}

This node identifies the collation algorithm to be used when collation is required for a reference to this global. The collation value *order* for a subscript-string *`subscript`*, and a collation algorithm *`collate`* may be determined by executing the expression:

`"set` *`order`*`="_`*`collate`*`_"(`*`subscript`*`)"`

In all cases a collation algorithm must return a distinct *`order`* for each distinct *`subscript`*.

When the first node of a global *`global`* is created, and the collation algorithm node **`^$global("`***`global`***`","COLLATE")`** has a `$data` value of zero, then the value of the current process's Character Set Profile collation algorithm

`$get(^$character(^$job($job,"CHARACTER"),"COLLATE"))`

is assigned as the global's collation algorithm

`^$global("`*`global`***`","COLLATE")`**

7.1.4.5 ^$job

`^$j[ob](`*processid***`)`**
processid `::=` ∈{*expr* → job number}

`^$job` provides information about the existence and characteristics of processes in a system.

When and only when a process identified by *processid* exists, **`^$job(`***processid***`)`** has a value; all nonempty string values are reserved for future enhancement of the standard. Process characteristics are stored beneath the **`^$job(`***processid***`)`** node:

7.1.4.5.1 · Characteristic: character set profile

`^$job(`*processid***`,`**∈{*expr* → **`"CHARACTER"`**}**`)=`***charsetexpr*

This node identifies the active Character Set Profile in use by the process indicated by *processid*. Unless otherwise modified via the *processparameters* of the **`job`** command, when a process is created **`^$job($job,"CHARACTER")`** is set to the *charset* of the process that created it.

7.1.4.5.2 · Characteristic: devices

`^$job(`*processid***`,`**∈{*expr* → **`"$IO"`**}**`)=`***deviceexpr* (current device)
`^$job(`*processid***`,`**∈{*expr* → **`"$PRINCIPAL"`**}**`)=`***deviceexpr* (principal device)

^$job(*processid***,**∈{*expr* → **"OPEN"**}**,***deviceexpr***)** (for each **open**ed device)

These nodes specify the device information associated with process *processid*. The node **"$PRINCIPAL"** is the value of **$principal** for that process. The node **"$IO"** is the value of **$io** (current device being used) for that process. The *devicexpr* nodes beneath **"OPEN"** are the device identifiers for all the devices which are currently **open**ed for that process.

7.1.4.5.3 · Characteristic: events

^$job(*processid***,**∈{*expr* → **"EVENT"**}**,**∈{*expr* → *evclass*}**,***evid***)=***entryref*

This node identifies the events which are enabled for event processing under either the synchronous or asynchronous event processing models, and specifies the event handler which is invoked to process the event. Setting this node enables the specified events for event processing. Killing this node disables the specified events for event processing, and removes all child nodes, even if **kvalue** is used. Implementations are expected to support all of the specified *evclass* and *evid* values with the understanding that some events may never occur on a given implementation. If an *evclass* or *evid* not defined in the *Standard* is used an error occurs with *ecode*=**"M38"**. Attempting to set this node when *evid* cannot be registered due to resource availability will produce an error with *ecode*=**"M110"**.

^$job(*processid***,**∈{*expr* → **"EVENT"**}**,**∈{*expr* → *evclass*}**,***evid***,"MODE")=** | **"DISABLED"**
"SYNCHRONOUS"
"ASYNCHRONOUS" |

This node identifies the processing mode for the specified event by the specified process. If the specified event class is currently enabled for asynchronous event processing by this process (see 8.2.3 **Astart**), the value will be **"ASYNCHRONOUS"**. If the specified event class is currently enabled for synchronous event processing by this process (see 8.2.10 **Estart**), the value will be **"SYNCHRONOUS"**. If the specified event class is not enabled for either form of processing by this process, the value will be **"DISABLED"**.

^$job(*processid***,**∈{*expr* → **"EVENT"**}**,**∈{*expr* → *evclass*}**,***evid***,"BLOCKS")=***intlit*

This node gives the count of blocks (see 8.2.1 **Ablock**, and 8.2.5 **Aunblock**) on the specified event for the specified process. It only exists if the event class is enabled in either synchronous or asynchronous event processing modes. If *intlit*=0, the events are not blocked. If *intlit*>0, the events are blocked.

7.1.4.5.4 · Characteristic: libraries

^$job(***processidexpr***,**∈{$expr_1$ → **"LIBRARY"}**,**$expr_2$**)=***libraryexpr*

This node identifies a *library* currently available to the process. The order in which the *library*s are searched to locate a specific *libraryelement* is defined by the collating order of the values of $expr_2$ for the specified *library*s.

7.1.4.5.5 · Characteristic: local variables

^$job(*processid***,**∈{*expr* → **"VAR"**}**,***lvnexpr***)**
lvnexpr ::= ∈{*expr* → *name*}

This node exists for all local variables in the context of the specified *processid*. The **$data** value of this node is determined by the **$data** value of the specified variable; likewise, the value of this node is the same as that of the specified variable (including undefined). Only variables which have a **$data** value other than zero are represented by these nodes. If a *lvn* is of the form: *Name*(s_1,s_2,…,s_n) for a process *Job*, then the reference to that *lvn* in an *expr* is identical to the reference: **^$job(***Job***,"VAR","***Name***",**s_1**,**s_2**,**…**,**s_n**)**.

Coordination issues may arise if these nodes are examined by another process (if permitted by an implementation); a specific reference may be atomic, but multiple references are not—the specified local variable may be **new**ed or **kill**ed while being examined through these nodes.

Note that for technical reasons or security concerns, implementations may restrict access to **^$job** nodes for *processid*s other than the current *processid*. An attempt to violate such a restriction causes an error condition with an implementor-specified *ecode* beginning with **Z**.

7.1.4.5.6 · Characteristic: user & group identification

^$job(*processid***,**∈{*expr* → **"USER"**}**)=***expr*

This is a write-once *ssvn*. When and only when the process identified by *processid* is associated by the implementation with a user for security purposes, this node contains an implementation specific unambiguous identifier of the user owning the process. At the time of initiation as the result of execution of a **job** command, the *ssvn* value associated with the initiating process is copied to the *ssvn* associated with the new process *processid* unless overridden in an implementation specific manner, by the *processparameters* on the **job** command's *jobargument*. If this node has a `$data` value of `0` or `10`, the process may create the node and assign an unconstrained value to it. When this node has a `$data` value of `1` or `11`, a value may not be assigned nor may the node be `kill`ed. At the termination of the process identified by *processid*, this *ssvn* becomes undefined.

^$job(*processid***,**∈{*expr* → **"GROUP"**}**)=***expr*

This is a write-once *ssvn*. When and only when the process identified by *processid* is associated by the implementation with a group of users for security purposes, this node contains an implementation specific unambiguous identifier of the user group to which the user owning the process belongs. At the time of initiation as the result of execution of a **job** command, the *ssvn* value associated with the initiating process is copied to the *ssvn* associated with the new process *processid* unless overridden, in an implementation specific manner, by the *processparameters* on the **job** command's *jobargument*. If this node has a `$data` value of `0` or `10`, the process may create the node and assign an unconstrained value to it. When this node has a `$data` value of `1` or `11`, a value may not be assigned nor may the node be `kill`ed. At the termination of the process identified by *processid*, this *ssvn* becomes undefined.

When a process attempts to `kill` or assign a value to one of these write-once *ssvn*s already containing a value, an error occurs with *ecode*=**"M96"**.

7.1.4.5.7 · *Ssvn*s specifying default *environment*s

The following *ssvn*s, specifying default *environment*s, are defined. This clause pertains to the following five *ssvn*s:

^$job(*processid***,"DEVICE")**	default device *environment*
^$job(*processid***,"GLOBAL")**	default global *environment*
^$job(*processid***,"JOB")**	default job *environment*
^$job(*processid***,"LOCK")**	default lock *environment*
^$job(*processid***,"ROUTINE")**	default routine *environment*

A process may always obtain and assign a value to these nodes, where *processid*=**$job**. However, for technical reasons or security concerns, implementations may restrict access to these nodes for *processid*s other than the current *processid*. An attempt to violate this restriction causes an error condition with an implementor-specified *ecode* beginning with **"Z"**.

When a process starts, the values of these *ssvn*s are, in general, defined by the implementation. However, a process initiated by a **job** command inherits the default routine *environment*s of the initiating process, unless explicitly specified in the *jobargument*.

Explicit qualification of a *devn*, *gvn*, *nref*, *labelref*, or *routineref* with an *environment* overrides the default *environment* for that one reference.

Assigning a non-existent *environment* to one of these *ssvn*s is not in itself erroneous. However, an attempt to refer to a device, global, lock, or routine in the non-existent *environment* causes an error condition with an *ecode*=**"M26"**.

7.1.4.6 **^$library**

^$li[**brary**]**(***libraryexpr***)**

`^$library` provides information about the availability of libraries and library elements in a system.

When and only when a library *L* exists, `^$library(`*L*`)` has a value; all non-empty string values are reserved for future expansion of the standard. Library information is stored beneath the `^$library(`*library*`)` node:

`^$library(`*libraryexpr*`,`∈{*expr* → `"ELEMENT"`}`,`*libraryelementexpr*`)`
libraryexpr ::= ∈{*expr* → *library*}
libraryelementexpr ::= ∈{*expr* → *libraryelement*}

When and only when a *library* *L* and *libraryelement* *e* exist, `^$library(`*L*`,"LIBRARY",`*e*`)` has a value; all non-empty string values are reserved for future expansion of the *Standard*.

7.1.4.7 `^$lock`

`^$l[ock](`∈{*expr* → *nref*}`)`

will provide information on the existence and operational characteristics of locked *name*s.

7.1.4.8 `^$routine`

`^$r[outine](`*routinexpr*`)`
routinexpr ::= ∈{*expr* → *routinename*}

`^$routine` provides information about the existence and characteristics of routines.

When and only when a routine identified by *routinexpr* exists, `^$routine(`*routinexpr*`)` has a value; all nonempty string values are reserved for future enhancement of the standard. Process characteristics are stored beneath the `^$routine(`*routinexpr*`)` node:

`^$routine(`*routinexpr*`,`∈{*expr* → `"CHARACTER"`}`)=`*charsetexpr*

This node identifies the Character Set Profile in which routine *routinexpr* is stored.

When a *routine* is created and `^$routine(`*routinexpr*`,"CHARACTER")` for that *routine* has a `$data` value of zero, then this node is assigned the current value of the node `^$job($job,"CHARACTER")`.

7.1.4.9 `^$system`

`^$s[ystem](`*systemexpr*`)`
systemexpr ::= ∈{*expr* → *system*}
system ::= syntax of `$system` intrinsic special variable

^$system provides information about the characteristics of systems. A system represents the domain of concurrent processes for which **$job** is unique; the current system is identified by the *svn* **$system**. The second level subscripts of **^$system** not beginning with the letter **"Z"** are reserved for future enhancement of the standard.

System Character Set Profile

^$system(*systemexpr***,**∈{*expr* → **"CHARACTER"**}**)=***charsetexpr*

This node specifies the *charset* which the specified *system* uses for interpretation of all system-wide *name* values (syntactic elements, e.g. *ssvn* names, *commandword*s, *svn* names, etc). Note that this allows an implementation to provide **$z**[*] *name*s, etc. which include *ident*s other than those in ASCII/M.

System Collation Algorithm

^$system(*systemexpr***,**∈{*expr* → **"COLLATE"**}**)=**∈{*expr* → *algoref*}

This node identifies the collation algorithm which the specified *system* uses for determining collation order for system syntactic elements.

7.1.4.10 **^$y**[unspecified]

^$y*name*[**(***list*{*expr*}**)**]

These *ssvn*s are reserved for users and are called user-defined structured system variables.

An implementation is required to provide a means of associating a *routine* with one or more specific user-defined structured system variables.

This *routine* is called whenever a reference is made to a user-defined structured system variable by calling one of the following *label*s in the *routine* with first parameter being **$name** of the reference and the second parameter, if any, as below:

Reference Type	Label	Second parameter	Result?
Evaluation	%VALUE		Yes
$data argument	%DATA		Yes
$get argument	%GET	Second argument of $get	Yes
$order argument	%ORDER	Second argument of $order	Yes
$query argument	%QUERY	Second argument of $query	Yes
kill command	%KILL		
ksubscripts command	%KSUBSCRIPTS		
kvalue command	%KVALUE		
merge command target	%MERGE	Source *glvn*	
merge command source	%MERGES	Target *glvn*	
Value assignment	%SET	Value to be assigned	

The usage of $order and $query with unsubscripted user-defined structured system variables has the same effect as if they were not user-defined.

If user-defined structured system variables are both the source and target of a merge command then only the merge label for the target *ssvn* is called.

If no such *routine* exists when a reference is made to a user-defined structured system variable then an error condition occurs with *ecode*="M97". If no such *label* exists when a reference is made to a user-defined structured system variable then an error condition occurs with *ecode*="M13".

Note: *name*s which differ only in the use of corresponding upper and lower case letters are NOT equivalent.

Note: Users providing such *routine*s are responsible for ensuring that any other side-effects (such as a change to $test or $data values which are not 0, 1, 10 or 11), which would not have taken place had the reference been to a global variable do not occur as a result of calling the *routine*.

7.1.4.11 ^$z[unspecified]

^$z[unspecified](unspecified)

will provide implementation-specific information. Z is the initial letter for defining non-standard structured system variables. The requirement that ^$z be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing programs which observe the rules of the standard.

7.1.5 Expression item *expritem*

$$\mathit{expritem} \ ::= \left| \begin{array}{l} \mathit{strlit} \\ \mathit{numlit} \\ \mathit{exfunc} \\ \mathit{exvar} \\ \mathit{svn} \\ \mathit{function} \\ \mathit{unaryop} \ \ \mathit{expratom} \\ \mathbf{(}\mathit{expr}\mathbf{)} \end{array} \right|$$

7.1.5.1 String literal *strlit*

$$\mathit{strlit} \ ::= \ \texttt{"} \left[\begin{array}{l} \texttt{""} \\ \mathit{nonquote} \end{array} \right] \ldots \ \texttt{"}$$

nonquote ::= any of the characters in *graphic* except the quote character

In words, a string literal is bounded by quotes and contains any string of printable characters, except that when quotes occur inside the string literal, they occur in adjacent pairs. Each such adjacent quote pair denotes a single quote in the value denoted by *strlit*, whereas any other printable character between the bounding quotes denotes itself. An empty string is denoted by exactly two quotes.

7.1.5.2 Numeric literal *numlit*

The integer literal syntax, *intlit*, which is a nonempty string of digits, is defined here.

$$\mathit{intlit} \ ::= \ \mathit{digit}\ldots$$

The numeric literal *numlit* is defined as follows.

$$\mathit{numlit} \ ::= \ \mathit{mant}[\mathit{exp}]$$

$$\mathit{mant} \ ::= \left| \begin{array}{l} \mathit{intlit}[\mathbf{.}\mathit{intlit}] \\ \mathbf{.}\mathit{intlit} \end{array} \right|$$

$$\mathit{exp} \ ::= \ \mathbf{E} \left[\begin{array}{c} \mathbf{+} \\ \mathbf{-} \end{array} \right] \ \mathit{intlit}$$

The value of the string denoted by an occurrence of *numlit* is defined in the following two subclauses.

7.1.5.3 Numeric data values

The set of numeric values is a subset of all values of data type *mval*. Only numbers that may be represented with a finite number of decimal digits are representable as numeric values. A value of data type *mval* has the form of a number if it satisfies the following restrictions.

a It shall contain only digits and the characters `"-"` and `"."`.
b At least one digit must be present.
c `"."` occurs at most once.
d The number zero is represented by the one-character string `"0"`.
e The representation of each positive number contains no `"-"`.
f The representation of each negative number contains the character `"-"` followed by the representation of the positive number which is the absolute value of the negative number. (Thus, the following restrictions describe positive numbers only.)
g The representation of each positive integer contains only digits and no leading zero.
h The representation of each positive number less than `1` consists of a `"."` followed by a nonempty digit string with no trailing zero. (This is called a *fraction*.)
i The representation of each positive non-integer greater than `1` consists of the representation of a positive integer (called the *integer part* of the number) followed by a fraction (called the *fraction part* of the number).

Note that the mapping between representable numbers and representations is one-to-one. An important result of this is that string equality of numeric values is a necessary and sufficient condition of numeric equality.

7.1.5.4 Meaning of *numlit*

Note that *numlit* denotes only nonnegative values. The process of converting the spelling of an occurrence of *numlit* into its numeric data value consists of the following steps.

a If the *mant* has no `"."`, place one at its right end.
b If the *exp* is absent, skip step *c*.
c If the *exp* has a plus or has no sign, move the `"."` a number of decimal digit positions to the right in the *mant* equal to the value of the *intlit* of *exp*, appending zeros to the right of the *mant* as necessary. If the *exp* has a minus sign, move the `"."` a number of decimal digit positions to the left in the *mant* equal to the value of the *intlit* of *exp*, appending zeros to the left of the *mant* as necessary.
d Delete the *exp* and any leading or trailing zeros of the *mant*.
e If the rightmost character is `"."`, remove it.
f If the result is empty, make it `"0"`.

7.1.5.5 Numeric interpretation of data

Certain operations, such as arithmetic, deal with the numeric interpretations of their operands. The numeric interpretation is a mapping from the set of all data values into the set of all numeric values, described by the following algorithm. Note that the numeric interpretation maps numeric values into themselves.

(Note: The *head* of a string is defined to be a substring which contains an identical sequence of characters in the string to the left of a given point and none of the characters in the string to the right of that point. A head may be empty or it may be the entire string.)

Consider the argument to be the string `S`.

First, apply the following sign reduction rules to `S` as many times as possible, in any order.

a If `S` is of the form `+T`, then remove the `+`. (Shorthand: `+T` → `T`)

b `-+T` → `-T`

c `--T` → `T`

Second, apply one of the following, as appropriate.

a If the leftmost character of `S` is not `"-"`, form the longest head of `S` which satisfies the syntax description of *numlit*. Then apply the algorithm of 7.1.5.4 to the result.

b If `S` is of the form `-T`, apply step a above to `T` and append a `"-"` to the left of the result. If the result is `"-0"`, change it to `"0"`.

The *numeric expression numexpr* is defined to have the same syntax as *expr*. Its presence in a syntax description serves to indicate that the numeric interpretation of its value is to be taken when it is executed.

numexpr `::=` *expr*

7.1.5.6 Integer interpretation

Certain functions deal with the integer interpretations of their arguments. The integer interpretation is a mapping from the set of all data values onto the set of all integer values, described by the following algorithm.

First, take the numeric interpretation of the argument. Then remove the fraction, if present. If the result is empty or `"-"`, change it to `"0"`.

The *integer expression intexpr* is defined to have the same syntax as *expr*. Its presence in a syntax definition serves to indicate that the integer interpretation of its value is to be taken when it is executed.

intexpr `::=` *expr*

7.1.5.7 Truth-value interpretation

The truth-value interpretation is a mapping from the set of all data values onto the two integer values `0` (false) and `1` (true), described by the following algorithm. Take the numeric interpretation. If the result is not `"0"`, make it `"1"`.

The *truth-value expression tvexpr* is defined to have the same syntax as *expr*. Its presence in a syntax definition serves to indicate that the truth-value interpretation of its value is to be taken when it is executed.

tvexpr `::=` *expr*

7.1.5.8 Extrinsic function *exfunc*

$$\mathit{exfunc} \ ::= \ \$ \left| \begin{array}{l} \$\ \mathit{labelref} \\ \mathit{externref} \\ \mathit{libraryref} \end{array} \right| \ \mathit{actuallist}$$

Extrinsic functions invoke a subroutine to return a value. When an extrinsic function is executed, the current value of **`$test`**, the current execution level, and the current execution location are saved in an *exfunc* frame on the ***process-stack***. The *actuallist* parameters are then processed as described in 8.1.8.

Execution continues either in the specified *externref* or at the first *command* of the *formalline* specified by the *labelref*. This *formalline* must contain a *formallist* in which the number of *names* is greater than or equal to the number of *names* in the *actuallist*, otherwise an error occurs with *ecode*=`"M58"`. Execution of an *exfunc* to a *levelline* causes an error condition with *ecode*=`"M20"`.

Upon return from the subroutine the value of **`$test`** and the execution level are restored, and the value of the argument of the **`quit`** command that terminated the subroutine is returned as the value of the *exfunc*.

7.1.5.9 Extrinsic special variable *exvar*

$$\mathit{exvar} \ ::= \ \$ \left| \begin{array}{l} \$\ \mathit{labelref} \\ \mathit{externref} \\ \mathit{libraryref} \end{array} \right|$$

An extrinsic special variable whose *labelref* is *x* is identical to the extrinsic function:

`$$`*x*`()`

Note that *label x* must have a (possibly empty) *formallist*.

7.1.5.10 Intrinsic special variable names *svn*

Intrinsic special variables are denoted by the prefix **`$`** followed by one of a designated list of *names*. Intrinsic special variable names differing only in the use of corresponding upper and lower case letters are equivalent. The standard contains the following intrinsic special variable names:

```
d[evice]
ec[ode]
es[tack]
et[rap]
h[orolog]
i[o]
ior[eference]
j[ob]
```

k[ey]
p[rincipal]
pior[eference]
q[uit]
r[eference]
st[ack]
s[torage]
sy[stem]
t[est]
tl[evel]
tr[estart]
x
y
z[unspecified]

Unused intrinsic special variable names beginning with an initial letter other than z are reserved for future enhancement of the *Standard*.

The formal definition of the syntax of *svn* is a choice from among all of the individual *svn* syntax definitions of this subclause.

```
        | syntax of $device intrinsic special variable           |
        | syntax of $ecode intrinsic special variable            |
        |               .                                        |
svn ::= |               .                                        |
        |               .                                        |
        | syntax of $y intrinsic special variable                |
        | syntax of $z[unspecified] intrinsic special variable   |
```

Any implementation of the language must be able to recognize both the abbreviation and the full spelling of each intrinsic special variable name.

Syntax **Definition**

$d[evice]

$device reflects the status of the current device. If the status of the device does not reflect any error-condition, the value of $device, when interpreted as a truth-value, will be 0 (false). If the status of the device would reflect any error-condition, the value of $device, when interpreted as a truth-value, will be 1 (true). When the process is initiated, $device is given the value of the empty string if $io is given a value which is the empty string, otherwise it is given an implementation-dependent value.

$device will give status code and meaning in one access. Its value is one of

Syntax **Definition**

M
M,*I*
M,*I*,*T*

where *M* is an MDC-defined value, *I* is an implementor-defined value, and *T* is explanatory text.

The value of *M*, when interpreted as a truth value, will be equal to `0` (zero) when no significant change of status is being reported. Any nonzero value indicates a significant change of status.

The value of *I* is an implementation-specific value for the relevant status-information.

The value of *T* is implementation specific.

Note: Since *M*, *I*, and *T* are separated by commas, the values of *M* and *I* cannot contain this character.

`$ec[ode]`

contains information about an error condition. When the value of `$ecode` is the empty string, normal routine execution rules are in effect. When `$ecode` contains anything else, the execution rules in 6.3.2 (Error processing) are active. When a process is initiated, but before any commands are processed, the value of `$ecode` is the empty string.

The syntax of a non-empty value returned by `$ecode` is as follows:

`,`*list*{*ecode*}`,`

ecode	::=	`M`
		`U`
		`Z`
			[*noncomma*...]

noncomma	::=	any of the characters in *graphic* except the comma character	

Note: *ecode*s beginning with:

`M` are reserved for the MDC
`U` are reserved for the user
`Z` are reserved for the implementation

All other values are reserved.

`$es[tack]`

counts stack levels in the same way as `$stack`, however, a `new $estack` saves the value of `$estack` and then assigns `$estack` the value of `0`. When a process is initiated, but before any commands are processed, the value of `$estack` is `0` (zero).

`$et[rap]`

contains code which is invoked in the event an error condition occurs. See 6.3.2 Error processing. When a process is initiated, but before any commands are processed, the value of `$etrap` is the empty string.

Syntax **Definition**

The value of $etrap may be stacked with the new command; new $etrap has the effect of saving the current instantiation of $etrap and creating a new instantiation initialized with the same value.

The value of $etrap is changed with the set command. Changing the value of $etrap with a set command instantiates a new trap; it does not save the old trap.

A quit from $etrap, either explicit or implicit (i.e., set $etrap="do ^etrap" has an implicit quit at its end with an empty argument, if appropriate) will function as if a quit had been issued at the "current" $stack. Behavior at the "popped" level will be determined by the value of $ecode. If $ecode is empty, execution proceeds normally. Otherwise, $etrap is invoked at the new level.

$h[orolog]

$horolog gives date and time with one access. Its value is *D*,*S* where *D* is an integer value counting days since an origin specified below, and *S* is an integer value modulo 86,400 counting seconds. The value of $horolog for the first second of December 31, 1840 is defined to be "0,0". *S* increases by 1 each second and *S* clears to 0 with a carry into *D* on the tick of midnight.

$i[o]

$io identifies the current I/O device (see 8.2.7 and 8.2.34). Its value has the form of *expr*. When the process is initiated, $io is given the value of $principal if an implicit open and use for the device specified by $principal is executed by the implementation. If the implementation does not execute this open and use then $io is given the value of the empty string.

$ior[eference]

$ioreference identifies the current I/O device (see 8.2.7 and 8.2.34) Its value has the syntax of *devn* with the following restrictions:

a When the process is initiated, $ioreference is given the value of $principal, if an implicit open and use for the device specified by $principal is executed by the implementation. If the implementation does not execute this open and use, then $ioreference is given the value of the empty string.

b If the last command that changed $ioreference included an *environment*, then the value returned by $ioreference shall include that *environment*; otherwise the value of $ioreference shall not include an *environment*.

c An *environment* whose value has the form of a number as defined in 7.1.5.3 appears as a *numlit*, spelled as its numeric interpretation.

Syntax **Definition**

d An *environment* whose value does not have the form of a number as defined in 7.1.5.3 appears as a *strlit*.

`$j[ob]`

Each executing process has its own job number, a positive integer which is the value of `$job`. The job number of each process is unique to that process within a domain of concurrent processes defined by the implementor. `$job` is constant throughout the active life of a process.

`$k[ey]`

`$key` contains the control-sequence which terminated the last `READ` command from the current device (including any introducing and terminating characters). If no `read` command was issued to the current device or when no terminator was used, the value of `$key` will be the empty string. The effect of a `read *`*glvn* on `$key` is unspecified. When the process is initiated, `$key` is given the value of the empty string if `$io` is given a value which is the empty string, otherwise it is given an implementation-dependent value.

If a Character Set Profile input-transform is in effect, then this is also applied to the value stored in `$key`.

Certain *mnemonicspace*s may also specify that `$key` contains values as a result of other I/O commands.

See 8.2.25 (`read` command) and 8.2.36 (`write` command).

`$p[rincipal]`

`$principal` identifies the principal I/O device.

The principal I/O device is defined in the following fashion:

a If the process is initiated by another MUMPS process then `$principal` is given the value of `$principal` of the initiating process, unless overridden by implementation-specific `job` parameters.

b If the process is initiated from a specific device then `$principal` is given the identifier of the device.

c Otherwise `$principal` is given an implementation-specific value.

`$principal` is constant throughout the active life of a process.

`$pior[eference]`

When the process is initiated, `$pioreference` is give the value of `$principal` with the following restrictions:

a If `$principal` is the empty string than `$pioreference` is the empty string.

b If `$principal` is not the empty string than `$pioreference` shall include an *environment*.

Syntax **Definition**

`$q[uit]`

`$quit` returns `1` if the current *process-stack* frame was invoked by an *exfunc* or *exvar*, and therefore a `quit` would require an argument. Otherwise, `$quit` returns `0` (zero). When a process is initiated, but before any commands are processed, the value of `$quit` is `0` (zero).

`$r[eference]`

`$reference` returns the *namevalue* of the most recently referenced *gvn*, on which the current value of the naked indicator is based; for the behavior after a reference to the function `$query` see 7.1.6.16.

The initial value of `$reference` is the empty string. The value of `$reference` may be set to either the empty string, or to a *namevalue*. indicating a *gvn*. A side-effect of setting `$reference` equal to the empty string is that the naked indicator will become undefined. A side-effect of setting `$reference` to a *namevalue* is that the naked indicator will change as if the indicated *gvn* had been referenced.

`$st[ack]`

`$stack` gives the current level of the *process-stack*. `$stack` contains an integer value of zero or greater. When a process is initiated, but before any commands are processed, the value of `$stack` is `0` (zero). See 7.1.3.3 (process-stack) for a description of stack behavior.

`$s[torage]`

Each implementation must return for the value of `$storage` an integer which is the number of characters of free space available for use. The method of arriving at the value of `$storage` is not part of the standard.

`$sy[stem]`

Each implementation must return a value in `$system` which represents uniquely the system representing the domain of concurrent processes for which `$job` is unique. Its value is `V,S` where `V` is an integer value allocated by the MDC to an implementor and `S` is defined by that implementor in such a way as to be able to be unique for all the implementor's systems.

`$t[est]`

`$test` contains the truth value computed from the execution of an `if` command containing an argument, or an `open`, `lock`, `job`, or `read` command with a timeout (see 7.1.5.8, 7.1.5.9, and 8.2.8). When the process is initiated, `$test` is given the value `0` (false).

`$tl[evel]`

`$tlevel` indicates whether a *transaction* is currently in progress. It is initialized to zero when a process begins. `Tstart` adds `1` to `$tlevel`. When `$tlevel` is greater than zero, `tcommit` subtracts `1` from `$tlevel`. A *rollback* or *restart* sets `$tlevel` to zero.

Syntax **Definition**

`$tr[estart]`

`$trestart` indicates how many *restart*s have occurred since the initiation of a *transaction*. It is initialized to zero when a process begins, and set to zero by the successful completion of `tcommit` or `trollback`. Each restart adds `1` to `$trestart`.

`$x`

`$x` has a nonnegative integer value which approximates the value of the horizontal co-ordinate of the active position on the current device. It is initialized to zero by any control-function or *format* that involves a move to the start of a line. When the process is initiated, `$x` is given the value `0` if `$io` is given a value which is the empty string, `""`. Otherwise it is given an implementation-dependent value.

The unit in which `$x` is expressed is initially equal to “characters”. Certain *format*s may change this.

When any control-function would leave the cursor in a position so that the horizontal co-ordinate would be uncertain, the value of `$x` will not be changed. In such cases the value of `$device` will be an error-code.

If a Character Set Profile input-transform is in effect, then `$x` is modified in accordance with the input prior to any transform taking place. If a Character Set Profile output-transform is in effect, then `$x` is modified in accordance with the output after any transform takes place.

See 8.2.25 (`read` command) 8.2.34 (`use` command) and 8.2.36 (`write` command).

`$y`

`$y` has a nonnegative integer value which approximates the value of the vertical co-ordinate of the active position on the current device. It is initialized to zero by any control-function or *format* that involves a move to the start of a page. When the process is initiated, `$y` is given the value `0` if `$io` is given a value which is the empty string, `""`. Otherwise it is given an implementation-dependent value.

The unit in which `$y` is expressed is initially equal to “lines”. Certain *format*s may change this.

When any control-function would leave the cursor in a position so that the vertical co-ordinate would be uncertain, the value of `$y` will not be changed. In such cases, the value of `$device` will be an error-code.

If a Character Set Profile input-transform is in effect, then `$y` is modified in accordance with the input prior to any transform taking place. If a Character Set Profile output-transform is in effect, then `$y` is modified in accordance with the output after any transform takes place.

Syntax	Definition
	See 8.2.25 (`read` command) 8.2.34 (`use` command) and 8.2.36 (`write` command).
`$z`[unspecified]	`Z` is the initial letter reserved for defining non-standard intrinsic special variables. The requirement that `$z` be used permits the unused initial letters to be reserved for future enhancement of the standard without altering the execution of existing routines which observe the rules of the standard.

7.1.5.11 Unary operator *unaryop*

unaryop `::=` | `'` (note: apostrophe) | `+` | `-` (note: hyphen) |

There are three unary operators: `'` (not), `+` (plus), and `-` (minus).

Not inverts the truth value of the *expratom* immediately to its right. The value of `'`*expratom* is `1` if the truth-value interpretation of *expratom* is `0`; otherwise its value is `0`. Note that `''` performs the truth-value interpretation.

Plus is merely an explicit means of taking a numeric interpretation. The value of `+`*expratom* is the numeric interpretation of the value of *expratom*.

Minus negates the numeric interpretation of *expratom*. The value of `-`*expratom* is the numeric interpretation of `-`*`N`*, where *`N`* is the value of *expratom*.

Note that the order of application of unary operators is right-to-left.

7.1.5.12 Name value *namevalue*

namevalue `::=` *expr*

A *namevalue* has the syntax of a *glvn* with the following restrictions:

a The *glvn* is not a naked reference.

b Each subscript whose value has the form of a number appears as specified in 7.1.5.3.

c Each subscript whose value does not have the form of a number as defined in 7.1.5.3 appears as a *sublit*, defined as follows:

sublit `::=` `"` | `""` / *subnonquote* | ... `"`

where *subnonquote* is defined as follows:

subnonquote `::=` any character valid in a subscript, excluding the quote symbol d The *environment* appears as defined in *b* and *c* for subscripts.

e If the *glvn* is an *ssvn*, the name part of the *ssvn* will appear in uppercase in the unabbreviated form.

7.1.6 Intrinsic function *function*

Intrinsic functions are denoted by the prefix **$** followed by one of a designated list of *name*s, followed by a parenthesized argument list. Intrinsic function names differing only in the use of corresponding upper and lower case letters are equivalent. The following function names are defined:

functionname ::=	**a**[**scii**] **c**[**har**] **d**[**ata**] **de**[**xtract**] **dp**[**iece**] **e**[**xtract**] **f**[**ind**] **fn**[**umber**] **g**[**et**] **h**[**orolog**] **j**[**ustify**] **l**[**ength**] **m**[**umps**] **na**[**me**] **o**[**rder**] **p**[**iece**] **ql**[**ength**] **qs**[**ubscript**] **q**[**uery**] **r**[**andom**] **re**[**verse**] **s**[**elect**] **st**[**ack**] **t**[**ext**] **tr**[**anslate**] **v**[**iew**] **z**[unspecified]

Unused function names beginning with an initial letter other than **z** are reserved for future enhancement of the *Standard*.

The formal definition of the syntax of *function* is a choice from among all of the individual *function* syntax definitions in this subclause.

function ::=
syntax of **$ascii** function
syntax of **$char** function
.
.
.
syntax of **$view** function
syntax of **$z**[unspecified] function

Any implementation of the language must be able to recognize both the abbreviation and the full spelling of each function name.

7.1.6.1 `$ascii`

$a[**scii**]**(***expr***)**

This form produces an integer value as follows:

a `-1` if the value of *expr* is the empty string.

b Otherwise, an integer *n* associated with the leftmost character of the value of *expr*, such that `$ascii($char(`*n*`))=`*n*.

$a[**scii**]**(***expr***,***intexpr***)**

This form is similar to **$ascii(***expr***)** except that it works with the *intexpr*th character of *expr* instead of the first. Formally, **$ascii(***expr***,***intexpr***)** is defined to be `$ascii($extract(`*expr*`,`*intexpr*`))`.

7.1.6.2 `$char`

$c[**har**]**(***list*{*intexpr*}**)**

This form returns a string whose length is the number of argument expressions which have nonnegative values. Each *intexpr* in the closed interval `[0,127]` maps into the ASCII character whose code is the value of *intexpr*; this mapping is order-preserving. Each negative-valued *intexpr* maps into no character in the value of **$char**. Each *intexpr* greater than `127` maps into a character in a manner defined by the current *charset* of the process.

7.1.6.3 `$data`

$d[**ata**]**(***glvn***)**

This form returns a nonnegative integer which is a characterization of the *glvn*. The value of the integer is *p*+*d*, where:

`d=1` if the *glvn* has a defined value, i.e., the *name-table* entry for the *name* of the *glvn* exists, and the subscript tuple of the *glvn* has a corresponding entry in the associated *data-cell*; otherwise, `d=0`.

`p=10` if the variable has descendants; i.e., there exists at least one tuple in the *glvn*'s *data-cell* which satisfies the following conditions:

a The degree of the tuple is greater than the degree of the *glvn*, and

b the first `N` arguments of the tuple are equal to the corresponding subscripts of the *glvn* where `N` is the number of subscripts in the *glvn*.

If no *name-table* entry for the *glvn* exists, or no such tuple exists in the associated *data-cell*, then `p=0`.

7.1.6.4 $dextract

$de[**xtract**]**(**[*initialrecordvalue*]**,***extracttemplate***,***list*[*recordfieldvalue*]**)**

initialrecordvalue	::=	*expr*
extracttemplate	::=	∈{*expr* → *extractfields*}
extractfields	::=	*list*{[**-**]*fieldwidth*[**:***fieldindex*]}
fieldwidth	::=	*intlit*
fieldindex	::=	*intlit*
recordfieldvalue	::=	*expr*[**:***fieldindex*]

This function assembles the *expr*s of the *recordfieldvalue*s into a single value. The value of *initialrecordvalue* is used as the starting value to which the *recordfieldvalue*s are applied. If *initialrecordvalue* is omitted, the empty string is used.

The *extracttemplate* specifies the **$extract** partitioning (*fieldwidth*s and alignments) of *initialrecordvalue* into consecutive fields. Unsigned values specify width for left-aligned fields. Negative values specify width (absolute value of *fieldwidth*) for right-aligned fields. The *fieldindex* specifies the relative field number. If omitted, it defaults to the next successive value. For omitted *recordfieldva1ue*s the corresponding field is obtained from the *initialrecordvalue*. Although *recordfieldvalue* is optional, at least one *recordfieldvalue* (not necessarily the first) in the list must be nonempty.

Left-aligned fields are padded on the right with `$char(32)` or truncated on the right as needed. Right-aligned fields are padded on the left with `$char(32)` or truncated on the left as needed.

Assignment to fields proceeds in a left-to-right fashion. If a field is referenced multiple times, the rightmost value is the final value of the field.

7.1.6.5 $dpiece

$dp[**iece**]**(**[*initialrecordvalue*]**,***piecedelimiter***,***list*[*recordfieldvalue*]**)**

piecedelimiter `::=` *expr*

This function assembles the *expr*s of *recordfieldvalue*s into a single value. The value of *initialrecordvalue* is used as the starting value to which the *recordfieldvalue*s are applied. If *initialrecordvalue* is omitted, the empty string is used. The *piecedelimiter* specifies the relative field number. If omitted, it defaults to the next successive value. For omitted *recordfieldvalue*s the corresponding field is obtained from the *initialrecordvalue*. Although *recordfieldvalue* is optional, at least one *recordfieldvalue* (not necessarily the first) in the list must be nonempty.

7.1.6.6 `$extract`

`$e[xtract](`*expr*`)`

This form returns the first (leftmost) character of the value of *expr*. If the value of *expr* is the empty string, the empty string is returned.

`$e[xtract](`*expr*`,`*intexpr*`)`

Let *s* be the value of *expr*, and let *m* be the integer value of *intexpr*. `$extract(s,m)` returns the *m*th character of `s`. If *m* is less than `1` or greater than `$length(s)`, the value of `$extract` is the empty string. (`1` corresponds to the leftmost character of `s`; `$length(s)` corresponds to the rightmost character.)

`$e[xtract](`*expr*`,`$\mathit{intexpr}_1$`,`$\mathit{intexpr}_2$`)`

Let *n* be the integer value of $\mathit{intexpr}_2$. `$extract(s,m,n)` returns the string between positions *m* and *n* of `s`. The following cases are defined:

a	`m>n`	Then the value of `$e` is the empty string
b	`m=n`	`$e(s,m,n)=$e(s,m)`
c	`m<n'>$l(s)`	`$e(s,m,n)=$e(s,m)` concatenated with `$e(s,m+1,n)`. That is, using the concatenation operator `_` of 7.2.1.1, `$e(s,m,n)=$e(s,m)_$e(s,m+1)_..._$e(s,m+(n-m))`
d	`m<n` & `$l(s)<n`	`$e(s,m,n)=$e(s,m,$l(s))`

7.1.6.7 `$find`

`$f[ind](`expr_1`,`expr_2`)`

This form searches for the leftmost occurrence of the value of expr_2 in the value of expr_1. If none is found, `$find` returns zero. If one is found, the value returned is the integer representing the number of the character position immediately to the right of the rightmost character of the found occurrence of *expr*$_2$ in *expr*$_1$. In particular, if the value of *expr*$_2$ is empty, **$find** returns 1.

$f[ind](***expr*$_1$**,***expr*$_2$,***intexpr***)**

Let *a* be the value of *expr*$_1$, let *b* be the value of *expr*$_2$, and let *m* be the value of *intexpr*. **$find(***a***,***b***,***m***)** searches for the leftmost occurrence of *b* in *a*, beginning the search at the *max*(*m*,1) position of *a*. Let *p* be the value of the result of $find($extract(*a*,*m*, $length(*a*)),*b*). If no instance of *b* is found (i.e., *p*=0), **$find** returns the value 0; otherwise, **$find(***a***,***b***,***m***)**=*p*+*max*(*m*,1)-1.

7.1.6.8 $fnumber

$fn[umber](*numexpr***,***fncodexpr***)**

```
fncodexpr ::= ∈{expr→fncode}
   fncode ::= [fncodatom...]
            | fncodp  |
            | fncodt  |
fncodatom ::= | ,       | (note: comma)
            | +       |
            | -       | (note: hyphen)

   fncodp ::= | P |
            | p |

   fncodt ::= | T |
            | t |
```

This form shall return a value that is the value of *numexpr* edited by applying each *fncodatom* according to the following rules. The order of application is not significant. Note: Zero is neither positive nor negative.

fncodatom	**Action**
fncodp	Represent negative *numexpr* values in parentheses. Let *A* be the absolute value of *numexpr*. Use of *fncodp* will result in the following: 1 If *numexpr*<0, the result will be "("_*A*_")". 2 If *numexpr*'<0, the result will be " "_*A*_" ".
fncodt	Represent *numexpr* with a trailing rather than a leading "+" or "-" sign. Note: if sign suppression is in force (either by default on positive values, or by design using the "-" *fncodatom*), use of *fncodt* will result in a trailing space character.

fncodatom	Action
`,`	Insert comma delimiters every third position to the left of the decimal (present or assumed) within *numexpr*. Note: no comma shall be inserted which would result in a leading comma character.
`+`	Force a plus sign (`"+"`) on positive values of *numexpr*. Position of the `"+"` (leading or trailing) is dependent on whether or not *fncodt* is present.
`-`	Suppress the negative sign `"-"` on negative values of *numexpr*.

All other values for *fncodatom* are reserved.

If *fncodexpr* equals an empty string, no special formatting is performed and the result of the expression is the original value of *numexpr*.

More than one occurrence of a particular *fncodatom* within a single *fncode* is identical to a single occurrence of that *fncodatom*. Erroneous conditions are produced, with *ecode*=`"M2"`, when a *fncodp* is present with any of the sign suppression or sign placement *fncodatom*s (`"+-"` or *fncodt*).

`$fn[umber](`*numexpr*,*fncodexpr*,*intexpr***`)`**

This form is identical to the two-argument form of **`$fnumber`**, except that *numexpr* is rounded to *intexpr* fraction digits, including possible trailing zeros, before processing any *fncodatom*s. If *intexpr* is zero, the evaluated *numexpr* contains no decimal point. Note: if `(-1<`*numexpr*`<1)`, the result of this form of **`$fnumber`** has a leading zero (`"0"`) to the left of the decimal point. Negative values of *intexpr* are reserved for future extensions of the **`$fnumber`** function.

7.1.6.9 `$get`

`$g[et](`*glvn***`)`**

This form returns the value of the specified *glvn* depending on its state, defined by **`$data(`***glvn***`)`**. The following cases are defined:

a `$data(`*glvn*`)#10=1`. The value returned is the value of the variable specified by *glvn*.
b Otherwise, the value returned is the empty string.

`$g[et](`*glvn*,*expr***`)`**

This form returns the value of the specified *glvn* depending on its state, defined by **`$data(`***glvn***`)`**. The following cases are defined:

a `$data(`*glvn*`)#10=1`. The value returned is the value of the variable specified by *glvn*.
b Otherwise, the value returned is the value of *expr*.

Both *glvn* and *expr* will be evaluated before the function returns a value, so that the behavior of this function with respect to the naked indicator is well defined.

7.1.6.10 `$horolog`

`$h[orolog](`*intexpr***`)`**

This gives date, time, and time-offset with one access. Let *m* be the value of *intexpr*. The following cases are defined:

- *D* an integer value counting days since the origin used by the **`$horolog`** intrinsic variable. (Like the *D* in **`$horolog`**)
- *S* a numeric value counting the seconds since local midnight (like the *S* in **`$horolog`** but not restricted to integer values, if the M implementation can supply it).
- *TO* the number of seconds in thc time-offset needed to get UCT (Greenwich Mean Time) from local time (*Localtime*+*TO*=*UCT*).

a *m*=`0`. Returns a string in the format *D*,*S*,*TO* where *S* is restricted to an integer value (the value of **`$horolog`** with offset).

b *m*=`1`. Returns a string for local time in the format *D*,*S*,*TO* where *S* is not restricted to an integer value.

c *m*=`-1`. Returns a string for UCT in the format *D*,*S* where *S* is not restricted to an integer value.

d All other values of *m* are reserved.

7.1.6.11 `$justify`

`$j[ustify](`*expr***`,`***intexpr***`)`**

This form returns the value of *expr* right-justified in a field of *intexpr* spaces. Let *m* be **`$length(`***expr***`)`** and *n* be the value of *intexpr*. The following cases are defined:

a *m*`'<`*n*. Then the value returned is *expr*.

b Otherwise, the value returned is *S*(*n*-*m*) concatenated with *expr*$_1$, where *S*(*x*) is a string of *x* spaces.

`$j[ustify](`*numexpr***`,`***intexpr*$_1$**`,`***intexpr*$_2$**`)`**

This form returns an edited form of the number *numexpr*. Let *r* be the value of *numexpr* after rounding to *intexpr*$_2$ fraction digits, including possible trailing zeros. (If *intexpr*$_2$ is the value `0`, *r* contains no decimal point.) The value returned is **`$justify(`***r***`,`***intexpr*$_1$**`)`**. Note that if `-1<`*numexpr*`<1`, the result of **`$justify`** does have a zero to the left of the decimal point. Negative values of *intexpr*$_2$ are reserved for future extensions of the **`$justify`** function.

7.1.6.12 `$length`

`$l[ength](`*expr*`)`

This form returns an integer which is the number of characters in the value of *expr*. If the value of *expr* is the empty string, `$length(`*expr*`)` returns the value `0`.

`$l[ength](`*expr*$_1$`,`*expr*$_2$`)`

This form returns the number plus one of nonoverlapping occurrences of *expr*$_2$ in *expr*$_1$. If the value of *expr*$_2$ is the empty string, then `$length` returns the value `0`.

7.1.6.13 `$mumps`

`$m[umps](`*expr*`)`

Let *s* be the value of *expr* with *eol* appended.

- a If *s* matches the syntactic definition of a *line*, without any implementation (`"z"`) extensions, `$mumps` returns the value `0`.
- b If *s* matches the syntactic definition of a *line* only because of syntactic extensions to the language available in the current implementation, `$mumps` either returns the value `0` or provides a return value as in case *c*.
- c If *s* does not match the syntactic definition of a *line*, the function returns a value of the form
  list{*mumpsreturn*}

mumpsreturn `::=` *intlit*`;`*ecode*`;`[*noncommasemi*...]
noncommasemi `::=` Any of the characters in *graphic* except the comma character and the semicolon character

In each *mumpsreturn*, the *ecode* must be one which actually describes an erroneous condition in *s*. If no standard *ecode* describes an erroneous condition in *s*, an *ecode* of `S0` shall be used. All nonpositive values of *intlit* are reserved for future enhancement of the standard.

7.1.6.14 `$name`

`$na[me](`*glvn*`)`

This form returns a string value which is the *namevalue* denoting the named *glvn*. Note that naked references are permitted in the argument, but that the returned value is always a non-naked reference. If *glvn* includes an *environment*, then the *namevalue* shall include that *environment*; otherwise the *namevalue* shall not include an *environment*.

$na[**me**]**(***glvn***,***intexpr***)**

This form returns a string value which is a *namevalue* denoting either all or part of the supplied *glvn*, depending on the value of *intexpr*. Let **$name(***glvn***)** applied to the supplied *glvn* be of the form *Name*(s_1,s_2,…,s_n), considering *n* to be zero if the *glvn* has no subscripts, and let *m* be the value of *intexpr*. Then **$name(***glvn***,***intexpr***)** is defined as follows:

1 It is erroneous for *m* to be less than zero (*ecode*="**M39**").
2 If *m*=0, the result is *Name*.
3 If *n*>*m,* the function returns the string returned by $name(*Name*($s_1$,$s_2$,…,$s_m$)).
4 Otherwise, the function returns the string returned by $name(*glvn*).

7.1.6.15 $order

$o[**rder**]**(***glvn***)**

This form returns a value which is a subscript according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a function, *CO*, which is used for definitional purposes only, to establish the collating sequence.

CO(s,*t*) is defined, for strings *s* and *t*, as follows:

When *t* follows *s* in the ordering sequence or if *s* is the empty string, *CO*(*s*,*t*) returns *t*.

Otherwise, *CO*(*s*,*t*) returns *s*.

The ordering sequence is defined using the *collation algorithm* determined as follows:

a If **$order** refers to a *ssvn*, then the algorithm is determined by the value of **^$system($system,"COLLATE")**; if that node does not exist, then the value of **$get(^$character(^$system($system,"CHARACTER"),"COLLATE"))**is used.
b If **$order** refers to a *gvn* with name *global* then the algorithm is determined by the value of **$get(^$global("*global*","COLLATE"))**; if that node does not exist, then the value of **$get(^$character(^$global("*global*","CHARACTER"),"COLLATE"))** is used.
c If **$order** does not refer to either of the above, then the algorithm is determined by the value of **$get(^$character(^$job($job,"CHARACTER"),"COLLATE"))**.
d If the resulting algorithm is the empty string, then the *collation algorithm* of the *charset* M (defined in Annex A) is used.

The collation value *order* of a string *subscript* using a collation algorithm *collate* may be determined by executing the expression "set *order*="_*collate*_"(*subscript*)". Two collation values are compared on a character-by-character basis using the **$ascii** values (i.e. equivalent to the follows (**]**) operator).

Only subscripted forms of *glvn* are permitted. Let *glvn* be of the form *NAME*(s_1,s_2,…,s_n)

where s_n may be the empty string. Let *A* be the set of subscripts such that, *s* is in *A* if and only if:

a $CO(s_n,s)=s$ and

b `$data`$(NAME(s_1,s_2,\ldots,s_{n-1},s))$ is not zero.

Then `$order`$(NAME(s_1,s_2,\ldots,s_n))$ returns that value *t* in *A* such that $CO(t,s)=s$ for all *s* not equal to *t*; that is, all other subscripts in *A* which follow s_n also follow *t*.

If no such *t* exists, `$order` returns the empty string.

`$o[rder](`*glvn*`,`*expr*`)`

Let *S* be the value of *expr*. Then `$order(`*glvn*`,`*expr*`)` returns:

a If *S*=1, the function returns a result identical to that returned by `$order(`*glvn*`)`.

b If *S*=-1, the function returns a value which is a subscript, according to a subscript ordering sequence. This ordering sequence is specified below with the aid of a functions *CO* and *CP*, which are used for definitional purposes only, to establish the collating sequence.

$CO(s,t)$ is defined, for strings *s* and *t*, according to the *collation algorithm* of the specific *charset*.

$CP(s,t)$ is defined, for strings *s* and *t*, as follows:

When *t* follows *s* in the ordering sequence and *s* is not the empty string, $CP(s,t)$ returns *s*.

Otherwise, $CP(s,t)$ returns *t*.

The following cases define the ordering sequence for *CP*:

1 *CP*("",*t*)=*t*.

2 $CP(s,t)=t$ if $CO(s,t)=s$; otherwise, $CP(s,t)=s$.

Only subscripted forms of *glvn* are permitted. Let *glvn* be of the form $NAME(s_1,s_2,\ldots,s_n)$ where s_n may be the empty string. Let *A* be the set of subscripts such that, *s* is in *A* if and only if:

1 $CP(s_n,s)=s$ and

2 `$data`$(NAME(s_1,s_2,\ldots,s_{n-1},s))$ is not zero.

Then `$order`$(NAME(s_1,s_2,\ldots,s_n),-1)$ returns that value *t* in *A* such that $CP(t,s)=t$ for all *s* not equal to *t*; that is, all other subscripts in *A* which precede *s* also precede *t*.

If no such *t* exists, `$order`$(NAME(s_1,s_2,\ldots,s_n),-1)$ returns the empty string.

c Values of *S* other than 1 and -1 are reserved for future extensions of the `$order` function.

7.1.6.16 `$piece`

`$p[iece](`$expr_1$`,`$expr_2$`)`

This form is defined here with the aid of a function, NF, which is used for definitional purposes only, called *find the position number following the* `m`*th occurrence*.

`NF(s,d,m)` is defined, for strings `s`, `d`, and integer `m`, as follows:

When `d` is the empty string, the result is zero.

When `m'>0`, the result is zero.

When `d` is not a substring of `s`, i.e., when `$find(s,d)=0`, then the result is `$length(s)+$length(d)+1`.

Otherwise, `NF(s,d,1)=$find(s,d)`.

For `m>1`, `NF(s,d,m)=NF($extract(s,$find(s,d),$length(s)),d,m-1)+$find(s,d)-1`.

That is, `NF` extends **`$find`** to give the position number of the character to the right of the `m`th occurrence of the string `d` in `s`.

Let `s` be the value of *expr*$_1$, and let `d` be the value of *expr*$_2$. **`$piece(s,d)`** returns the substring of `s` bounded on the right but not including the first (leftmost) occurrence of `d`.

`$piece(s,d)``=$extract(s,0,NF(s,d,1)-$length(d)-1)`

`$p[iece](`*expr*$_1$**`,`***expr*$_2$**`,`***intexpr***`)`**

Let `m` be the integer value of *intexpr*. **`$piece(s,d,m)`** returns the substring of `s` bounded by but not including the `m-1`th and the `m`th occurrence of `d`.

`$piece(s,d,m)``=$extract(s,NF(s,d,m−1),NF(s,d,m)-$length(d)-1)`

`$p[iece](`*expr*$_1$**`,`***expr*$_2$**`,`***intexpr*$_1$**`,`***intexpr*$_2$**`)`**

Let `m` be the integer value of *intexpr*$_1$. Let `n` be the integer value of *intexpr*$_2$. **`$piece(s,d,m,n)`** returns the substring of `s` bounded on the left but not including the `m-1`th occurrence of `d` in `s`, and bounded on the right but not including the `n`th occurrence of `d` in `s`.

`$piece(s,d,m,n)``=$extract(s,NF(s,d,m-1),NF(s,d,n)-$length(d)-1)`

Note that **`$piece(s,d,m,m)`**`=$piece(s,d,m)`, and that **`$piece(s,d,1)`**`=$piece(s,d)`.

7.1.6.17 `$qlength`

`$ql[ength](`*namevalue***`)`**

See 7.1.5.12 for the definition of *namevalue*.

This form returns a value which is derived from *namevalue*. If *namevalue* has the form `NAME(`s_1`,`s_2`,…,`s_n`)`, considering `n` to be zero if there are no subscripts, then the function returns `n`.

Note that the *namevalue* is not “executed”, and will not affect the naked indicator, nor generate an error if the *namevalue* represents an undefined *glvn*. The naked indicator will only be affected by the last *gvn* reference (if any) executed while evaluating the argument.

7.1.6.18 `$qsubscript`

`$qs[ubscript](`*namevalue***`,`***intexpr***`)`**

This form returns a value which is derived from *namevalue*. If *namevalue* has the form `NAME(`s_1`,`s_2`,…,`s_n`)`, considering *n* to be zero if there are no subscripts, and *m* is the value of intexpr, then **`$qsubscript(`***namevalue***`,`***intexpr***`)`** is defined as follows:

a Values of *m* less than `-1` are reserved for possible future use by extension of the standard.
b If *m*`=-1`, the result is the *environment* if *namevalue* includes an *environment*; otherwise the empty string.
c If *m*`=0`, the result is *NAME* without an *environment* even if one is present.
d If *m*`>n`, the result is the empty string.
e Otherwise, the result is the subscript value denoted by s_m.

Note that the *namevalue* is not "executed", and will not affect the naked indicator, nor generate an error if the *namevalue* represents an undefined *glvn*. The arguments are evaluated in left to right order, and the naked indicator will only be affected by the last *gvn* reference (if any) executed while evaluating them.

7.1.6.19 `$query`

`$q[uery](`*glvn***`)`**

Follow these steps:

a Let *glvn* be a variable reference of the form *Name*`(`s_1`,`s_2`,…,`s_q`)` where s_q may be the empty string. If *glvn* is unsubscripted, initialize *V* to the form *Name*`("")`; otherwise, initialize *V* to *glvn*.
b If the last subscript of *V* is empty, go to step *e*.
c If `$data(`*V*`)\10=1`, append the subscript `""` to *V*, i.e., *V* is *Name*`(`s_1`,`s_2`,…,`s_q`,"")`.
d If *V* has no subscripts, return `""`.
e Let *s*`=$order(`*V*`)`.
f If *s*`=""`, truncate the last subscript off *V*, go to step *d*.
g If *s*`'=""`, replace the last subscript in *V* with *s*.
h If `$data(`*V*`)#2=1`, return *V* formatted as a *namevalue*.
i Go to step *c*.

`$q[uery](`*glvn***`,`***expr***`)`**

Let *S* be the value of *expr*. Then **`$query(`***glvn***`,`***expr***`)`** returns:

1 If *S*`=1`, the function returns a result identical to that returned by `$query(`*glvn*`)`.
2 if *S*`=-1`, the function returns a value which is either the empty string (`""`) or a *namevalue* according to the following steps:

a Let *glvn* be a variable reference of the form *Name*(s_1,s_2,…,s_q) where s_q may be the empty string. If *glvn* is unsubscripted, initialize *V* to the form *Name*(""); otherwise, initialize *V* to *glvn*.

b If the last subscript of *V* is empty, go to step *e*.

c If `$data(`*V*`)\10=1`, append the subscript `""` to *V*, i.e. *V* is *Name*(s_1,s_2,…,s_q,"").

d If *V* has no subscripts, return `""`.

e Let *s*`=$order(`*V*`,-1)`.

f If *s*`=""`, truncate the last subscript of *V*, go to step *j*.

g If *s*`'=""`, replace the last subscript of *V* with *s*.

h If `$data(`*V*`)#2=1`, return *V* formatted as *namevalue*.

i Go to step *c*.

j If `$data(`*V*`)#2=1`, return *V* formatted as *namevalue*.

k Go to step *d*.

3 Values of *S* other than `1` or `-1` are reserved for future extension of the **`$query`** function.

If the value of **`$query(`***glvn*[**`,`***expr*]**`)`** is not the empty string and *glvn* includes an *environment*, then the *namevalue* shall include the *environment*; otherwise the *namevalue* shall not include an *environment*.

If the argument of **`$query`** is a *gvn*, the naked indicator will become undefined and the value of **`$reference`** will become equal to the empty string.

7.1.6.20 `$random`

`$r``[`**`andom`**`]`**`(`***intexpr***`)`**

This form returns a random or pseudo-random integer uniformly distributed in the closed interval `[0,`*intexpr*`-1]`. If the value of *intexpr* is less than `1`, an error condition occurs with *ecode*=`"`**`M3`**`"`.

7.1.6.21 `$reverse`

`$re``[`**`verse`**`]`**`(`***expr***`)`**

See Clause 7 for the definition of *expr*.

This form returns a string whose characters are reversed in order compared to *expr*.

`$reverse(`*expr***`)`** is computationally equivalent to `$$rev(`*expr*`)` which is defined by the following code:

```
rev(E) quit $select(E="":"",1:$$REV($extract(E,2,$length(E)))_$extract(E,1))
```

7.1.6.22 `$select`

`$s[elect](`*list*{*tvexpr*`:`*expr*}`)`

This form returns the value of the leftmost *expr* whose corresponding *tvexpr* is true. The process of evaluation consists of evaluating the *tvexpr*s, one at a time in left-to-right order, until the first one is found whose value is true. The *expr* corresponding to this *tvexpr* (and no other) is evaluated and this value is made the value of `$select`. An error condition occurs, with *ecode*=`"M4"`, if all *tvexpr*s are false. Since only one *expr* is evaluated at any invocation of `$select`, that is the only *expr* which must have a defined value.

7.1.6.23 `$stack`

`$st[ack](`*intexpr*`)`

This form returns a string as follows:

a If *intexpr* is `-1`
  1 if `$ecode` is not empty, returns the highest value where `$stack(`*intexpr*`)` can return non-empty results.
  2 if `$ecode` is empty, returns `$stack`.
b If *intexpr* is `0` (zero), returns an implementation specific value indicating how this process was started.
c If *intexpr* is greater than `0` (zero) and less than or equal to `$stack` indicates how this level of the *process-stack* was created:
  1 If due to a *command*, the *commandword* fully spelled out and in uppercase.
  2 if due to an *exfunc* or *exvar*, the string `"$$"`.
  3 if due to an error, the *ecode* representing the error that created the result returned by `$stack(`*intexpr*`)`.
d If *intexpr* is greater than `$stack`, returns an empty string.

Values of *intexpr* less than `-1` are reserved for future extensions of the `$stack` function.

`$st[ack](`*intexpr*`,`*stackcodeexpr*`)`

stackcodeexpr `::=` ∈{*expr*→*stackcode*}

stackcode `::=`
`PLACE`
`MCODE`
`ECODE`

This form returns information about the action that created the level of the *process-stack* identified by *intexpr* as follows:

intexpr a Values of *intexpr*`<0` are reserved.
b Values of *intexpr*>**`$stack`** return the empty string.

Stackcode **Returned String**

`ECODE` the list of any *ecode*s added at the level of the *process-stack* identified by *intexpr*.

`MCODE` the value (in the case of an **`xecute`**) or the *line* for the location identified by **`$stack(`***intexpr*`,"PLACE"`**`)`**. If the *line* is not available, an empty string is returned.

`PLACE` the location of a *command* at the *intexpr* level of the *process-stack* as follows:

a if *intexpr* is not equal to **`$stack`** and **`$stack(`***intexpr*`,"ECODE"`**`)`** would return the empty string, the last *command* executed.
b if *intexpr* is equal to **`$stack`** and **`$stack(`***intexpr*`,"ECODE"`**`)`** would return the empty string, the currently executing *command*.
c if **`$stack(`***intexpr*`,"ECODE"`**`)`** would return a non-empty string, the last *command* to start execution while **`$stack(`***intexpr*`,"ECODE"`**`)`** would have returned the empty string.

The location is in the form:

place *sp* **`+`***eoffset*

place `::=` `[`*label*`]` `[`**`+`***intlit*`]` `[`**`^`***vb* *environment* *vb* *routinename*`]`
`@`

eoffset `::=` *intlit*

In *place*, the first case is used to identify the *line* being executed at the time of creation of this level of the *process-stack*. The second case (**`@`**) shows the point of execution occurring in an **`xecute`**.

Eoffset is an offset into the code or data identified by *place* at which the error occurred. The value might point to the first or last character of a "token" just before or just after a "token", or even to the command or line in which the error occurred. Implementors should provide as accurate a value for *eoffset* as practical.

All values of *stackcode* beginning with the letter **`z`** are reserved for the implementation. All other values of *stackcode* are reserved for future extensions of the **`$stack`** function. *stackcode*s differing only in the use of corresponding upper and lower case letters are equivalent.

7.1.6.24 **`$text`**

`$t``[`**`ext`**`]`**`(`***textarg***`)`**

		`+`*intexpr*`[^`*routineref*`]`
textarg	`::=`	*entryref*
		`@`∈{*expratom* → *textarg*}

This form returns a string whose value is the contents of the *line* specified by the argument. Specifically, the entire *line*, with *eol* deleted, is returned.

If the argument of `$text` is an *entryref*, the *line* denoted by the *entryref* is specified. If *entryref* does not contain *dlabel* then the *line* denoted is the first *line* of the routine. If the argument is of the form `+`*intexpr*`[^`*routineref* `]`, two cases are defined. If the value of *intexpr* is greater than `0`, the *intexpr*th *line* of the *routine* is specified; if the value of *intexpr* is equal to `0`, the *routinename* of the *routine* is specified. An error condition occurs, with *ecode*=`"M5"`, if the value of *intexpr* is less than `0`. In all cases, if no *routine* is explicitly specified, the currently-executing *routine* is used.

If no such *line* as that specified by the argument exists, an empty string is returned. If the *line* specification is ambiguous, the results are not defined.

If a Character Set Profile input-transform is in effect, then the string is modified in accordance with the transform.

7.1.6.25 `$translate`

`$tr[anslate](`$expr_1$`,`$expr_2$`)`

Let *s* be the value of $expr_1$, `$translate(`$expr_1$`,`$expr_2$`)` returns an edited form of `s` in which all characters in `s` which are found in $expr_2$ are removed.

`$tr[anslate](`$expr_1$`,`$expr_2$`,`$expr_3$`)`

Let `s` be the value of $expr_1$, `$translate(`$expr_1$`,`$expr_2$`,`$expr_3$`)` returns an edited form of `s` in which all characters in `s` which are found in $expr_2$ are replaced by the positionally corresponding character in $expr_3$. If a character in `s` appears more than once in $expr_2$ the first (leftmost) occurrence is used to positionally locate the translation.

Translation is performed once for each character in `s`. Characters which are in `s` that are not in $expr_2$ remain unchanged. Characters in $expr_2$ which have no corresponding character in $expr_3$ are deleted from `s` (this is the case when $expr_3$ is shorter than $expr_2$).

Note: If the value of $expr_2$ is the empty string, no translation is performed and `s` is returned unchanged.

7.1.6.26 `$type`

`$ty[pe](`*expratom*`)`

This form returns the string value **"MVAL"** if the value of *expratom* is not a value of data type *oref*. Otherwise **$type** returns the string value **"OBJECT"**.

7.1.6.27 $view

$v[iew](unspecified**)**

makes available to the implementor a call for examining machine-dependent information. It is to be understood that routines containing occurrences of **$view** may not be portable.

7.1.6.28 $z

$z[unspecified]**(**unspecified**)**

is the initial letter reserved for defining non-standard intrinsic functions. This requirement permits the unused function names to be reserved for future use.

7.1.7 Library

7.1.7.1 Library definitions

A *library* consists of a set of *libraryelements*—functions and data which are accessed from Mumps and which have unique names within the *library*. The access method for each *libraryelement* is the external calling syntax, which normally has no side-effects.

A *library* is defined as being either mandatory or optional. *Library* names starting with a **z** are reserved for implementors. *Library* names starting with a **y** are reserved for users. All other unused *library* names are reserved for future use.

The Mumps Standard Library is the set of *library* definitions in this *Standard*.

The following *librarys* are defined:

7.1.7.1.1 · Mandatory libraries

CHARACTER

MATH

STRING

7.1.7.1.2 · Optional libraries

None defined at this time.

7.1.7.2 Library element definitions

The definition of a *libraryelement* states which *library* the element belongs to, return value type and full specification.

Libraryelement names starting with a **z** are reserved for implementors. *Libraryelement* names starting with a **y** are reserved for users. All other unused *libraryelement* names are reserved for future use.

A *libraryelement* definition is of the form:

```
libraryelementdef ::= libraryelement^library libraryresult [(list{libraryparam})]
     libraryparam ::= [.]name[:[libdatatype][:libraryopt]]
    libraryresult ::= [:libdatatype]
                      | BOOLEAN          |
                      | COMPLEX          |
                      | INTEGER          |
                      | MATRIX           |
      libdatatype ::= | NAME             |
                      | REAL             |
                      | STRING           |
                      | Z[unspecified]   |
                      | M |
       libraryopt ::= | O |
```

If a *libraryparam* starts with a period then this parameter is passed by reference.

Z is the initial letter reserved for implementation specific *libdatatype*s. All other values for *libdatatype*s are reserved for future expansion of the standard.

Input and output values to *libraryelement*s undergo the appropriate data interpretation below:

a for **BOOLEAN** see 7.1.5.7 Truth-value interpretation.
b for **COMPLEX** see 7.1.7.2.1 Complex interpretation of data.
c for **INTEGER** see 7.1.5.6 Integer interpretation.
d for **MATRIX** see 7.1.7.2.2 Matrix interpretation of data.
e for **NAME** see 7.1.5.12 Name value *namevalue*.
f for **REAL** see 7.1.5.5 Numeric interpretation of data.
g **STRING** is a string made up of any characters and not constrained in format.

If no *libdatatype* is specified for a *libraryparam* or *libraryresult* then the *libdatatype* defaults to **STRING**.

If no *libraryopt* is specified then the *libraryparam* is **M** (mandatory). A *libraryopt* of **O** specifies that the *libraryparam* is optional.

The term **STANDARD** in the specification of the domain of a *libraryelement* means that all of its *libraryparam*s can assume any valid values of their respective *libdatatype*s. Similarly, **STANDARD** in the specification of the range means that the *libraryresult* and all of its output *libraryparam*s can assume any valid values of their respective *libdatatype*s.

7.1.7.2.1 · Complex interpretation of data

COMPLEX numbers are represented as strings of the format **REAL_"%"_REAL** (that is, two **REAL** numbers separated by the **%** character). Any string has a value when interpreted as a complex number. The real part of the complex number is the numeric interpretation of the first "`%`" piece and the imaginary part is the numeric interpretation of the second "`%`" piece. The canonic representation of a complex number is a string created by concatenating the canonic numerical representation of the real part, a percent sign, and the canonic numerical representation of the imaginary part.

7.1.7.2.2 · Matrix interpretation of data

In the specification of matrix functions, the notation ***A***`[`*`R`*`,`*`C`*`]` means that matrix ***A*** contains *`R`* rows of *`C`* columns.

Any matrix may be sparse in the sense that not all elements *`name`*`(`*`row`*`,`*`col`*`)` with `1≤`*`row`*`≤`*`number of rows`* and `1≤`*`col`*`≤`*`number of columns`* need to have a defined value. Only those elements that have a defined value in an input matrix will cause a result to be stored into an output matrix. Elements will be removed from an output matrix when the corresponding element in the relevant input matrix or matrices does not have a defined value; descendants of such elements will not be affected.

In the specification of the various functions below, the typification **`MATRIX`** identifies a parameter as a sparse array, of which elements with two (integer valued) subscripts will be accessed. There is no prohibition that these arrays contain other descendants. The functions in the matrix library are not affected by any such descendants, nor will any such descendants be affected by the functions in the matrix library.

When, in the definition of a matrix library element, a parameter is preceded by a period, this indicates that the parameter in question is intended to be passed by reference.

7.1.7.3 Availability of library elements

An implementation of Mumps shall

a provide the mandatory *library*s defined in this *Standard*
and

b provide a means by which replacement difinitions in *routine*s of *libraryelement*s can be installed so that a *routine* can access them as if they were part of the implementation. An implementation may additionally provide a means by which non-Mumps code can be installed to implement *libraryelement*s.

An implementation may also provide a means by which specific *library*s or *libraryelement*s of the Mumps Standard Library are only optionally installed.

7.1.7.4 `CHARACTER` library elements

1 `$%COLLATE^CHARACTER` string collation value
2 `$%COMPARE^CHARACTER` String comparison

7.1.7.4.1 · `$%COLLATE^CHARACTER`

`COLLATE^CHARACTER(`*`A`*`,`*`CHARMOD`*`::0)`

This function returns the collation value of a string according to the specification of the collation algorithm.

If *CHARMOD* is a Character Set Profile then the collation algorithm used is that specified in `^$character` for the profile.

If *CHARMOD* is a global name then the collation algorithm used is that specified in `^$global` for that name.

If *CHARMOD* is not specified, or the node specified above does not exist, then the collation algorithm used is the default process collating algorithm.

Domain: *CHARMOD* is either a Character Set Profile specification in the form *charset* or a global name specification in the form ^*name*.

Range: standard

Side effects: none

7.1.7.4.2 · $%COMPARE^CHARACTER

`COMPARE^CHARACTER:INTEGER(A,B,CHARMOD::0)`

This function compares two strings according to the specification of the collation algorithm.

If *CHARMOD* is a Character Set Profile then the two strings are compared using the collation algorithm specified in `^$CHARACTER` for the profile.

If *CHARMOD* is a global name then the two strings are compared using the collation algorithm specified in `^$GLOBAL` for that name.

If *CHARMOD* is not specified, or the node specified above does not exist, then the two strings are compared using the default process collating algorithm.

Domain: *CHARMOD* is either a Character Set Profile specification in the form *charset* or a global name specification in the form ^*name*.

Range:

- -1 = *A* compares before *B*
- 0 = *A* compares the same as *B*
- 1 = *A* compares after *B*

Side effects: none

7.1.7.5 MATH library elements

1	$%ABS^MATH	absolute value
2	$%ARCCOS^MATH	inverse trigonometric cosine
3	$%ARCCOSH^MATH	inverse hyperbolic cosine
4	$%ARCCOT^MATH	inverse trigonometric cotangent
5	$%ARCCOTH^MATH	inverse hyperbolic cotangent
6	$%ARCCSC^MATH	inverse trigonometric cosecant
7	$%ARCSEC^MATH	inverse trigonometric secant
8	$%ARCSIN^MATH	inverse trigonometric sine
9	$%ARCSINH^MATH	inverse hyperbolic sine
10	$%ARCTAN^MATH	inverse trigonometric tangent
11	$%ARCTANH^MATH	inverse hyperbolic tangent

12	`$%CABS^MATH`	absolute value of a complex number
13	`$%CADD^MATH`	add complex numbers
14	`$%CCOS^MATH`	trigonometric cosine of a complex number
15	`$%CDIV^MATH`	divide complex numbers
16	`$%CEXP^MATH`	exponentiate (raise e to the power of a complex number)
17	`$%CLOG^MATH`	Napierian logarithm of complex number ($\log_e$ of a complex number)
18	`$%CMUL^MATH`	multiply complex numbers
19	`$%COMPLEX^MATH`	convert number to complex number
20	`$%CONJUG^MATH`	conjugate of complex number
21	`$%COS^MATH`	trigonometric cosine
22	`$%COSH^MATH`	hyperbolic cosine
23	`$%COT^MATH`	trigonometric cotangent
24	`$%COTH^MATH`	hyperbolic cotangent
25	`$%CPOWER^MATH`	raise complex number to the power of another complex number
26	`$%CSC^MATH`	trigonometric cosecant
27	`$%CSCH^MATH`	hyperbolic cosecant
28	`$%CSIN^MATH`	trigonometric sine of a complex number
29	`$%CSUB^MATH`	subtract complex numbers
30	`$%DECDMS^MATH`	convert degrees to ° ' " notation
31	`$%DEGRAD^MATH`	convert degrees to radians
32	`$%DMSDEG^MATH`	convert degrees from ° ' " notation
33	`$%E^MATH`	Euler's number (*e*)
34	`$%EXP^MATH`	exponentiate (raise *e* to the power of a number)
35	`$%LOG^MATH`	Napierian logarithm ($\log_e$)
36	`$%LOG10^MATH`	Briggsian logarithm ($\log_{10}$)
37	`$%MTXADD^MATH`	matrix addition
38	`$%MTXCOF^MATH`	matrix cofactor
39	`$%MTXCOPY^MATH`	matrix copy
40	`$%MTXDET^MATH`	matrix determinant
41	`$%MTXEQU^MATH`	solve matrix equation
42	`$%MTXINV^MATH`	matrix inversion
43	`$%MTXMUL^MATH`	matrix multiplication (matrix × matrix)
44	`$%MTXSCA^MATH`	matrix multiplication (matrix × scalar)
45	`$%MTXSUB^MATH`	matrix subtraction
46	`$%MTXTRP^MATH`	matrix transpose
47	`$%MTXUNIT^MATH`	unit matrix
48	`$%PI^MATH`	pi (π)
49	`$%RADDEG^MATH`	convert radians to degrees
50	`$%SEC^MATH`	trigonometric secant
51	`$%SECH^MATH`	hyperbolic secant
52	`$%SIGN^MATH`	transfer sign
53	`$%SIN^MATH`	trigonometric sine

54	`$%SINH^MATH`	hyperbolic sine
55	`$%SQRT^MATH`	square root
56	`$%TAN^MATH`	trigonometric tangent
57	`$%TANH^MATH`	hyperbolic tangent

7.1.7.5.1 · `$%ABS^MATH`

ABS^MATH:REAL**(***X*:REAL**)**

This function returns the absolute value of its parameter.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.2 · `$%ARCCOS^MATH`

ARCCOS^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the trigonometric arccosine in radians of *X*. The number of significant digits in the arccosine is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: -1≤*X*≤+1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: 0<$%ARCCOS~MATH(*X*)<π

Side effects: none

7.1.7.5.3 · `$%ARCCOSH^MATH`

ARCCOSH^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the hyperbolic arccosine in radians of *X*. The number of significant digits in the hyperbolic arccosine is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: *X*≥1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: $%ARCCOSH^MATH(*X*)≥0

Side effects: none

7.1.7.5.4 · `$%ARCCOT^MATH`

ARCCOT^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the trigonometric arccotangent in radians of *X*. The number of significant digits in the arccotangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range: 0<$%ARCCOT^MATH(*X*)<π

Side effects: none

7.1.7.5.5 · `$%ARCCOTH^MATH`

`ARCCOTH^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the hyperbolic arccotangent in radians of *X*. The number of significant digits in the hyperbolic arccotangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: *X*<-1 or *X*>1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range:

`$%ARCCOTH^MATH(`*`X`*`)<0` when *X*≤-1 and

`$%ARCCOTH^MATH(`*`X`*`)>0` when *X*≥1

Side effects: none

7.1.7.5.6 · `$%ARCCSC^MATH`

`ARCCSC^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric arccosecant in radians of *X*. The number of significant digits in the arccosecant is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: -1≤*X*≤+1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: `0≤$%ARCCSC^MATH(`*`X`*`)≤π`

Side effects: none

7.1.7.5.7 · `$%ARCSEC^MATH`

`ARCSEC^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric arcsecant in radians of *X*. The number of significant digits in the arcsecant is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: -1≤*X*≤+1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: `0≤$%ARCSEC^MATH(`*`X`*`)≤π`

Side effects: none

7.1.7.5.8 · `$%ARCSIN^MATH`

`ARCSIN^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric arcsine in radians of *X*. The number of significant digits in the arcsine is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: -1≤*X*≤1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: `-π/2≤$%ARCSIN^MATH(`*`X`*`)≤π/2`

Side effects: none

7.1.7.5.9 · `$%ARCSINH^MATH`

ARCSINH^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the hyperbolic arcsine in radians of *X*. The number of significant digits in the hyperbolic arcsine is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.10 · `$%ARCTAN^MATH`

ARCTAN^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the trigonometric arctangent in radians of *X*. The number of significant digits in the arctangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range:

|$%ARCTAN^MATH(*Y*)|≤π/2

0≤$%ARCTAN^MATH(*Y*)≤π when *Y*≥0

-π≤$%ARCTAN^MATH(*Y*)≤0 when *Y*≤0

Side effects: none

7.1.7.5.11 · `$%ARCTANH^MATH`

ARCTANH^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the hyperbolic arctangent in radians of *X*. The number of significant digits in the hyperbolic arctangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: -1<*X*<1. When the value of parameter *X* is out of range, an error will result with *ecode*="**M28**".

Range: standard

Side effects: none

7.1.7.5.12 · `$%CABS^MATH`

CABS^MATH:REAL**(***Z*:COMPLEX**)**

The function returns the absolute value of the complex number *Z*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.13 · `$%CADD^MATH`

CADD^MATH:COMPLEX**(***X*:COMPLEX**,***Y*:COMPLEX**)**

The function returns the sum of *X*+*Y*, where *X* and *Y* are complex numbers.

Domain: standard

Range: standard
Side effects: none

7.1.7.5.14 · `$%CCOS^MATH`

`CCOS^MATH``:COMPLEX(`*`Z`*`:COMPEX,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric cosine `cos(`*`Z`*`)` of the angle *`Z`* in radians. *`Z`* is interpreted as a complex number. The number of significant digits in the complex cosine is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard
Range: `-1≤$%COS^MATH(`*`X`*`)≤1`
Side effects: none

7.1.7.5.15 · `$%CDIV^MATH`

`CDIV^MATH``:COMPLEX(`*`X`*`:COMPLEX,`*`Y`*`:COMPLEX)`

The function returns the value *`X`*`/`*`Y`*, where *`X`* and *`Y`* are complex numbers.

Domain: standard
Range: standard. If the complex numeric interpretation of *`Y`* is equal to `"0%0"`, an error condition occurs with *ecode*=`"`**`M9`**`"`.
Side effects: none

7.1.7.5.16 · `$%CEXP^MATH`

`CEXP^MATH``:COMPLEX(`*`Z`*`:COMPLEX,`*`PREC`*`:INTEGER:0)`

The function returns the value of `e` raised to the power of the complex number *`Z`*. The number of significant digits in the complex exponent is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard
Range: standard
Side effects: none

7.1.7.5.17 · `$%CLOG^MATH`

`CLOG^MATH``:COMPLEX(`*`Z`*`:COMPLEX,`*`PREC`*`:INTEGER:0)`

The function returns the Napierian logarithm of the complex number *`Z`*. The number of significant digits in the complex logarithm is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: if `Im(`*`Z`*`=0)`, then `Re(`*`Z`*`>0)`
Range:
`Re($%CLOG^MATH(`*`Z`*`))` can be any number
`-π≤Im($%CLOG^MATH(`*`Z`*`))≤π`
Side effects: none

7.1.7.5.18 · `$%CMUL^MATH`

`CMUL^MATH``:COMPLEX(`*`X`*`:COMPLEX,`*`Y`*`:COMPLEX)`

The function returns the value of *`X*Y`*, where *`X`* and *`Y`* are complex numbers.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.19 · `$%COMPLEX^MATH`

`COMPLEX^MATH``:COMPLEX(`*`X`*`:REAL)`

The function returns the complex representation of the number specified in *`X`*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.20 · `$%CONJUG^MATH`

`CONJUG^MATH``:COMPLEX(`*`Z`*`:COMPLEX)`

The function returns the value of the conjugate of the complex number *`Z`*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.21 · `$%COS^MATH`

`COS^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric cosine of the angle *`X`* in radians. The number of significant digits in the cosine is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `-1≤$%COS^MATH(`*`X`*`)≤1`

Side effects: none

7.1.7.5.22 · `$%COSH^MATH`

`COSH^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the hyperbolic cosine of the angle *`X`* in radians. The number of significant digits in the cosine is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `$%COSH^MATH(`*`X`*`)≥1`

Side effects: none

7.1.7.5.23 · `$%COT^MATH`

`COT^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric cotangent of the angle *`X`* in radians.

The number of significant digits in the cotangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range: `-∞≤$%COT^MATH(X)≤∞`

Side effects: none

7.1.7.5.24 · $%COTH^MATH

COTH^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the hyperbolic cotangent of the angle *X* in radians. The number of significant digits in the hyperbolic cotangent is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range:

`$%COTH^MATH(X)<-1` when `X<0` and

`$%COTH^MATH(X)>1` when `X>0`

Side effects: none

7.1.7.5.25 · $%CPOWER^MATH

CPOWER^MATH:COMPLEX**(***Z*:COMPLEX**,***X*:COMPLEX**,***PREC*:INTEGER:0**)**

The function returns the value of the complex number *Z* raised to the power of the complex number *X*. The number of significant digits in the complex power is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard. If both *Z* and *X* are equal to zero (`0` or `"0%0"`), an error will occur with *ecode*=**"M28"**.

Range: standard

Side effects: none

7.1.7.5.26 · $%CSC^MATH

CSC^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the trigonometric cosecant of the angle *X* in radians. The number of significant digits in the cosecant is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range:

`$%CSC^MATH(X)≤-1` or

`$%CSC^MATH(X)≥1`

Side effects: none

7.1.7.5.27 · $%CSCH^MATH

CSCH^MATH:REAL**(***X*:REAL**,***PREC*:INTEGER:0**)**

The function returns the value of the hyperbolic cosecant of the angle *X* in radians. The number of significant digits in the hyperbolic cosecant is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.28 · $%CSIN^MATH

CSIN^MATH:COMPLEX**(***Z*:COMPLEX**,***PREC*:INTEGER:O**)**

The function returns the value of the trigonometric sine sin(*Z*) of the angle *Z* in radians. *Z* is interpreted as a complex number. The number of significant digits in the complex sine is specified by the optional parameter *PREC*. If not specified, a default value of 11 digits is assumed for *PREC*.

Domain: standard

Range:

-1≤Re($%CSIN^MATH(*Z*))≤1

-1≤Im($%CSIN^MATH(*Z*))≤1

Side effects: none

7.1.7.5.29 · $%CSUB^MATH

CSUB^MATH:COMPLEX**(***X*:COMPLEX**,***Y*:COMPLEX**)**

The function returns the value of *X*-*Y*, where *X* and *Y* are complex numbers.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.30 · $%DECDMS^MATH

DECDMS^MATH:STRING**(***X*:REAL**,***PREC*:INTEGER:O**)**

This function returns a string, containing the ° ' " notation for the angle that is specified in *X* in degrees. Since the symbols for degrees, minutes, and seconds are not in the ASCII set, the fields in the result-value are separated by colons (**":"**).

The (optional) parameter *PREC* specifies the precision to which *X* is rounded before the conversion takes place. If not specified, a default value of 5 digits is assumed for *PREC*.

Domain: standard

Range: The value of $%DECDMS^MATH(*X*) consists of three **":"** separated parts. The value of the first part is an integer in the range [0,359]; the value of the second part is an integer in the range [0,59], the value of the third part is a real number in the range [0,60].

Side effects: none

7.1.7.5.31 · $%DEGRAD^MATH

DEGRAD^MATH:REAL**(***X*:REAL**)**

The function returns the value in radians that is equal to the angle specified in *X* in degrees. A full circle is 2π radians, or 360 degrees.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.32 · `$%DMSDEC^MATH`

`DMSDEC^MATH``:REAL(`*`X`*`:REAL)`

The function returns the value in degrees that is equal to the angle specified in *`X`* in ° ' " notation.

Domain: The value of *`X`* consists of three `":"` separated parts. The value of the first part is an integer in the range `[0,+359]`; the value of the second part is an integer in the range `[0,59]`, the value of the third part is a rational number in the range `[0,60]`.

Any further `":"` separated parts in the value of *`X`* are ignored.

Range: standard

Side effects: none

7.1.7.5.33 · `$%E^MATH`

`E^MATH``:REAL()`

The function returns the value of Euler's number, approximated to at least 15 significant digits.

Domain: standard

Range: not applicable

Side effects: none

7.1.7.5.34 · `$%EXP^MATH`

`EXP^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of *e* to the power *`X`*. The exponentiation is approximated with as many significant digits as specified by the optional parameter *`PREC`*. If not specified, a default value of `11` is assumed for *`PREC`*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.35 · `$%LOG^MATH`

`LOG^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the Napierian logarithm of *`X`*. The number of significant digits in the logarithm is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: *`X`*`>0`. When the value of parameter *`X`* is out of range, an error will result with *ecode*=`"M28"`.

Range: standard

Side effects: none

7.1.7.5.36 · `$%LOG10^MATH`

`LOG10^MATH:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the Briggsian logarithm of *`X`*. The number of significant digits in the logarithm is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: *`X`*`>0`. When the value of parameter *`X`* is out of range, an error will result with *ecode*=`"`**`M28`**`"`.

Range: standard

Side effects: none

7.1.7.5.37 · `$%MTXADD^MATH`

`MTXADD^MATH``:BOOLEAN`**`(.`*`A`***`:MATRIX`**`,.`*`B`***`:MATRIX`**`,.`*`R`***`:MATRIX`**`,`***`ROWS`*`:INTEGER`**`,`***`COLS`*`:INTEGER`**`)`**

This function adds matrix ***`B`***`[`*`ROWS`*`,`*`COLS`*`]` to matrix ***`A`***`[`*`ROWS`*`,`*`COLS`*`]`, and stores the result into matrix ***`R`***`[`*`ROWS`*`,`*`COLS`*`]`. It is permissible that the actual parameter for matrix ***`R`*** is equal to either of the actual parameters for matrices ***`A`*** and ***`B`***. The return value is `1` if both matrices ***`A`*** and ***`B`*** exist, or `0` if there are no defined values in one or both of the matrices ***`A`*** and ***`B`***.

Domain: *`ROWS`*`>0`, *`COLS`*`>0`, other parameters are standard

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.38 · `$%MTXCOF^MATH`

`MTXCOF^MATH``:REAL`**`(.`*`A`***`:MATRIX`**`,`***`I`*`:INTEGER`**`,`***`K`*`:INTEGER`**`,`***`N`*`:INTEGER`**`)`**

This function computes the cofactor in matrix ***`A`***`[`*`N`*`,`*`N`*`]` for element ***`A`***`(`*`I`*`,`*`K`*`)`. The return value is the value of the cofactor.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.39 · `$%MTXCOPY^MATH`

`MTXCOPY^MATH``:BOOLEAN`**`(.`*`A`***`:MATRIX`**`,.`*`R`***`:MATRIX`**`,`***`ROWS`*`:INTEGER`**`,`***`COLS`*`:INTEGER`**`)`**

This function copies the matrix ***`A`***`[`*`ROWS`*`,`*`COLS`*`]` into the matrix ***`R`***`[`*`ROWS`*`,`*`COLS`*`]`. The return value is `1` if matrix ***`A`*** exists, or `0` if there are no defined values in the matrix ***`A`***.

Domain: *`ROWS`*`>0`, *`COLS`*`>0`, other parameters are standard

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.40 · `$%MTXDET^MATH`

`MTXDET^MATH``:REAL`**`(.`*`A`***`:MATRIX`**`,`***`N`*`:INTEGER`**`)`**

This function computes the determinant of matrix ***`A`***`[`*`N`*`,`*`N`*`]`. The return value is the value of the determinant, or (empty) if the determinant cannot be computed.

Domain: *`N`*`>0`, other parameters are standard

Range: standard, or an empty string when no determinant can be computed

Side effects: none

7.1.7.5.41 · `$%MTXEQU^MATH`

`MTXEQU^MATH:BOOLEAN(.A:MATRIX,.B:MATRIX,.R:MATRIX,N:INTEGER,M:INTEGER)`

This function solves the matrix-equation `A[M,M]*R[M,N]=B[M,N]`, with matrix `R[M,N]` being the unknown to be resolved. The return value is `1` if a solution to the equation can be computed, or `0` otherwise.

Domain: `M>0`, `N>0`, other parameters are standard

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.42 · `$%MTXINV^MATH`

`MTXINV^MATH:BOOLEAN(.A:MATRIX,.R:MATRIX,N:INTEGER)`

This function inverts matrix `A[N,N]` into matrix `R[N,N]`. It is permissible that the actual parameter for matrix **R** is equal to the actual parameter for matrix **A**. The return value is `1` if matrix **A** has been inverted into matrix **R**, or `0` if no inverse matrix can be computed.

Domain: `N>0`, other parameters are standard

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.43 · `$%MTXMUL^MATH`

`MTXMUL^MATH:BOOLEAN(A:MATRIX,.B:MATRIX,.R:MATRIX,M:INTEGER,L:INTEGER,N:INTEGER)`

This function multiplies matrix `A[M,L]` with matrix `B[L,N]`; the result is stored into matrix `R[M,N]`. The actual parameter for matrix **R** may not be equal to the actual parameter for matrix **A** or the actual parameter for matrix **B**. The return value is `1` if both matrices **A** and **B** exist, or `0` if there are no defined values in one or both of the matrices **A** and **B**.

Domain: `L>0`, `M>0`, `N>0`, other parameters are standard.

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.44 · `$%MTXSCA^MATH`

`MTXSCA^MATH:BOOLEAN(.A:MATRIX,.R:MATRIX,ROWS:INTEGER,COLS:INTEGER,S:REAL)`

This function multiplies scalar value *S* with matrix `A[ROWS,COLS]`, and stores the result into matrix `R[ROWS,COLS]`. It is permissible that the actual parameter for matrix **R** is equal to the actual parameters for matrix **A**. The return value is `1` if matrix **A** exists, or `0` if there are no defined values in the matrix **A**.

Domain: `ROWS>0`, `COLS>0`, other parameters are standard.

Range: standard

Side effects: elements of the result matrix may be redefined

7.1.7.5.45 · `$%MTXSUB^MATH`

`MTXSUB^MATH``:BOOLEAN(.`***`A`***`:MATRIX,.`***`B`***`:MATRIX,.`***`R`***`:MATRIX,`*`ROWS`*`:INTEGER,`*`COLS`*`:INTEGER)`

This function subtracts matrix ***B***[*ROWS*,*COLS*] from matrix ***A***[*ROWS*,*COLS*], and stores the result into matrix ***R***[*ROWS*,*COLS*]. It is permissible that the actual parameter for matrix ***R*** is equal to either of the actual parameters for matrices ***A*** and ***B***. The return value is `1` if both matrices ***A*** and ***B*** exist, or `0` if there are no defined values in one or both of the matrices ***A*** and ***B***.

Domain: *ROWS*>0, *COLS*>0, other parameters are standard
Range: standard
Side effects: elements of the result matrix may be redefined

7.1.7.5.46 · `$%MTXTRP^MATH`

`MTXTRP^MATH``:BOOLEAN(.`***`A`***`:MATRIX,.`***`R`***`:MATRIX,`*`M`*`:INTEGER,`*`N`*`:INTEGER)`

This function transposes matrix ***A***[*M*,*N*] into matrix ***R***[*N*,*M*]. It is permissible that the actual parameter for matrix ***R*** is equal to the actual parameter for matrix ***A***. The return value is `1` if matrix ***A*** exists, or `0` if there are no defined values in the matrix ***A***.

Domain: *M*>0, *N*>0, other parameters are standard
Range: standard
Side effects: elements of the result matrix may be redefined

7.1.7.5.47 · `$%MTXUNIT^MATH`

`MTXUNIT^MATH``:BOOLEAN(.`***`R`***`:MATRIX,`*`N`*`:INTEGER,`*`SPARSE`*`:BOOLEAN:0)`

This function creates matrix ***R***[*N*,*N*] as a unit matrix. The return value is `1` if a unit matrix can be created, `0` otherwise. If the value of the optional parameter *SPARSE* is `1` (true), a sparse unit matrix will be created, i.e. only the diagonal elements of the result matrix will be defined.

Domain: *N*>0
Range: standard
Side effects: elements of the result matrix may be redefined

7.1.7.5.48 · `$%PI^MATH`

`PI^MATH``:REAL()`

The function returns the value of π (pi), approximated to at least `15` significant digits.

Domain: not applicable
Range: standard
Side effects: none

7.1.7.5.49 · `$%RADDEG^MATH`

`RADDEG^MATH``:REAL(`*`X`*`:REAL)`

The function returns the value in degrees that is equal to the angle specified in *X* in radians. A full circle is 2π radians, or `360` degrees.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.50 · `$%SEC^MATH`

`SEC^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric secant of the angle *`X`* in radians. The number of significant digits in the secant is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range:

`$%SEC^MATH(`*`X`*`)≤-1` or

`$%SEC^MATH(`*`X`*`)≥1`

Side effects: none

7.1.7.5.51 · `$%SECH^MATH`

`SECH^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the hyperbolic secant of the angle *`X`* in radians. The number of significant digits in the hyperbolic secant is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `0<$%SECH^MATH(`*`X`*`)≤1`

Side effects: none

7.1.7.5.52 · `$%SIGN^MATH`

`SIGN^MATH``:REAL(`*`X`*`:REAL)`

The function returns `0`, `-1`, or `1`, depending on the value of *`X`*.

Domain: standard

Range:

`X<0→$%SIGN^MATH(`*`X`*`)=-1`

`X=0→$%SIGN^MATH(`*`X`*`)=0`

`X>0→$%SIGN^MATH(`*`X`*`)=1`

Side effects: none

7.1.7.5.53 · `$%SIN^MATH`

`SIN^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:0)`

The function returns the value of the trigonometric sine of the angle *`X`* in radians. The number of significant digits in the sine is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `-1≤$%SIN^MATH(`*`X`*`)≤1`

Side effects: none

7.1.7.5.54 · `$%SINH^MATH`

`SINH^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:O)`

The function returns the value of the hyperbolic sine of the angle *`X`* in radians. The number of significant digits in the hyperbolic sine is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: standard

Side effects: none

7.1.7.5.55 · `$%SQRT^MATH`

`SQRT^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:O)`

The function returns the square root of *`X`*. The number of significant digits in the square root is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: *`X`*`≥0`. When the value of parameter *`X`* is out of range, an error will result with *ecode*=`"`**`M28`**`"`.

Range: standard

Side effects: none

7.1.7.5.56 · `$%TAN^MATH`

`TAN^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:O)`

The function returns the value of the trigonometric tangent of the angle *`X`* in radians. The number of significant digits in the tangent is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `-∞≤$%TAN^MATH(`*`X`*`)≤∞`

Side effects: none

7.1.7.5.57 · `$%TANH^MATH`

`TANH^MATH``:REAL(`*`X`*`:REAL,`*`PREC`*`:INTEGER:O)`

The function returns the value of the hyperbolic tangent of the angle *`X`* in radians. The number of significant digits in the hyperbolic tangent is specified by the optional parameter *`PREC`*. If not specified, a default value of `11` digits is assumed for *`PREC`*.

Domain: standard

Range: `-1<$%TANH^MATH(`*`X`*`)≤1`

Side effects: none

7.1.7.6 `STRING` library elements

1 `$%CRC16^STRING` sixteen (16) bit cyclic redundancy code
2 `$%CRC32^STRING` thirty-two (32) bit Cyclic Redundancy Code
3 `$%CRCCCIT^STRING` sixteen (16) bit cyclic redundancy code

4	`$%LOWER^STRING`	lower case conversion
5	`$%PRODUCE^STRING`	string handling function; substring replacement
6	`$%REPLACE^STRING`	string handling function; substring replacement
7	`$%UPPER^STRING`	upper case conversion

7.1.7.6.1 · `$%CRC16^STRING`

`CRC16^STRING``:INTEGER(`*`S`*`:STRING,`*`SEED`*`:INTEGER:0)`

This function computes a Cyclic Redundancy Code of the 8-bit character string *S* using $X^{16}+X^{15}+X^{2}+1$ as the polynomial. The optional *SEED* parameter supplies an initial value, which allows running CRC calculations on multiple strings. If missing, a default value of zero is used. The message bytes are considered shifted in low order bit first and the return value shifted out low order bit first.

Domain:

S: standard

SEED: $0 \le SEED \le 2^{16}$, when *SEED* is outside its domain, an error condition occurs with *ecode*="**M28**"

Range: $0 \le result \le 2^{16}$

Side effects: none

7.1.7.6.2 · `$%CRC32^STRING`

`CRC32^STRING``:INTEGER(`*`S`*`:STRING,`*`SEED`*`:INTEGER:0)`

This function computes a Cyclic Redundancy Code of the 8-bit character string *S* using $X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^{8}+X^{7}+X^{5}+X^{4}+X^{2}+X+1$ as the polynomial. The optional *SEED* parameter supplies an initial value, which allows running CRC calculations on multiple strings. If missing, a default value of zero is used. The *SEED* is ones-complemented before being used, the message bytes are considered shifted in low order bit first and the return value is ones-complemented and shifted out low order bit first.

Domain:

S: standard

SEED: $0 \le SEED \le 2^{32}$, when *SEED* is outside its domain, an error condition occurs with *ecode*="**M28**"

Range: $0 \le result \le 2^{32}$

Side effects: none

7.1.7.6.3 · `$%CRCCCIT^STRING`

`CRCCCIT^STRING``:INTEGER(`*`S`*`:STRING,`*`SEED`*`:INTEGER:0)`

This function computes a Cyclic Redundancy Code of the 8-bit character string *S* using $X^{16}+X^{12}+X^{5}+1$ as the polynomial. The optional *SEED* parameter supplies an initial value, which allows running CRC calculations on multiple strings. If missing, a default value of 65535 (2^{16}-1) is used. The message bytes are considered shifted in high order bit first and the return value shifted out high order bit first.

Domain:

S: standard

SEED: 0≤*SEED*≤2^{16}, when *SEED* is outside its domain, an error condition occurs with *ecode*="**M28**"

Range: 0≤*result*≤2^{16}

Side effects: none

7.1.7.6.4 · $%FORMAT^STRING

FORMAT^STRING:STRING**(***IN*:STRING**,***FORMAT*:*fdirectives***)**

See 6.1 Routine head *routinehead* for the definition of *graphic* and 7 Expression *expr* for the definition of *expr*. *Sepchar* is a string which resolves to a single instance of a *graphic*.

```
fdirectives ::= @∈{expr → fspec[:fspec]...}
               | CS=curstr  | (note: currency specifier)
               | DC=dechar  | (note: decimal character)
               | EC=erchar  | (note: error character)
               | FL=filstr  | (note: fill string)
      fspec ::= | FM=fmask   | (note: format mask)
               | FO=fillon  | (note: fill on)
               | SC=sepchar | (note: separator character (s) left)
               | SR=sepchar | (note: separator character (s) right)
     string ::= @∈{expr → graphic...}
     curstr ::= @∈{expr → string}
     dechar ::= @∈{expr → graphic}
     erchar ::= @∈{expr → graphic}
     fillon ::= @∈{expr → tvalue}
     filstr ::= @∈{expr → string}
                          | c  | (note: currency, left justified)
                          | d  | (note: decimal, singular occurrence per mask)
                          | f  | (note: float either type at the end of the f run)
                          | l  | (note: left justified numeric)
                          | m  | (note: money, right justified)
      fmask ::= @∈{expr → | n  | (note: numeric)
                          | s  | (note: separator)
                          | x  | (note: spacer)
                          | -  | (note: display negative only if negative)
                          | +  | (note: display sign always)
                          | sp | (note: insert space for the spaces in the mask)
    sepchar ::= @∈{expr → string}
```

This function returns a formatted string. It will return *expr* unchanged. *Fmask* is the description of the field to be output. It provides a rich set of alternatives with enough undefined richness to provide future extension. Should a sign not be specified (+, 0, or -), the absolute value of *IN* is returned.

The *curstr* is a string of characters which represents the single local currency designator or the multiple character international reference. This string only occurs once in a *fmask* specification stream.

The *dechar* is the definition of the decimal character. It provides a set of rich possibilities. There is the case where the currency in Portugal is used as the decimal indicator. By defining the *dechar* as the currency, the Portuguese can be accommodated.

The *erchar* is the definition of the error condition character. It provides a method of indicating that the specification does not match the format of *IN*. This will generate an output string of the prescribed length which contains as many copies or parts of copies of the string in *erchar* as will fit in that length. This is very similar to the function of mismatched data in FORTRAN.

The *fillon* is a truth value which when true, fill characters are applied to the left of the implied decimal position. The default is considered 0 or False unless overridden.

The *filstr* is a string of repeating pattern to be used instead of spaces to fill unused numeric columns.

The *sepchar* is a mechanism for specifying the separator character for the grouping of columns. This should be a single character for each occurrence of the separator fmask sentinel character, s.

7.1.7.6.5 · $%LOWER^STRING

LOWER^STRING(*A*,*CHARMOD*::O)

This function returns a string with upper case characters converted to lower case.

If *CHARMOD* is a global name then the conversion used is that specified in **^$global** for that name.

If *CHARMOD* is a Character Set Profile then the conversion used is that specified in **^$character** for the profile.

If *CHARMOD* is not specified, or the node specified above does not exist, then the conversion defined by the process Character Set Profile is used.

If no algorithm is specified in **^$character** then the characters A ... Z are converted to a ... z respectively.

Domain: *CHARMOD* is either a Character Set Profile specification in the form *charset* or a global name specification in the form ^|*environment*|*name*.

Range: standard

Side effects: none

7.1.7.6.6 · $%PRODUCE^STRING

PRODUCE^STRING(*IN*,.*SPEC*,*MAX*:INTEGER:O)

The function scans a string for the occurrence of certain substrings and replace all such occurrences by another substring. This process is repeated until none of these substrings can be found anymore. The resulting string will be passed back to the caller as the function-value.

The function has two required parameters, a string-value (*IN*) and a translation-specification array (*SPEC*).

The function converts the value of *IN*, according to the specification in *SPEC*. If the optional parameter *MAX* is specified, its value is used as the maximum number of iterations allowed to perform the intended conversion.

For the purpose of this discussion, the string-value will be called *IN* and the translation-specifications will be called *SPEC*(*I*,1)=*FIND*, *SPEC*(*I*,2)=*OUT*, *FIND* being a substring to be located and *OUT* being the substring to be put in its place.

For each element of the form *SPEC*(*I*,1), the function will scan whether *IN* contains the substring *FIND*. If such a substring occurs, it is replaced by *OUT*, which is the value of *SPEC*(*I*,2). After the replacement has been made, *IN* is scanned again for the occurrence of *FIND*. This process continues until the substring is no longer found. When *IN* no longer contains any instance of *FIND*, the next translation-specification is tried.

NOTE: The array *SPEC* may contain overlapping find-strings, e.g. *SPEC*(1,1)="ABCDE" and *SPEC*(2,1)="ABC". Since the array *SPEC* is scanned using `$ORDER` on the first subscript, the longer substring will be replaced, before the shorter one is attempted. If the opposite behaviour is required, the order of the values in *SPEC* should be reversed: *SPEC*(1,1)="ABC" and *SPEC*(2,1)="ABCDE".

Since any translation may cause a substring to be translated to be inserted again, the above process will be repeated until for no specification in the array *SPEC* a matching substring could be found.

Domain: subscripts in the array *SPEC* must contorm to the portability requirement on subscripts

Range: standard

Side effects: none

7.1.7.6.7 · $%REPLACE^STRING

REPLACE^STRING(*IN*,.*SPEC*)

The function scans a string for the occurrence of certain substrings and replace all such occurrences by another substring. This process is repeated until none of these substrings can be found anymore, yet no character in the input-string is replaced more than once. Replaced characters are not affected. The resulting string will be passed back to the caller as the function-value.

The function has two required parameters, a string-value and a translation-specification array. The function converts the value of *IN*, according to the specification in *SPEC*. The function scans a string for the occurrence of certain substrings and replace all such occurrences by another substring. This proces is repeated until none of these substrings can be found anymore, but in such a way that no character in the input-string is translated more than once. The resulting string will be passed back to the caller as the function-value.

For the purpose of this discussion, the string-value will be called *IN* and the translation-specifications will be called *SPEC*(*I*,1)=*FIND*, *SPEC*(*I*,2)=*OUT*, *FIND* being a substring to be located and *OUT* being the substring to be put in its place.

For each element of the form *SPEC*(*I*,1)=*FIND*, the function will scan whether *IN* contains the substring *FIND*. If such a substring occurs, and none of the characters in that substring has been translated because it was part of another substring to be translated, it is replaced by *OUT*, which is the value of *SPEC*(*I*,2). After the replacement has been made, *IN* is scanned again for the occurrence of *FIND*. This process continues until the substring is no longer found with no characters marked as being used before. After that, the next translation-specification is tried.

NOTE: The array *SPEC* may contain overlapping find-strings, e.g. *SPEC*(1,1)="ABCDE" and *SPEC*(2,1)="ABC". Since the array *SPEC* is scanned using $ORDER on the first subscript, the longer substring will be replaced, before the shorter one is attempted. If the opposite behaviour is required, the order of the values in *SPEC* should be reversed: *SPEC*(1,1)="ABC" and *SPEC*(2,1)="ABCDE".

Since any translation may cause a substring to be translated to be inserted again, the above process will not translate any character from the input-string more than once, and will not translate any character that is inserted as the result of a translation.

Domain: subscripts in the array *SPEC* must contorm to the portability requirement on subscripts

Range: standard

Side effects: none

7.1.7.6.8 · $%UPPER^STRING

UPPER^STRING(*A*,*CHARMOD*::0)

This function returns a string with lower case characters converted to upper case.

If *CHARMOD* is a global name then the conversion used is that specified in **^$global** for that name.

If *CHARMOD* is a Character Set Profile then the conversion used is that specified in **^$character** for the profile.

If *CHARMOD* is not specified, or the node specified above does not exist, then the conversion defined by the process Character Set Profile is used.

If no algorithm is specified in **^$character** then the characters a … z are converted to A … Z respectively.

Domain: *CHARMOD* is either a Character Set Profile specification in the form *charset* or a global name specification in the form ^|*environment*|*name*.

Range: standard

Side effects: none

7.2 Expression tail *exprtail*

```
                 | | binaryop   |          |
exprtail ::=     | | [']truthop | expratom |
                 | [']?pattern             |
```

The order of evaluation is as follows:

a Evaluate the left-hand *expratom*.

b If an *exprtail* is present immediately to the right, evaluate its *expratom* or *pattern* and apply its operator.

c Repeat step *b* as necessary, moving to the right.

In the language of operator precedence, this sequence implies that all binary string, arithmetic, and truth-valued operators are at the same precedence level and are applied in left-to-right order.

Any attempt to evaluate an *expratom* containing an *lvn*, *gvn*, *svn*, or *ssvn* with an undefined value is erroneous. A reference to a *lvn* with an undefined value causes an error condition with *ecode*=**"M6"**. A reference to a *gvn* with an undefined value causes an error condition with *ecode*=**"M7"**. A reference to a *svn* with an undefined value causes an error condition with *ecode*=**"M8"**. A rcfercnce to a *ssvn* with an undefined value, where the semantics of that action is not specified for that specific *ssvn*, causes an error condition with *ecode*=**"M60"**.

7.2.1 Binary operator *binaryop*

```
              | _  | (note: underscore)
              | +  |
              | -  | (note: hyphen)
binaryop ::=  | *  |
              | /  |
              | #  |
              | \  |
              | ** |
```

7.2.1.1 Concatenation operator

The underscore symbol _ is the concatenation operator. It does not imply any numeric interpretation. The value of A_B is the string obtained by concatenating the values of A and B, with A on the left.

7.2.1.2 Arithmetic binary operators

The binary operators **+ - * / \ # **** are called the arithmetic binary operators. They operate on the numeric interpretations of their operands, and they produce numeric (in one case, integer) results.

+ produces the algebraic sum.

- produces the algebraic difference.

***** produces the algebraic product.

/ produces the algebraic quotient. Note that the sign of the quotient is negative if and only if one operand is positive and one operand is negative. Division by zero causes an error condition with *ecode*=**"M9"**.

\ produces the integer interpretation of the result of the algebraic quotient.

produces the value of the left operand modulo the right argument. It is defined only for nonzero values of its right operand, as follows.

*A#B=A-(B*floor(A/B))*

where *floor*(*x*)=the largest integer'>*x*. A value of 0 (zero) for *B* will produce an error condition with *ecode*=**"M9"**.

****** produces the exponentiated value of the left operand, raised to the power of the right operand. Results producing complex numbers (e.g., even numbered roots of negative numbers) are not defined. On an attempt to compute **0**(*a negative number*)**, an error will occur with *ecode*=**"M9"**. On an attempt to compute **0**0**, an error will occur with *ecode*=**"M94"**. On an attempt to compute the result of an exponentiation, the true value of which is a complex number with a nonzero imaginary part, an error will occur with *ecode*=**"M95"**.

7.2.2 Truth operator *truthop*

truthop ::= | *relation*
 logicalop |

```
truthop ::= | relation  |
            | logicalop |
```

7.2.2.1 Relational operator *relation*

```
            | =   |
            | ==  |
            | <   |
            | >   |
            | <=  |
relation ::=| >=  |
            | [   |
            | ]   |
            | ]]  |
            | ]=  |
            | ]]= |
```

The operators **=** **==** **<** **>** **<=** **>=** **[** **]** **]]** **]=** and **]]=** produce the truth value **1** if the relation between their operands which they express is true, and **0** otherwise. The dual operators **'***relation* are defined by:

`A` `'`*relation* `B` has the same value as `'(` `A` *relation* `B` `)`.

7.2.2.2 Numeric relations

The inequalities `<` `>` `<=` and `>=` operate on the numeric interpretations of their operands; they denote the conventional algebraic *less than, greater than, less than or equal to,* and *greater than or equal to.*

7.2.2.3 String relations

The relations `=` `==` `[` `]` `]]` `]=` and `]]=` do not imply any numeric interpretation of either of their operands.

The relation `=` tests string identity. If the operands are not known to be numeric and numeric equality is to be tested, the programmer may apply an appropriate unary operator to the nonnumeric operands. If both arguments are known to be in numeric form (as would be the case, for example, if they resulted from the application of any operator except `_`), application of a unary operator is not necessary. The uniqueness of the numeric representation guarantees the equivalence of string and numeric equality when both operands are numeric. Note, however, that the division operator `/` may produce inexact results, with the usual problems attendant to inexact arithmetic.

The relation `==` tests object reference identity. `A==B` is true if and only if all of the following conditions are met:

a The value of `A` has the data type *oref*

b The value of `B` has the data type *oref*

c `A` and `B` identify the same object

In the context of the `==` operator, values are never coerced to any specific data type or interpretation.

The relation `[` is called *contains.* `A[B` is true if and only if `B` is a substring of `A`; that is, `A[B` has the same value as `''$find(A,B)`. Note that the empty string is a substring of every string.

The relation `]` is called *follows.* `A]B` is true if and only if `A` follows `B` in the sequence, defined here. `A` follows `B` if and only if any of the following is true.

a `B` is empty and `A` is not.

b Neither `A` nor `B` is empty, and the leftmost character of `A` follows (i.e., has a numerically greater `$ascii` value than) the leftmost character of `B`.

c There exists a positive integer `n` such that `A` and `B` have identical heads of length `n`, (i.e., `$extract(A,1,n)=$extract(B,1,n)`) and the remainder of `A` follows the remainder of `B` (i.e., `$extract(A,n+1,$length(A))` follows `$extract(B,n+1,$length(B))`).

The relation `]]` is called *sorts after.* `A]]B` is true if and only if `A` follows `B` in the subscript ordering sequence defined by the single argument `$order` function as if that `$order` refers to a *lvn*.

The relation `]=` is called *follows or equal to.* `A]=B` is true if and only if `A` follows `B` as defined above or `A` is identical to `B`.

The relation **]]=** is called *sorts after or equal to*. `A]]=B` is true if and only if `A` sorts after `B` as defined above or `A` is identical to `B`.

7.2.2.4 Logical operator *logicalop*

logicalop ::= | **&**
| **!**
| **!!**

The operators **!**, **!!**, and **&** are called logical operators. (They are given the names *or*, *exclusive or*, and *and,* respectively.) They operate on the truth-value interpretations of their arguments, and they produce truth-value results.

`A!B` = (`0` if both `A` and `B` have the value `0`)
(`1` otherwise)
`A!!B` = (`0` if both `A` and `B` have the value `0` or if both `A` and `B` have the value `1`)
(`1` otherwise)
`A&B` = (`1` if both `A` and `B` have the value `1`)
(`0` otherwise)

The dual operators **'&**, **'!**, and **'!!** are defined by:

`A'&B` = `'(A&B)`
`A'!B` = `'(A!B)`
`A'!!B` = `'(A!!B)`

7.2.3 Pattern match *pattern*

The pattern match operator **?** tests the form of the string which is its left-hand operand. `S?P` is true if and only if `S` is a member of the class of strings specified by the pattern `P`.

A pattern is a concatenated list of pattern atoms.

pattern ::= | *patatom*...
| **@**∈{*expratom* → *pattern*}

Assume that *pattern* has `n` *patatom*s. `S`**?***pattern* is true if and only if there exists a partition of `S` into `n` substrings

$S = S_1\ S_2\ \ldots\ S_n$

such that there is a one-to-one order-preserving correspondence between the S_i and the pattern atoms, and each S_i *satisfies* its respective pattern atom. Note that some of the S_i may be empty.

Each pattern atom consists of a repeat count *repcount*, followed by either a pattern code *patcode* or a string literal *strlit*. A substring S_i of `S` satisfies a pattern atom if it, in turn, can be decomposed into a number of concatenated substrings, each of which satisfies the associated *patcode* or *strlit*.

patatom	::=	*repcount*	*patcode* *patstr* *alternation*	[*patsetdest*]
repcount	::=		*intlit* [*intlit*$_1$]`.`[*intlit*$_2$]	

patcode	::=	[`'`]	`Y` *patnony* `Y`
			`Z` *patnonz* `Z`
			patnonyz
			ob *charspec* *cb*
				... (note: apostrophe)

patnony	::=	any of the characters in *ident* except `Y`		
patnonz	::=	any of the characters in *ident* except `Z`		
patnonyz	::=	any of the characters in *ident* except `Y` and `Z`		
charspec	::=	*strconst*$_1$[`:`*strconst*$_2$]		
strconst	::=		`$c[har](`*list*{*numlit*}`)` *strlit*	
patstr	::=	[`'`] *strlit* (note: apostrophe)		
alternation	::=	`(`*list*{*patgrp*}`)`		
patgrp	::=	*patatom*...		
patsetdest	::=	`(`*setdestination*`)`		

*patcode*s beginning with the initial letter `Y` are available for use by M programmers. *patcode*s beginning with the initial letter `Z` are available for use by implementors. *patcode*s are specified in Character Set Profiles.

a If a *patcode* has the form of a *charspec*, determination of whether a character belongs to the *patcode* is made as follows:

A character belongs to a *charspec* containing only one *strconst* if it is contained in the string represented by that *strconst*.

A character belongs to a *charspec* containing two *strconst*s if it is (inclusively) between them.

Formally, *X* is a member of *S* if *S*`[`*X*, and *X* is a member of *S1*`:`*S2* if *S1* does not trail *X* and *X* does not trail *S2*, but the check against the value of *S2* will be omitted if *S2* is the empty string. If *S2* is present, then neither *S1* nor *S2* may contain more than one character.

If a *strconst* is of the form `$c[har](`...`)`, then it has the same value as the result of the function `$char` called with the same parameters. Use of upper, lower, or mixed case in the name `$char` is permitted.

b Otherwise, *patcode*s differing only in the use of corresponding upper and lower case letters are equivalent. If the apostrophe is not present in a given *patcode*, the *patcode* is satisfied by any single character in the union of the classes of characters represented, each class denoted by its own *patcode* letter. If the apostrophe is present, the *patcode* is satisfied by any single character which is not in the union of the classes of characters represented. Whether or not a specific character belongs to a *patcode* class is determined by a process's Character Set Profile (*charset*).

An *alternation* is satisfied if any one of its *patgrp* components individually matches the corresponding S_i.

Each *patstr* in which an apostrophe is not present is satisfied by, and only by, the value of *strlit*. Each *patstr* in which an apostrophe is present is satisfied by any string of the same length as *strlit* which is not identical to *strlit*.

If *repcount* has the form of an indefinite multiplier ".", *patatom* is satisfied by a concatenation of any number of S_i (including none), each of which meets the specification of *patatom*.

If *repcount* has the form of a single *intlit*, *patatom* is satisfied by a concatenation of exactly *intlit* S_i, each of which meets the specification of *patatom*. In particular, if the value of *intlit* is zero, the corresponding S_i is empty.

If *repcount* has the form of a range, $intlit_1.intlit_2$, the first *intlit* gives the lower bound, and the second *intlit* the upper bound. If the upper bound is less than the lower bound an error condition occurs with *ecode*="**M10**". If the lower bound is omitted, so that the range has the form $.intlit_2$, the lower bound is taken to be zero. If the upper bound is omitted, so that the range has the form $intlit_1$, the upper bound is taken to be indefinite; that is, the range is at least $intlit_1$ occurrences. Then *patatom* is satisfied by the concatenation of a number of S_i, each of which meets the specification of *patatom*, where the number must be within the expressed or implied bounds of the specified range, inclusive.

If more than one one-to-one order-preserving correspondence between the S_i and the pattern atoms exist the following rules are used to select the correspondence used in the two paragraphs following the rules. These rules are applied to each *patatom* in the *pattern*, from left to right and recursively in the case of *alternation*s.

A If the *patatom* is not an *alternation*, select the longest matching substring that produces a match in the pattern as a whole.

B If the *patatom* is an *alternation*, use the below rules and apply rules *A* and *B* recursively to each *patatom* in the selected *patgrp*(s) from left to right.

   1 Select the correspondence(s) that uses the smallest possible value of the alternation's *repcount*.

   2 If multiple correspondences satisfy *1*, for each sequential application of the *alternation* (i.e., each value of the *repcount*) select the *patgrp*(s) within the *alternation* that correspond to the longest possible substring.

   3 If multiple correspondences satisfy *1* and *2*, select the leftmost *patgrp* in the *alternation*.

Each optional *patsetdest*, if any, is executed only if *S***?***pattern* is true, and only if the associated pattern atom is satisfied by one of the S_i in the selected correspondence. If these conditions hold, these (and only these) *patsetdest*s are executed from left to right as follows:

For each of the substrings S_i of S satisfying the pattern atom in the selected correspondence, in the order in which they (the S_i) appear in the string, perform all the actions of `set` *setdestination*=S_i as defined in section 8.2.28.

The dual operator `'?` is defined by:

```
A'?B = '(A?B)
```

8 Commands

8.1 General command rules

Every *command* starts with a *commandword* which dictates the syntax and interpretation of that *command* instance. *Commandword*s differing only in the use of corresponding upper and lower case letters are equivalent. The standard contains the following *commandword*s:

```
                  | ab[lock]          |
                  | a[ssign]          |
                  | asta[rt]          |
                  | asto[p]           |
                  | aunb[lock]        |
                  | b[reak]           |
                  | c[lose]           |
                  | d[o]              |
                  | e[lse]            |
                  | esta[rt]          |
                  | esto[p]           |
                  | et[rigger]        |
                  | f[or]             |
                  | g[oto]            |
                  | h[alt]            |
                  | h[ang]            |
                  | i[f]              |
                  | j[ob]             |
                  | k[ill]            |
commandword ::=   | ks[ubscripts]     |
                  | kv[alue]          |
                  | m[erge]           |
                  | n[ew]             |
                  | o[pen]            |
                  | q[uit]            |
                  | r[ead]            |
                  | rl[oad]           |
                  | rs[ave]           |
                  | s[et]             |
                  | tc[ommit]         |
                  | th[en]            |
                  | tre[start]        |
                  | tro[llback]       |
                  | ts[tart]          |
                  | u[se]             |
                  | v[iew]            |
                  | w[rite]           |
                  | x[ecute]          |
                  | z[unspecified]    |
```

Unused *commandwords* other than those starting with the letter **z** are reserved for future enhancement of the *Standard*.

Any implementation of the language must be able to recognize both the abbreviated *commandword* (i.e., the character(s) to the left of the "[" in the list above) and the full spelling of each *commandword*. When two *commands* have a common abbreviated *commandword*, their argument syntax uniquely distinguishes them.

The formal definition of the syntax of *command* is a choice from among all of the individual *command* syntax definitions of 8.2.

command ::=
| syntax of **assign** command |
| syntax of **break** command |
| . |
| . |
| . |
| syntax of **xecute** command |
| syntax of **z**[unspecified] command |

For all *commands* allowing multiple arguments, the form

commandword *arg*$_1$**,***arg*$_2$**,**...*arg*$_n$

is equivalent in execution to

commandword *arg*$_1$ *commandword* *arg*$_2$ ... *commandword* *arg*$_n$

Within a *command*, all *expratoms* are evaluated in a left-to-right order with all *expratoms* that occur to the left of the *expratom* being evaluated, including the complete resolution of any indirection, prior to the evaluation of that *expratom*, except as explicitly noted elsewhere in this document. The *expratom* is formed by the longest sequence of characters that satisfies the definition of *expratom*. (See 7.1 for a description of *expratom*).

An error condition occurs, with *ecode*="**M11**", when execution begins of any *formalline* unless that *formalline* has just been reached as a result of an *exvar*, an *exfunc*, a **job** command *jobargument*, or a **do** command *doargument* that contains an *actuallist*.

8.1.1 Spaces in *commands*

Spaces are significant characters. The following rules apply to their use in *lines*.

a If a *command* instance contains no argument and it is not the last *command* of the *line*, or if a *comment* or *extsyntax* follows, the *commandword* or *postcond* is followed by at least two spaces. If it is the last *command* of the *line* and no *comment* or *extsyntax* follows, the *commandword* or *postcond* may be followed by zero or more spaces.

b In all other cases, the use of spaces is defined by the appropriate *command* definition and subclauses of 6.2 Routine body, and 6.4 Embedded programs.

8.1.2 Comment *comment*

If a semicolon appears in the *commandword* initial-letter position, it is the start of a *comment*. The remainder of the *line* to *eol* must consist of graphics only, but is otherwise ignored and nonfunctional.

8.1.3 Command argument indirection

Indirection is available for evaluation of either individual command arguments or contiguous sublists of command arguments. The opportunities for indirection are shown in the syntax definitions accompanying the command descriptions.

Typically, where a *commandword* carries an argument list, as in

commandword *sp* *list*{*argument*}

the *argument* syntax will be expressed as

$$argument \ ::= \left| \begin{array}{l} \text{individual argument syntax} \\ \mathtt{@}\in\{expratom \rightarrow list\{argument\}\} \end{array} \right|$$

This formulation expresses the following properties of argument indirection.

a Argument indirection may be used recursively.

b A single instance of argument indirection may evaluate to one complete argument or to a sublist of complete arguments.

Unless the opposite is explicitly stated, the text of each command specification describes the arguments *after* all indirection has been evaluated.

Unless expressed otherwise, if individual argument syntax allows the **@***expratom* contruct, then argument indirection has precedence, i.e., the restriction on the value of *expratom* comes from the ∈{→} operator of the argument indirection, not any other type of indirection.

8.1.4 Generic command indirection

If the evaluation of a *command* or a *command*'s argument(s) encounters an indirect expression (of the form: **@***expritem*) which cannot be resolved using the syntax/metalanguage defined for the *command* (if appropriate), the **@** and *expritem* are replaced with the value returned by the *expritem* and the result interpreted again as if it were part of the original *command* or *linebody*. Note that this permits an indirect expression to be interpreted as if it were a *command* in the *linebody* definition. If this replacement results in a syntax which does not match the definition of a *routine*, an error condition occurs with *ecode*=**"S0"**.

The replacement with the contents of the *expritem* may not result in the insertion of *cs*, *eol*, *line*, and *eor* elements into the routine execution.

8.1.5 Post conditional *postcond*

All commands except **`else`**, **`for`**, and **`if`** may be made conditional as a whole by following the *commandword* immediately by the post-conditional *postcond*.

postcond `::=` `[:`*tvexpr*`]`

If the *postcond* is absent or the *postcond* is present and the value of the *tvexpr* is true, the *command* is executed. If the *postcond* is present and the value of the *tvexpr* is false, the *commandword* and its arguments are passed over without execution.

The *postcond* may also be used to conditionalize the arguments of `do`, `goto`, and `xecute`. In such cases the arguments' *expratom*s that occur prior to the *postcond* are evaluated prior to the evaluation of the *postcond*.

8.1.6 Command timeout *timeout*

The `open`, `lock`, `job`, and `read` commands employ an optional timeout specification, associated with the testing of an external condition.

timeout `::=` `:`*numexpr*

If the optional *timeout* is absent, the *command* will proceed if the condition, associated with the definition of the *command*, is satisfied; otherwise, it will wait until the condition is satisfied and then proceed.

`$test` will not be altered if the *timeout* is absent.

If the optional *timeout* is present, the value of *numexpr* must be nonnegative. If it is negative, the value `0` is used. *Numexpr* denotes a `t`-second timeout, where `t` is the value of *numexpr*.

If `t=0`, the condition is tested. If it is true, `$test` is set to `1`; otherwise, `$test` is set to `0`. Execution proceeds without delay.

If `t` is positive, execution is suspended until the condition is true, but in any case no longer than `t` seconds. If, at the time of resumption of execution, the condition is true, `$test` is set to `1`; otherwise, `$test` is set to `0`.

8.1.7 Line reference *lineref*

The `do`, `goto`, and `job` commands, extrinsic functions and extrinsic variables, as well as the `$text` function, contain in their arguments means for referring to particular *line*s within any *routine*. This subclause describes the means for making *line* references.

A reference to a *line* is either an *entryref* or a *labelref*. An *entryref* allows the specification of integer offsets from a label (eg, `loop+5` references the fifth *line* after the *line* that has `loop` for a *label*). Also, an *entryref* allows indirection of both the *label* and the *routinename*. A *labelref*, on the other hand, allows neither *label* offsets nor indirection.

lineref `::=` | *entryref* |
| *labelref* |

8.1.7.1 Entry reference *entryref*

The total line specification in `do`, `goto`, `job`, and `$text` is in the form of *entryref*.

entryref `::=` | *dlabel*`[`+*intexpr*`]``[`^*routineref*`]` |
| ^*routineref* |

If the routine reference (`^`*routineref*) is absent, the routine being executed is implied. If the line reference (*dlabel*`[+`*intexpr*`]`) is absent, the first *line* is implied.

If `+`*intexpr* is absent, the *line* denoted by *dlabel* is the one containing *label* in a defining occurrence. If `+`*intexpr* is present and has the value `n'<0`, the *line* denoted is the *n*th line after the one containing *label* in a defining occurrence. A negative value of *intexpr* causes an error condition with *ecode*=`"M12"`. When *label* is an instance of *intlit*, leading zeros are significant to its spelling.

In the context of `do`, `goto`, or `job`, either of the following conditions causes an error condition with *ecode*=`"M13"`.

a A value of *intexpr* so large as not to denote a *line* within the bounds of the given *routine*.

b A spelling of *label* which does not occur in a defining occurrence in the given *routine*.

In any context, reference to a particular spelling of *label* which occurs more than once in a defining occurrence in the given *routine* will have undefined results.

`Do`, `goto`, and `job` commands, as well as the `$text` *function*, can refer to a *line* in a *routine* other than that in which they occur; this requires a means of specifying a *routinename*.

Any *line* in a given *routine* may be denoted by mention of a *label* which occurs in a defining occurrence on or prior to the *line* in question.

dlabel `::=` | *label*
| `@`∈{*expratom* → *dlabel*}

routineref `::=` | `[`*vb* *environment* *vb*`]`*routinename*
| `@`∈{*expratom* → *routineref*}

If the *routineref* includes an *environment*, then the *routine* is fetched from the specified *environment*. Reference to a non-existent *environment* causes an error condition with an *ecode*=`"M26"`.

8.1.7.2 Label reference *labelref*

When the `do` or `job` commands or *exfunc* or *exvar* include parameters to be passed to the specified *routine*, the `+`*intexpr* form of *entryref* is not permitted and the specified *line* must be a *formalline*. The *line* specification *labelref* is used instead:

labelref `::=` | *label*`[^[`*vb* *environment* *vb*`]`*routinename*`]`
| `^[`*vb* *environment* *vb*`]`*routinename*

If the routine reference (`^[|`*environment*`|]`*routinename*) is absent, the routine being executed is implied. If the line reference (*label*) is absent, the first *line* is implied.

If the *labelref* includes an *environment*, then the *routine* is fetched from the specified *environment*. Reference to a non-existent *environment* causes an error condition with an *ecode*=`"M26"`.

In the context of a **do** or **job** command, an *exfunc*, or an *exvar*, a spelling of *label* which does not occur in a defining occurrence in the given *routine* causes an error condition with *ecode*="**M13**".

8.1.7.3 External reference *externref*

externref ::= **&**[*packagename*.]*externalroutinename*

packagename ::= *name*

externalroutinename ::= *name*[^*name*]

The ampersand (**&**) character designates a program whose namespace is external to the current M environment. The effects of passing parameters are as defined in 8.1.8 (Parameter Passing).

The *packagename* shall be from a namespace of those determined by the appropriate namespace registry. If *packagename* is not specified, implementors may, optionally, choose to provide a default package.

Bindings may have one or more namespaces; requirements to use these namespaces must be clearly stated in the specification of the binding. The term *package* is used herein to denote programs that are in possibly external environments. No implied one-to-one correspondence for all possible external packages exists.

The *externalroutinename* namespace is undefined; this is a function of a binding. Any external mapping between the *externalroutinename* and any name used by an external package is an implementation-specific issue. The *externalroutinename* shall be of the form *name* or *name*^*name*.

8.1.7.4 Library reference *libraryref*

libraryref ::= **%***libraryelement*[^*library*]

libraryelement ::= *name*

library ::= *name*

If no *library* is specified as part of a *libraryref* then the libraries specified in **^$job($job,"LIBRARY")** are used. Note: This does not imply that the libraries specified in **^$job($job,"library")** can necessarily be dynamically changed during the lifetime of a process.

Unless explicitly specified in an individual *libraryelement* definition, accessing a *libraryref* has no effect on loca1 variables for a process, **$reference**, and **$test**, except for a return value and changes to variables passed by reference.

If an argument to a *libraryref* has an invalid value (such as a value outside the domain of the function) the behaviour of the reference to the *libraryref* is undefined.

The restrictions specified in 8.1.8 Parameter passing also apply to the referencing of *libraryref*s.

If a *libraryelement* or a *library* is not available for a library reference then an error condition occurs with *ecode*=`"M13"`.

8.1.8 Parameter passing

Parameter passing is a method of passing information in a controlled manner to and from a subroutine or process as the result of an *exfunc*, an *exvar*, or a **`do`** command with an *actuallist*, or to a process as the result of a **`job`** command with an *actuallist*.

$$\mathit{actuallist}\ \texttt{::=}\ \texttt{(}[\mathit{list}\{\mathit{actual}\}]\texttt{)}$$

$$\mathit{actual}\ \texttt{::=}\ \left[\begin{array}{l}\texttt{.}\mathit{actualname}\\ \mathit{expr}\end{array}\right]$$

$$\mathit{actualname}\ \texttt{::=}\ \left|\begin{array}{l}\mathit{name}\\ \texttt{@}\in\{\mathit{expratom}\rightarrow\mathit{actualname}\}\end{array}\right|$$

When parameter passing occurs, the *formalline* designated by the *labelref* must contain a *formallist* in which the number of *name*s is greater than or equal to the number of *actual*s in the *actuallist*. The correspondence between *actual* and *formallist name* is defined such that the first *actual* in the *actuallist* corresponds to the first *name* in the *formallist*, the second *actual* corresponds to the second *formallist name*, etc. Similarly, the correspondence between the parameter list entries, as defined below, and the *actual* or *formallist name*s is also by position in left-to-right order. If the syntax of *actual* is `.`*actualname*, then it is said that the *actual* is of the call-by-reference format; if the syntax of *actual* is *expr* it is said that the *actual* is of the call-by-value format; otherwise it is said that the *actual* is of the omitted-parameter format.

When parameter passing occurs, the following steps are executed:

a Process the *actual*s in left-to-right order to obtain a list of *data-cell* pointers called the parameter list. The parameter list contains one item per *actual*. The parameter list is created according to the following rules:

 1 If the *actual* is call-by-value, then evaluate the *expr* and create a *data-cell* with a zero tuple value equal to the result of the evaluation. An *expr* that returns a value of data type *oref* is coerced into a value of data type *mval* in the *actuallist* of *externref*s and **`job`** *commands* (see 7.1.1.2 for the coercion rules), but not in any other *actuallist*s. The pointer to this *data-cell* is the parameter list item.

 2 If the *actual* is call-by-reference, search the *name-table* for an entry containing the *actual name*. If an entry is found, the parameter list item is the *data-cell* pointer in this *name-table* entry. If the *actual name* is not found, create a *name-table* entry containing the *name* and a pointer to a new (empty) *data-cell*. This pointer is the parameter list item. If a *jobargument* contains a call-by-reference *actual* an error occurs with *ecode*=`"M40"`.

3 If the *actual* is omitted-parameter, create a new (empty) *data-cell*.

b Place the information contained in the *formallist* in the *process-stack* frame.

c For each *name* in the *formallist*, search the *name-table* for an entry containing the *name* and if the entry exists, copy the *name-table* entry into the parameter frame and delete it from the *name-table*. This step performs an implicit **new** on the *formallist names*.

d For each item in the parameter list, create a *name-table* entry containing the corresponding *formallist name* and the parameter list item (*data-cell* pointer). This step binds the *formallist names* to their respective *actual*s.

As a result of these steps, two (or more) *name-table* entries may point to the same *data-cell*. As long as this common linkage is in effect, an **assign**, set, or **kill** of an *lvn* with one of the *name*s appears to perform an implicit **assign**, **set**, or **kill** of an *lvn* with the other *name*(s). Note that a **kill** does not undo this linkage of multiple *name*s to the same *data-cell*, although subsequent parameter passing or **new** commands may.

Execution is then initiated at the first *command* following the *ls* of the *line* specified by the *labelref*. Execution of the subroutine continues until an *eor* or a **quit** is executed that is not within the scope of a subsequently executed *doargument*, argumentless **do**, *xargument*, *exfunc*, *exvar*, or **for**. In the case of an *exfunc* or *exvar*, the subroutine must be terminated by a **quit** with an argument.

At the time of the **quit**, the *formallist* names are unbound and the original variable environment is restored. See 8.2.24 for a discussion of the semantics of the **quit** operation.

When calling to an *externref*, call-by-reference has the following additional implementation independent definition:

a Upon return of control to M, changes to the value of the *lvn* referenced by the *actualname* shall be as if the *lvn* was modified by an **assign** or **set** command, as appropriate. The exact mechanism performing this operation is unspecified.

b The resultant events are unspecified, if the data in the M environment is modified while an external routine call is being made that references the modified data.

c Local variables (see 7.1.2 Variables) that are not passed as parameters, will not necessarily be available to the external environment.

8.1.9 Object usage

An *object* is an identifiable, encapsulated, entity that has state and that provides one or more *service*s called *method*s and *propertie*s. A *service* may be accessed from a routine. The only means of observing or changing the state or behavior of an object is by use of its *service*(s).

*Object*s are not named. A value of data type *oref* (object reference) is a value that identifies an *object* in an implementation-specific way. A value may be of one of two data types: either data type *oref*, or data type *mval*.

A value of data type *oref* may be assigned to an *lvn* and may be used only in certain specified contexts.

```
   oref ::= any value with data type oref
   mval ::= any value with data type mval
```

8.1.9.1 Accessing a service

A *service* is identified by a name, called a *servicename*.

```
     servicename ::= | name   |
                     | strlit |
```

A *service* is accessed explicitly by means of an *owservice* (object with service):

```
       owservice ::= | owmethod   |
                     | owproperty |
        owmethod ::= object.fservice
      owproperty ::= object.[fservice]
          object ::= @∈{expratom → oref}
        fservice ::= servicename[namedactuallist]
 namedactuallist ::= ( | list{actual}[,list{namedactual}] | )
                       | list{namedactual}                 |
     namedactual ::= actualkeyword:=actual
   actualkeyword ::= | name   |
                     | strlit |
```

The *object* specifies an *object*, and the *fservice* specifies the *service* to be requested of that *object*.

If an *expratom* is used in a context where an *object* is expected, and the *expratom* does not return a value of the data type *oref*, an error will occur with *ecode*=**"M108"** (not an object).

If, in the context of an *owmethod* or an *owproperty*, an *expratom* returns a value of data type *oref* that refers to an *object* that is not currently accessible, an error will occur with *ecode*=**"M105"** (inaccessible object).

If an *fservice* fails to specify a *service* provided by the *object*, or that *service* does not support the context of the access, an error will occur with *ecode*=**"M106"** (invalid service). In the case of a *property*, if no *fservice* is specified, the value of the default *property*, if any, for the *object* is used. If there is no default *property*, an error will occur with *ecode*=**"M107"** or (no default value).

A *namedactuallist* may contain positional parameters and named parameters. An *actualkeyword* specifies the name of the parameter in the *service* being accessed that will receive the *actual*. An *actual* without an *actualkeyword* is a positional parameter. Note: names of *services* and parameters that do not conform to the syntax of a *name* can be used with external *objects*. In these cases, the name of the *service* must be represented as a *strlit* i.e., such names must be enclosed in quotation marks, and any quotation marks within the name must be spelled twice. A *strlit* that evaluates to a *name* is equivalent to that *name* when used as a *servicename* or an *actualkeyword*.

Upon completion of a *service*, the values of **`$test`** and the naked indicator will be restored to their respective values prior to execution of the *service*.

The meaning of invoking an *object*'s services, either implicitly or explicitly, when the value of **`$tlevel`** is greater than `0` is reserved.

8.1.10 User-defined *mnemonicspaces*

When a *controlmnemonic* is used for a device which has a user-defined *mnemonicspace* (see 8.2.34) then the usage of the *controlmnemonic* in a **`read`** and **`write`** command format in the form

/*controlmnemonic***(***expr***,**...**)**

is computationally equivalent, with the exception of the effect on **`$test`** and the naked indicator, to

`do` *`label^routine`*`(`*expr*`,`...`)`

where *`routine`* is the user-defined *mnemonicspace* routine and *`label`* is *controlmnemonic*, unless *controlmnemonic* commences with a **`?`** in which case it is replaced by **`&`**.

`$test` and the naked indicator are restored to their value prior to the execution of the *controlmnemonic* associated routine. **`$test`** is not restored if there is a *timeout* on the original *command*.

Any reference to a *controlmnemonic* within a user-defined *mnemonicspace* for which there is no associated *line* causes an error condition with *ecode*=`"M32"`.

```
                        | CLOSE |
devicecommand  ::=      | OPEN  |
                        | USE   |
```

If a label of the form **`%`***`command`*, where *`command`* is a *devicecommand*, exists in a *mnemonicspace* command routine then execution of a *command* which is a *devicecommand* with at least one *deviceparam* is computationally equivalent to

```
new keyword,attrib,i
set (keyword,attrib)=no
for i=1:1:no do
. set keyword(i)=key_i
. if $data(att_i) set attrib(i)=att_i
do %label^routine(expr,.keyword,.attrib,time)
```

where *`label`* is the *commandword* converted to upper-case and expanded to the fully spelled out *devicecommand*, *`routine`* is the user-defined *mnemonicspace* command routine, *`no`* is the number of *deviceparams*, `keyword` and `attrib` contain the individual *deviceparams* in *deviceparameters* fully evaluated with **key$_i$**=*devicekeyword$_i$* or *deviceattribute$_i$* as appropriate and **att$_i$**=*expr$_i$* if *deviceparam* is in the *deviceattribute* form, and **time** is absent or the evaluated expression from *timeout* if *timeout* is present.

The usage of the *deviceparam* form *expr* is implementation specific.

Any action implied by the presence of a *mnemonicspace* in such a *command* takes effect before the above code is executed.

```
iocommand ::= | READ  |
              | WRITE |
```

If a label of the form **%command**, where **command** is an *iocommand*, exists in a *mnemonicspace* command routine then execution of an *iocommand* of the form

a **w[rite]** *ffformat*
b **w[rite]** *nlformat*
c **w[rite]** *tabformat*
d **w[rite]** *expr*
e **w[rite]** ***intexpr**
f **r[ead]** *glvn*[*readcount*][*timeout*]
g **r[ead]** *ffformat*
h **r[ead]** *nlformat*
i **r[ead]** *tabformat*
j **r[ead]** *strlit*
k **r[ead]** ***glvn**[*timeout*]

is respectively computationally equivalent, with the exception of the effect on **$test** and the naked indicator, to

a **do %WRITEFF^routine()**
b **do %WRITENL^routine()**
c **do %WRITETAB^routine(***intexpr***)**
d **do %WRITE^routine(***expr***)**
e **do %WRITE^routine(***intexpr***)**
f **set** *glvn***=$$%READ^routine(***intexpr$_1$*[**,***intexpr$_2$*]**)**
g **do %WRITEFF^routine(1)**
h **do %WRITENL^routine(1)**
i **do %WRITETAB^routine(,***intexpr***),1)**
j **do %WRITE^routine(***strlit***,1)**
k **set** *glvn***=$$%READS^routine(**[**,***intexpr$_2$*]**)**

where **routine** is the user-defined *mnemonicspace* command routine. *intexpr* is the *intexpr* from *readcount*, or absent if no *readcount* is present. If *timeout* is present, *intexpr$_2$* is the *intexpr* from *timeout*.

During the execution of any user-defined *mnemonicspace* command routine, **read** and **write** re-direction for the device which caused the routine to be executed is disabled.

Upon completion of execution of a routine associated with a user-defined *mnemonicspace*:

a the naked indicator and

b **$test**

are restored to their original values.

During the execution of any user-defined *mnemonicspace* routine the effect of user-defined processing of *controlmnemonic*s and *command*s for the same *mnemonicspace* is unspecified.

Note: **$storage** may be affected by the execution of user-defined *mnemonicspace* code.

Note: It is the responsibility of the user-defined *mnemonicspace* routine to process the *deviceparameter*s in the appropriate order and modify **$test** appropriately in the event that a *timeout* is present.

8.2 Command definitions

The specifications of all *command*s follow.

8.2.1 Ablock

```
                            ┌                ┐
                            │ list{evclass}  │
ab[lock] postcond sp        │ (list{evclass})│
                            └                ┘
```

```
                       │ COMM            │
                       │ IPC             │
                       │ INTERRUPT       │
evclass ::= ∈{expr →   │ POWER           │ }
                       │ TIMER           │
                       │ USER            │
                       │ Z[unspecified]  │
```

Event classes not specified above are reserved for future use.

Ablock temporarily blocks events during critical sections of a process. The three forms of **ablock** are given the following names:

a	*list*{*evclass*}	Selective **ablock**
b	**(***list*{*evclass*}**)**	Exclusive **ablock**
c	Empty argument list	**ablock** All

In the Selective **ablock**, the named event classes are blocked as described below. In the Exclusive **ablock**, all event classes except the named event classes are blocked as described below. In the **ablock** All, all event classes are blocked as described below.

When an event class is blocked, an internal counter for that event class is incremented. If the counter has a positive value, all events of that class are blocked from interrupting the process executing the **ablock** command. If a registered event occurs while blocked, the event is queued. Unregistered events are not queued. Additional subsequent events may be queued if space is provided by the implementation (space for only one event is guaranteed). Events, if queued, will occur in the order in which they occurred when the block is removed (i.e., when the counter becomes zero). All events for a process are stored in one of two queues (one for synchronous events, the other for asynchronous events), rather than a separate queue for each class. Each process, however, must maintain its own queues, as each process blocks and unblocks events independently.

8.2.2 Assign

a[ssign] *postcond* *sp* *list*{*assignargument*}

assignargument ::= | *assigndestination*=*object*
 | **@**∈{*expratom* → *list*{*assignargument*}} |

assignargument ::=	*assigndestination*=*object* **@**∈{*expratom* → *list*{*assignargument*}}
assigndestination ::=	*assignleft* **(***list*{*assignleft*}**)**
assignleft ::=	*lvn* *owproperty*

Assign is a special means for explicitly assigning a reference to an *object* to an *lvn*. The **assign** command behaves similar to the **set** command, with the exception that the final value of the *expr* to the right-hand side of the **=** sign must be of data type *oref* and will not be coerced into a value of data type *mval*. See notes under 7.1.1.2 for the results of the *assign* command with various operands.

This special behavior allows the **assign** command to transfer the value of data type *oref* to the *assignleft*.

8.2.3 Astart

asta[rt] *postcond* *sp* [*list*{*evclass*}
 (*list*{*evclass*}**)**]

Astart enables asynchronous event processing for all or selected event classes. The three forms of **astart** are given the following names:

a	*list*{*evclass*}	Selective **astart**
b	**(***list*{*evclass*}**)**	Exclusive **astart**
c	Empty argument list	**astart** All

In the Selective **astart**, the named event classes are enabled for asynchronous event processing as described below. In the Exclusive **astart**, all event classes except the named event classes are enabled for asynchronous event processing as described below. In the **astart** All, all event classes are enabled for asynchronous event processing as described below.

If any of the classes being enabled for asynchronous event processing are currently enabled for synchronous event processing an error occurs with *ecode*=**"M102"**.

Event classes are enabled by **astart** only for the process executing the **astart** command. It is not an error to enable an event class which is already enabled for the asynchronous model.

8.2.4 Astop

$$\texttt{asto[p]}\ \textit{postcond}\ \textit{sp} \begin{bmatrix} \textit{list}\{\textit{evclass}\} \\ \texttt{(}\textit{list}\{\textit{evclass}\}\texttt{)} \end{bmatrix}$$

Astop disables asynchronous event processing for all or selected event classes. The three forms of **astop** are given the following names:

a	*list*{*evclass*}	Selective **astop**
b	**(***list*{*evclass*}**)**	Exclusive **astop**
c	Empty argument list	**astop** All

In the Selective **astop**, the named event classes are disabled for asynchronous event processing as described below. In the Exclusive **astop**, all event classes except the named event classes are disabled for asynchronous event processing as described below. In the **astop** All, all event classes are disabled for asynchronous event processing as described below.

When asynchronous event processing is disabled for a given event class, events of that class have no effect on the process. Event classes are disabled by **astop** only for the process executing the **astop** command. It is not an error to disable an event class which is already disabled.

8.2.5 Aunblock

$$\texttt{au[nblock]}\ \textit{postcond}\ \textit{sp} \begin{bmatrix} \textit{list}\{\textit{evclass}\} \\ \texttt{(}\textit{list}\{\textit{evclass}\}\texttt{)} \end{bmatrix}$$

Aunblock removes a temporary block on events that was imparted by **ablock**. The three forms of **aunblock** are given the following names:

a	*list*{*evclass*}	Selective **aunblock**
b	**(***list*{*evclass*}**)**	Exclusive **aunblock**
c	Empty argument list	**aunblock** All

In the Selective **aunblock**, the named event classes are unblocked as described below. In the

Exclusive **aunblock**, all event classes except the named event classes are unblocked as described below. In the **aunblock** All, all event classes are unblocked as described below.

When an event class is unblocked, the internal counter for the event class (see 8.2.1 **ablock**) is decremented, unless it is already zero (the counter may not be negative). If the counter is zero, the temporary block, if any, on the event class is removed. Pending events (see 8.2.1 **ablock**), if any, occur in the order in which they arrived. Blocks are removed only for the process executing the **aunblock** command. It is not an error to unblock events which are not currently blocked.

8.2.6 Break

b[reak] *postcond*	[*sp*] argument syntax unspecified

Break provides an access point within the standard for nonstandard programming aids. **break** without arguments suspends execution until receipt of a signal, not specified here, from a device.

8.2.7 Close

c[lose] *postcond* *sp* *list*{*closeargument*}

closeargument	::=	*devn*[**:***deviceparameters*] **@**∈{*expratom* → *list*{*closeargument*}}
devn	::=	[*vb* *environment* *vb*]*expr*
deviceparameters	::=	*deviceparam* **(**[[*deviceparam*]**:**]...*deviceparam***)**
deviceparam	::=	*expr* *devicekeyword* *deviceattribute***=***expr*
devicekeyword	::=	[**/**]*name*
deviceattribute	::=	[**/**]*name*

Devn identifies a device. (In this paragraph, *device* encompasses I/O devices, files, data sets, and other objects supporting **open**, **use**, **read**, **write**, and **close** commands.) When *environment* is omitted, the value of *expr* denotes one device. When *environment* is present, the value of *environment* denotes one set of devices, while the value of *expr* denotes one member of the set. The interpretation of the values is left to the implementor. Reference to a non-existent *environment* causes an error condition with *ecode*=**"M26"**.

Each designated device is released from ownership. If a device is not owned at the time that it is named in an argument of an executed **close**, the command has no effect upon the ownership and the values of the associated parameters of that device. If the current device is named in an argument of an executed `close`, `$io` is given a value of the empty string.

The *deviceparameters* may be used to specify termination procedures or other information associated with relinquishing ownership, in accordance with implementor interpretation. The order of execution of *deviceparam*s is from left to right within a *deviceparameters* usage. When a *deviceparam* is encountered that contains a *deviceattribute* for which there is no defined meaning in the current *mnemonicspace*, the implementation may or may not cause an error condition with *ecode*="`M109`".

If there is no *mnemonicspace* in use for a device or the current *mnemonicspace* is the empty string then the implementation may allow any of the forms of *deviceparam*. The *expr* form may not be mixed with the other forms within the same *deviceparameters*.

In all other cases the *expr* form is not allowed.

Device parameters in effect at the time of the execution of `close` are retained for possible future use in connection with the device to which they apply. (But see 8.3.1, which specifies an exception for output time out).

8.2.8 `Do`

```
d[o] postcond | [sp]                |
              | sp list{doargument} |

                | entryref postcond               |
                | labelref actuallist postcond    |
doargument ::=  | externref [actuallist] postcond |
                | owmethod postcond               |
                | @∈{expratom → doargument}       |
```

An argumentless `do` initiates execution of an inner block of *line*s. If *postcond* is present and its *tvexpr* is false, the execution of the *command* is complete. If *postcond* is absent, or the *postcond* is present and its *tvexpr* is true, the `do` places a `do` frame containing the current execution location, the current execution level, and the current value of `$test` on the *process-stack*, increases the execution level by one, and continues execution at the next *line* in the routine. (See 6.3 for an explanation of routine execution.) When encountering an implicit or explicit `quit` not within the scope of a subsequently executed *doargument*, argumentless `do`, *xargument*, *exfunc*, *exvar*, or `for`, execution of this block is terminated (see 8.2.24 for a description of the actions of `quit`). Execution resumes at the *command* (if any) following the argumentless `do`.

`Do` with arguments is a generalized call to the subroutine specified by the *entryref*, the *labelref*, or the *externref*, or to the method specified by the *owmethod*, in each *doargument*. The *line* specified by the *entryref* or *labelref*, must have a *level* of one. If the line specified is an *externref* then an implicit level of `1` is assumed, unless otherwise specified within the binding. Execution of a *doargument* to a *line* whose level is not one causes an error condition with *ecode*="`M14`".

If the *actuallist* is present in an executed *doargument*, parameter passing occurs and the *formalline* designated by *labelref* must contain a *formallist* in which the number of *names* is greater than or equal to the number of *actuals* in the *actuallist*. If the call is to an *externref* and an *actuallist* is present, then parameter passing occurs, and data is transferred (with any conversion as defined in the binding to the external package).

Each *doargument* is executed, one at a time in left-to-right order, in the following steps.

a Evaluate the *expratoms* of the *doargument*.

b If *postcond* is present and its *tvexpr* is false, execution of the *doargument* is complete. If *postcond* is absent, or *postcond* is present and its *tvexpr* is true, proceed to the step *c*.

c A **do**-frame containing the current execution location and the execution level are placed on the *process-stack*.

d If the *actuallist* is present, execute the sequence of steps described in 8.1.8 Parameter Passing.

e Continue execution at the first *command* position specified by the reference as follows:

1 For *entryref* and *labelref*, this is the first *command* that follows the *ls* of the *line* specified by *entryref* or *labelref*. Execution of the subroutine (within the M environment) continues until an *eor* or a **quit** is executed that is not within the scope of a subsequently executed **for**, argumentless **do**, *doargument*, *xargument*, *exfunc*, or *exvar*. The scope of this internally referenced *doargument* is said to extend to the execution of that **quit** or *eor*. (See 8.2.24 for a description of the actions of **quit**.) Execution then returns to the first character position following the *doargument*.

2 For *externref*, this is the first executable item as specified within the package environment. If the reference is external to M, execution proceeds in the specified environment until termination, as defined within that environment, occurs. Execution then returns to the first character following the *doargument*.

3 For *owmethod*, refer to 8.1.9 Object usage.

8.2.9 Else

e[**lse**] [*sp*]

If the value of **$test** is 1, the remainder of the *line* to the right of the **else** is not executed. If the value of **$test** is 0, execution continues normally at the next *command*.

8.2.10 Estart

esta[**rt**] *postcond* *sp* [*list*{*wevclass*} | **(***list*{*wevclass*}**)**]

wevclass ::= | *evclass* |

| **@**∈{*expr* → **"WAPI"**} |

Estart enables synchronous event processing for the selected event classes. The additional class **"WAPI"** is provided to enable just the synchronous event processing specified in *X11.6,* the MWAPI. If any of the event classes being enabled for synchronous event processing is currently enabled for asynchronous event processing, an error occurs with *ecode*=**"M102"**. It is not an error to enable an event class which is already enabled for synchronous event processing.

Synchronous event processing remains activated until the termination of execution of the **estart** command, except that synchronous event processing is implicitly deactivated at the initiation of call back processing for each event. At the conclusion of call back processing for each event, synchronous event processing is implicitly reactivated.

The three forms of **estart** are given the following names:

a	*L evclass*	Selective **estart**
b	**(** *L evclass* **)**	Exclusive **estart**
c	Empty argument list	**estart** All

In the Selective **estart**, the named event classes are enabled for synchronous event processing as described below. In the Exclusive **estart**, all event classes except the named event classes are enabled for synchronous event processing as described below. In the **estart** All, all event classes are enabled for synchronous event processing as described below.

When synchronous event processing is enabled for a given event class, events of that class will cause the execution of the registered event handler, if any, for that specific event (call back processing). Event classes are enabled by **estart** only for the process executing the **estart** command.

Call back processing can execute an **estart** command. In this case, the effect is to change the event classes which are enabled for subsequent synchronous event processing. **Estart** commands are not nested. It is not an error to issue a second **estart** command on the same event classes.

The execution of an **estart** command which starts synchronous event processing is terminated when an **estop** command is executed during call back processing for that **estart** command. When execution of an **estart** command which starts synchronous event processing is terminated, execution continues with the command following that **estart** command.

8.2.11 Estop

esto[**p**] *postcond* [*sp*]

The **estop** command implicitly performs the number of **quit** commands necessary to return to the execution level of the most recently executed **estart** command that started synchronous event processing, and then terminates that **estart** command. If synchronous event processing is not activated, execution of an **estop** command has no effect. It is not possible to **estop** only selected event classes.

8.2.12 Etrigger

et[**rigger**] *postcond* *sp* *especref*

$$\textit{especref} ::= \in\{\textit{expr} \rightarrow \left|\begin{array}{l}\texttt{\^{}\$W[INDOW](}\textit{espec}\texttt{)[:}\textit{einforef}\texttt{]}\\ \texttt{\^{}\$J[OB](}\textit{erspec}\texttt{)}\end{array}\right|\}$$

Note: *espec* and *einforef* should be as defined in *X11.6,* the MWAPI.

$$\begin{array}{rl}\textit{erspec} ::= & \textit{processid}\texttt{,"EVENT",}\in\{\textit{expr} \rightarrow \textit{evclass}\}\texttt{,}\in\{\textit{expr} \rightarrow \textit{evid}\}\\ \textit{evid} ::= & \textit{expr}\end{array}$$

Note that the range of values allowed for *evid* depends on the value of *evclass*, and may be implementation specific.

Etrigger causes an event to occur, though use of a *processid* other than the current job's own *processid* may be restricted by the implementation. This restricted use does not generate an error, but will not generate an event. Restrictions (if any) must be specified in the implementation's conformance statement.

If the use is not restricted and the specified event is enabled for either synchronous or asynchronous event processing, the event processing for it will occur subsequently. The event that occurs is specified by *evclass* and *evid*. If *evid* does not specify a valid event, an error condition occurs with *ecode*="**M103**".

If *evclass* evaluates to "**IPC**" and *evid* is not the current job's *processid*, an error condition occurs with *ecode*="**M104**".

8.2.13 For

$$\texttt{f[or]}\left|\begin{array}{l}[\textit{sp}]\\ \textit{sp}\ \ \textit{lvn}\texttt{=}\textit{list}\{\textit{forparameter}\}\end{array}\right|$$

$$\textit{forparameter} ::= \left|\begin{array}{l}\textit{expr}\\ \textit{numexpr}_1\texttt{:}\textit{numexpr}_2\texttt{:}\textit{numexpr}_3\\ \textit{numexpr}_1\texttt{:}\textit{numexpr}_2\end{array}\right|$$

The *scope* of the **for** command begins at the next *command* following the **for** on the same *line* and ends just prior to the *eol* on this *line*.

The **for** with arguments specifies repeated execution of the *commands* within its scope for different values of the local variable *lvn*, under successive control of the *forparameters*, from left to right. Any expressions occurring in *lvn*, such as might occur in subscripts or indirection, are evaluated once per execution of the `for`, prior to the first execution of any *forparameter*.

For each *forparameter*, control of the execution of the *commands* in the scope is specified as follows. (Note that `A`, `B`, and `C` are hidden temporaries.)

- a If the *forparameter* is of the form *expr*$_1$.
    - 1 Set *lvn*=*expr*.
    - 2 Execute the *commands* in the scope once.
    - 3 Processing of this *forparameter* is complete.
- b If the *forparameter* is of the form *numexpr*$_1$`:`*numexpr*$_2$`:`*numexpr*$_3$ and *numexpr*$_2$ is nonnegative.
    - 1 Set `A`=*numexpr*$_1$.
    - 2 Set `B`=*numexpr*$_2$.
    - 3 Set `C`=*numexpr*$_3$.
    - 4 Set *lvn*=`A`.
    - 5 If *lvn*>`C`, processing of this *forparameter* is complete.
    - 6 Execute the *commands* in the scope once.
    - 7 If *lvn*>`C-B`, processing of this *forparameter* is complete; an undefined value for *lvn* causes an error condition with *ecode*=`"M15"`.
    - 8 Otherwise, set *lvn*=*lvn*+`B`.
    - 9 Go to 6.
- c If the *forparameter* is of the form *numexpr*$_1$`:`*numexpr*$_2$`:`*numexpr*$_3$ and *numexpr*$_2$ is negative.
    - 1 Set `A`=*numexpr*$_1$.
    - 2 Set `B`=*numexpr*$_2$.
    - 3 Set `C`=*numexpr*$_3$.
    - 4 Set *lvn*=`A`.
    - 5 If *lvn*<`C`, processing of this *forparameter* is complete.
    - 6 Execute the *commands* in the scope once.
    - 7 If *lvn*<`C-B`, processing of this *forparameter* is complete; an undefined value for *lvn* causes an error condition with *ecode*=`"M15"`.
    - 8 Otherwise, set *lvn*=*lvn*+`B`.
    - 9 Go to 6.
- d If the *forparameter* is of the form *numexpr*$_1$`:`*numexpr*$_2$.
    - 1 Set `A`=*numexpr*$_1$.
    - 2 Set `B`=*numexpr*$_2$.
    - 3 Set *lvn*=`A`.
    - 4 Execute the *commands* in the scope once.
    - 5 Set *lvn*=*lvn*+`B`; an undefined value for *lvn* causes an error condition with *ecode*=`"M15"`.
    - 6 Go to 4.

If the `for` command has no argument:

a Execute the *commands* in the scope once; since no *lvn* has been specified, it cannot be referenced.

b Goto *a*.

Note that form *d* and the argumentless **for**, specify endless loops. Termination of these loops must occur by execution of a **quit** or **goto** within the scope of the **for**. These two termination methods are available within the scope of a **for** independent of the form of *forparameter* currently in control of the execution of the scope; they are described below. Note also that no *forparameter* to the right of one of form *d* can be executed.

Note that if the scope of a **for** (the *outer* **for**) contains an *inner* **for**, one execution of the scope of *commands* of the outer **for** encompasses all executions of the scope of *commands* of the inner **for** corresponding to one complete pass through the inner **for** command's *forparameter* list.

Execution of a **quit** within the scope of a **for** has two effects.

a It terminates that particular execution of the scope at the **quit**; *commands* to the right of the **quit** are not executed.

b It causes any remaining values of the *forparameter* in control at the time of execution of the **quit**, and the remainder of the *forparameters* in the same *forparameter* list, not to be calculated and the *commands* in the scope not to be executed under their control.

In other words, execution of **quit** effects the immediate termination of the innermost **for** whose scope contains the **quit**.

Execution of **goto** effects the immediate termination of all **for** commands in the *line* containing the **goto**, and it transfers execution control to the point specified. Note that the execution of an argumentless **quit** within the scope of a **for** does not affect the variable environment. Execution of an argumented **quit** within the scope of a **for** command causes an error condition with an *ecode*="**M16**".

8.2.14 Goto

g[oto] *postcond* *sp* *list*{*gotoargument*}

gotoargument ::= | *entryref* *postcond*
 @∈{*expratom* → *gotoargument*} |

Goto is a generalized transfer of control. If provision for a return of control is desired, **do** may be used.

Each *gotoargument* is examined, one at a time in left-to-right order, until the first one is found whose *postcond* is either absent, or whose *postcond* is present and its *tvexpr* is true. If no such *gotoargument* is found, control is not transferred and execution continues normally. If such a *gotoargument* is found, execution continues at the left of the *line* it specifies, provided that the following conditions hold for the *line* containing the **goto** and the *line* specified by the *gotoargument*:

a they have the same *level*, and

b if that level is greater than one then they

1 must have no *line*s of lower execution level between them, and

2 must be in the same routine.

If either *a* or *b* is not met, an error occurs with *ecode*=`"M45"`.

8.2.15 `Halt`

`h[alt]` *postcond* [*sp*]

If the value of **`$tlevel`** is greater than zero, a *rollback* is performed. In any case, all *nref*s are removed from the *lock-list* associated with this process. Finally, execution of this process is terminated.

8.2.16 `Hang`

`h[ang]` *postcond* *sp* `list`{*hangargument*}

hangargument `::=` | *numexpr*
 `@`∈{*expratom* → *hangargument*} |

Let `t` be the value of *numexpr*. If `t'>0`, **`hang`** has no effect. Otherwise, execution is suspended for `t` seconds.

8.2.17 `IF`

`I[F]` | [*sp*]
 sp *L* *ifargument* |

ifargument `::=` | *tvexpr*
 `@` *expratom* *V* *L* *ifargument* |

In its argumentless form, **`IF`** is the inverse of **`ELSE`**. That is, if the value of **`$TEST`** is `0`, the remainder of the *line* to the right of the **`IF`** is not executed. If the value of **`$TEST`** is `1`, execution continues normally at the next *command*. **`@`**∈{*expr* → *actualname*}

If exactly one argument is present, the value of *tvexpr* is placed into **`$TEST`**; then the function described above is performed.

`IF` with *n* arguments is equivalent in execution to *n* **`IF`** commands, each with one argument, with the respective arguments in the same order. This may be thought of as an implied *and* of the conditions expressed by the arguments.

8.2.18 JOB

`J[OB]` *postcond sp L jobargument*

jobargument `::=`	`[` *jobenv* `]` *entryref* `[ :` *jobparameters* `]`
	`[` *jobenv* `]` *labelref actuallist* `[ :` *jobparameters* `]`
	`@` *expratom V L jobargument*

jobenv `::=` *vb environment vb*

jobparameters `::=`	*processparameters* `[` *timeout* `]` *timeout*

processparameters `::=`	*expr* `( [ [` *expr* `] : ] ...` *expr* `)`

For each *jobargument*, the **JOB** command attempts to initiate another M process. If the *actuallist* is present in a *jobargument*, the *formalline* designated by *labelref* must contain a *formallist* in which the number of names is greater than or equal to the number of *expr*s in the *actuallist*. **@**∈{*expr* → *actualname*}

The **JOB** command initiates this process at the *line* specified by the *entryref* or *labelref*. There is no linkage between the started process and the process that initiated it. It is erroneous for a *jobargument* to contain a call-by-reference *actual* (*ecode*="**M40**"). If the *actuallist* is not present, the process will have no variables initially defined. (See 7.1.3.3 Process-Stack, and 8.1.8 Parameter passing).

The *processparameters* can be used in an implementation-specific fashion to indicate partition size, principal device, and the like.

If a *timeout* is present, the condition reported by **$TEST** is the success of initiating the process. If no *timeout* is present, the value of **$TEST** is not changed, and process execution is suspended until the process named in the *jobargument* is successfully initiated. The meaning of success in either context is defined by the implementation.

If *jobenv* is explicitly specified, the **JOB** command attempts to initiate this process in the environment specified by *jobenv*. Reference to a non-existent *jobenv* causes an error condition with an *ecode*="**M26**". If *jobenv* is not explicitly specified, then the value of **^$JOB($JOB,"JOB")** is used.

8.2.19 KILL

`K[ILL]`
`KV[ALUE]`
`KS[UBSCRIPTS]`
	postcond	`[` *sp* `]`
		sp L killargument

killargument `::=` | *glvn*
| `(` *L lname* `)`
| `@` *expratom V L killargument*

lname `::=` | *name*
| `@` *expratom V lname*

The three argument forms of **`KILL`**, **`KVALUE`**, and **`KSUBSCRIPTS`** are given the following names. **`@`**∈{*expr* → *actualname*}

a	*glvn*	Selective **`KILL`**
b	**`(`** *L lname* **`)`**	Exclusive **`KILL`**
c	Empty argument list	**`KILL`** All

`KILL`, **`KVALUE`**, and **`KSUBSCRIPTS`** are defined using a subsidiary function `K(V,val,subs)` where `V` is a *glvn*, `val` is `0` or `1`, and `subs` is `0` or `1`.

a Search for the *name* of `V` in the NAME-TABLE. If no such entry is found, the function is completed. Otherwise, extract the DATA-CELL pointer and proceed to step *b*.

b If `val=1` and `subs=1` then in the DATA-CELL identified in step `a`:

1 let `N` be the number of subscripts in `V`. If `V` is unsubscripted, let `N` be `0`.

2 If `N` is `0`, then delete all tuples. The function is completed.

3 Otherwise (if `N>0`), delete all tuples of degree `N` or greater whose first `N` subscripts are the same as those in `V`. The function is completed.

c If `val=1` and `subs=0` then in the DATA-CELL identified in step `a`:

1 If `V` is unsubscripted, delete the tupleof degree `0` (if found). The function is completed.

2 Otherwise, let `N` be the number of subscripts in `V`. Delete (if found) only the tuple of degree `N` whose first `N` subscripts are the same as those in `V`. The function is completed.

d If `val=0` and `subs=1` then in the DATA-CELL identified in step `a`:

1 Let `N` be the number of subscripts in `V`. If `V` is unsubscripted, let `N` be `0`.

2 Delete all tuples of degree `N+1` or greater whose first `N` subscripts are the same as those in `V`. The function is completed.

Note that as a result of procedure `K(V,1,1)`, `$DATA(V)=0`, i.e., the value of `V` is undefined, and `V` has no descendants.

Note that as a result of procedure `K(V,1,0)`, `$D(V)=0` if `V` had no descendents before procedure `K` was applied, or `$D(V)=10` if `V` had descendents before procedure `K` was applied, i.e., only the value of `V` is deleted.

Note that as a result of procedure `K(V,0,1)`, `$D(V)=1` if `V` had a value before procedure `K` was applied, or `$D(V)=0` if `V` had no descendents before procedure `K` was applied, i.e., only the descendents of `V` are deleted.

The actions of the three forms of **`KILL`** are then defined as:

a Selective **KILL** Apply procedure:
K(*glvn*,1,1), if **KILL**,
K(*glvn*,1,0), if **KVALUE**,
K(*glvn*,0,1), if **KSUBSCRIPTS**.

b Exclusive **KILL** For all names, *V*, in the locals NAME-TABLE except those in the argument list, apply procedure:
K(*V*,1,1), if **KILL**,
K(*V*,1,0), if **KVALUE**,
K(*V*,0,1), if **KSUBSCRIPTS**.
Note that the names in the argument list of an Exclusive **KILL** may not are restricted to unsubscripted locals.

c **KILL** All For all names, *V*, in the locals NAME-TABLE, apply procedure:
K(*V*,1,1), if **KILL**,
K(*V*,1,0), if **KVALUE**,
K(*V*,0,1), if **KSUBSCRIPTS**.
Note that **KILL** All applies procedure *K* to the local variable NAME-TABLE only.

If a variable *N*, a descendant of *M*, is killed, the killing of *N* affects the value of **$DATA(*M*)** as follows: if *N* was not the only descendant of *M*, **$DATA(*M*)** is unchanged; otherwise, if *M* has a defined value, **$DATA(*M*)** is changed from 11 to 1; if *M* does not have a defined value, **$DATA(*M*)** is changed from 10 to 0.

8.2.20 LOCK

```
L[OCK] postcond | [ sp ]               |
                | sp L lockargument    |

lockargument ::= | [ + ] | nref         | [ timeout ] |
                 | [ - ] | ( L nref )   |             |
                 | @ expratom V L lockargument        |

nref ::= | rnrefind              |
         | @ expratom V nref     |

rnref ::= | [ ^ ] [ vb environment vb ] name [ ( L expr ) ] |
          | @ nrefind @ ( L expr )                          |

nrefind ::= rexpratom V nref
```

LOCK provides a generalized interlock facility available to concurrently executing M processes to be used as appropriate to the applications being programmed. Execution of

LOCK is not affected by, nor does it directly affect, the state or value of any global or local variable, or the value of the naked indicator. Its use is not required to access globals, nor does its use inhibit other processes from accessing globals. It is an interlocking mechanism whose use depends on programmers establishing and following conventions. **@**∈{*expr* → *actualname*}

An *nref* either unsubscripted or subscripted; if it is subscripted, any number of subscripts separated by commas is permitted.

When *nrefind* is present it is always a component of an *rnref*. If the value of the *rnref* is a subscripted form of *nref* then some of its subscripts may have originated in the *nrefind*. In this case, the subscripts contributed by the *nrefind* appear as the first subscripts in the value of the resulting *rnref*, separated by a comma from the (non-empty) list of subscripts appearing in the rest of the *rnref*.

Each *lockargument* specifies a subspace of the total M LOCK-UNIVERSE for the *environment* upon which the executing process seeks to make or release an exclusive claim; the details of this subspace specification are given below.

A special space for the lockspace is needed to create a synchronization mechanism for the executing process for each of the *environments* referenced by the executing process. A *timeout* refers to the time spent at the target *environment*, any time delays due to communication delays are not part of the *timeout*.

For the purposes of this discussion, the LOCK-UNIVERSE is defined as the union of all possible *nrefs* in one *environment* after resolution of all indirection. Further, there exists for each process a LOCK-LIST that contains zero or more *nrefs*. Execution of *lockarguments* has the effect of adding or removing *nrefs* from the process' LOCK-LIST. A given *nref* may appear more than once within the LOCK-LIST. The *nrefs* in the LOCK-LIST specify a subset of the LOCK-UNIVERSE. This subspace, called the process' LOCKSPACE, consists of the union of the subspaces specified by all *nrefs* in the LOCK-LIST, as follows:

a If the *nref* is unsubscripted, then the subspace is the set of the following points: one point for the unsubscripted variable name *nref* and one point for each subscripted variable name $N(s_1,\ldots,s_i)$ where N has the same spelling as *nref*.

b If the occurrence of *nref* is subscripted, let the *nref* be $N(s_1,s_2,\ldots,s_n)$. Then the subspace is the set of the following points: one point for $N(s_1,s_2,\ldots,s_n)$ and one point for each descendant (see 7.1.6.3 **$DATA** function for a definition of descendant) of *nref*.

If the **LOCK** command is argumentless, **LOCK** removes all *nrefs* from the LOCK-LIST associated with this process.

Execution of *lockargument* occurs in the following order:

a Any expression evaluation involved in processing the *lockargument* is performed.

b If the form of *lockargument* does not include an initial **+** or **-** sign, then prior to evaluating or executing the rest of the *lockargument*, **LOCK** first removes all *nrefs* from the LOCK-LIST associated with this process. Then it appends each of the *nrefs* in the *lockargument* to the process' LOCK-LIST.

c If the *lockargument* has a leading **+** sign, **LOCK** appends each of the *nref*s in the *lockargument* to the process' LOCK-LIST.

d If the *lockargument* has a leading **-** sign, then for each *nref* in the *lockargument*, if the *nref* exists in the LOCK-LIST for this process, one instance of *nref* is removed from the LOCK-LIST.

An error occurs, with *ecode*=**"M41"**, if a process within a TRANSACTION attempts to remove from its LOCK-LIST any *nref* that was present when the TRANSACTION started. With respect to each other process, the effect of removing any *nref* from the LOCK-LIST is deferred until the global variable modifications made since that *nref* was added to the LOCK-LIST are available to that other process.

LOCK affects concurrent execution of processes having LOCK-SPACES that OVERLAP. Two LOCK-SPACEs OVERLAP when their intersection is not empty. **LOCK** imposes the following constraints on the concurrent execution of processes:

a The LOCK-SPACEs of any two processes executing *command*s outside the scope of a TRANSACTION may not OVERLAP.

b All global variable modifications produced by the execution of *commands* by processes having LOCK-SPACEs that OVERLAP must be equivalent to the modifications resulting from some execution schedule during which their LOCK-SPACEs do not OVERLAP.

See the TRANSACTION Processing subclause for the definition of TRANSACTION.

The constraints imposed by **LOCK** on the execution of processes having LOCK-SPACEs that OVERLAP may cause execution of one or more processes to be delayed. The maximum duration of such a delay may be specified with a *timeout*.

If present, *timeout* modifies the execution of **LOCK**, described above, as follows:

a If execution of the process is delayed and cannot be resumed prior to the expiration of *timeout*, then the execution of the *lockargument* is unsuccessful. In this event the value of **$TEST** is set to zero and any *nref*s added to the LOCK-LIST as a result of executing the *lockargument* are removed.

b Otherwise, the execution of the *lockargument* is successful and **$TEST** is set to one.

If no *timeout* is present, then the value of **$TEST** is not affected by execution of the *lockargument*.

8.2.21 Merge

m[erge] *postcond* *sp* *list*{*mergeargument*}

mergeargument ::= | $glvn_1$**=**$glvn_2$
 @∈{*expratom* → *mergeargument*} |

Merge provides a facility to copy a $glvn_2$ into a $glvn_1$ and all descendants of $glvn_2$ into descendants of $glvn_1$ according to the scheme described below.

Merge does not **kill** any nodes in $glvn_1$, or any of its descendants.

Assume that *glvn*$_1$ is represented as $A(i_1,i_2,\ldots,i_x)$ (x'<0) and that *glvn*$_2$ is represented as $B(j_1,j_2,\ldots,j_y)$ (y'<0).

Then:

a If `$data`($B(j_1,j_2,\ldots,j_y)$) has a value of `1` or `11`, then the value of *glvn*$_2$ is given to *glvn*$_1$.

b The value for every occurrence of z, such that z>0 and `$data`($B(j_1,j_2,\ldots,j_{y+z})$) has a value of `1` or `11`, the value of $B(j_1,j_2,\ldots,j_{y+z})$ is given to $A(i_1,i_2,\ldots,i_x,j_{y+1},j_{y+2},\ldots,j_{y+z})$.

The state of the naked indicator will be modified as if `$data`(*glvn*$_2$)`#10=1` and the command `set` *glvn*$_1$=*glvn*$_2$ would have been executed.

If *glvn*$_1$ is a descendant of *glvn*$_2$ or if *glvn*$_2$ is a descendant of *glvn*$_1$ an error condition occurs with *ecode*=`"M19"`.

8.2.22 NEW

`N[EW]` *postcond* | `[` *sp* `]`
| *sp* *L* *newargument* |

newargument `::=` | *lname*
newsvn
`(` *L* *lname* `)`
`@` *expratom* *V* *L* *newargument* |

newsvn `::=` | `$ET[RAP]`
`$ES[TACK]`
`$R[EFERENCE]`
`$T[EST]` |

`NEW` provides a means of performing variable scoping. **`@`**∈{*expr* → *actualname*}

The three argument forms of **`NEW`** are given the following names:

a	*lname*	Selective **`NEW`**
b	`(`*L lname*`)`	Exclusive **`NEW`**
c	Empty argument list	**`NEW`** All
d	*newsvn*	**`NEW`** *svn*

The following discussion uses terms defined in the Variable Handling (see 7.1.3.2) and Process-Stack (see 7.1.3.3) models and, like those subclauses, does not imply a required implementation technique. Each argument of the **`NEW`** command creates a CONTEXT-STRUCTURE consisting of a NEW NAME-TABLE and an exclusive indicator, attaches it to a linked list of CONTEXT-STRUCTUREs associated with the current PROCESS-STACK frame, and modifies currently active NAME-TABLEs as follows:

a **`NEW`** All marks the CONTEXT-STRUCTURE as exclusive, copies the currently active NAME-TABLE to the NEW NAME-TABLE and makes all entries in the currently active local variable NAME-TABLE point to empty DATA-CELLs.

b Exclusive **NEW** marks the CONTEXT-STRUCTURE as exclusive, copies the currently active NAME-TABLE to the NEW NAME-TABLE and changes all entries in the currently active local variable NAME-TABLE, except for those corresponding to names specified by the command argument, to point to empty DATA-CELLs.

c Selective **NEW** copies the entry corresponding to the name specified by the *command* argument to the NEW NAME-TABLE and makes that entry in the currently active NAME-TABLE point to an empty DATA-CELL.

d **NEW** svn copies the entry corresponding to the name specified by the *command* argument to the NEW NAME-TABLE and updates that entry as follows:

1 if the argument specifies **$ESTACK**, points to a DATA-CELL with a value of `0` (zero).

2 if the argument specifies **$ETRAP**, points to a DATA-CELL with a value copied from the prior DATA-CELL (as pointed to by the just-copied NAME-TABLE entry).

3 If the argument specifies **$R[EFERENCE]**, points to a DATA-CELL with a value copied from the prior DATA-CELL (as pointed to by the just-copied NAME-TABLE entry).

4 if the argument specifies **$TEST**, points to a DATA-CELL with a value copied from the prior DATA-CELL (as pointed to by the just-copied NAME-TABLE entry).

8.2.23 OPEN

`O[PEN]` *postcond sp L openargument*

```
openargument ::=    | devn [ : openparameters ]      |
                    | @ expratom V L openargument    |

openparameters ::=  | deviceparameters [ timeout [ : mnemonicspec ] ] |
                    | [ deviceparameters ] :: mnemonicspec            |
                    | timeout [ : mnemonicspec ]                      |

mnemonicspec ::=    | mnemonicspace          |
                    | ( L mnemonicspace )    |

mnemonicspace ::= expr V mnemonicspacename
```

$$\textit{mnemonicspacename} \ ::= \ \textit{ident} \ \left[\begin{array}{l} \textit{ident} \\ \textit{digit} \\ . \\ - \end{array}\right] \ldots$$

(note: hyphen)

There is a large overlap in specification between the commands **OPEN**, **USE**, and **CLOSE**. As a side-effect of the alphabetical ordering of the commands, many features are described in clause 8.2.7 **CLOSE**. As a matter of style in this document, these features are not repeated in this clause. See 8.2.7 for the syntax and interpretation of *devn* and *deviceparameters*.
@∈{*expr* → *actualname*}

The **OPEN** command is used to obtain ownership of a device, and does not affect which device is the current device or the value of **$IO**. (see the discussion of **USE** in 8.2.34)

For each *openargument*, the **OPEN** command attempts to seize exclusive ownership of the specified device. **OPEN** performs this function effectively instantaneously as far as other processes are concerned; otherwise, it has no effect regarding the ownership of devices and the values of the device parameters.

If a *timeout* is present, the condition reported by **$TEST** is the success of obtaining ownership. If no *timeout* is present, the value of **$TEST** is not changed and process execution is suspended until seizure of ownership has been successfully accomplished by the process that issued the **OPEN** *command*.

Upon establishing ownership of a device, any parameter for which no specification is present in the *openparameters* is given the value most recently used for that device; if none exists, an implementor-defined default value is used.

In the case that a process has successfully executed an **OPEN** command for a certain device and has established certain operational parameters for that device, and subsequently the same process makes an attempt to execute an **OPEN** command for the same device while specifying different operational parameters, those established operational parameters that are controlled by the implementation, and for which new values are supplied, will be discarded, and an attempt will be made to establish the newly specified parameters as the current ones for the device in question.

Ownership is relinquished by execution of the **CLOSE** command. When ownership is relinquished, all device parameters are retained.

Mnemonicspace specifies the set of *controlmnemonic*s that may be used within *format* arguments to subsequent **READ** and **WRITE** commands. The *mnemonicspace* may be an empty string and may not provide any defined *controlmnemonic*s. *Mnemonicspacename*s that start with any character other than **Y** or **Z** are reserved for *mnemonicspace* definitions registered by the MDC; those that start with **Z** are implementor-specific.

When a *mnemonicspec* contains a list of *mnemonicspace*s, the first one determines the active *mnemonicspace*, which may be changed by a **USE** command. If the device does not support any *mnemonicspace* of a *mnemonicspec*, an error condition occurs with *ecode*="**M35**". If any *mnemonicspace*s in the *mnemonicspec* are incompatible, an error occurs with *ecode*="**M36**".

In addition to *controlmnemonic*s a *mnemonicspace* also defines the valid *deviceattribute*s and *devicekeyword*s which are associated with a device. *Deviceattribute*s and *devicekeyword*s which start with the character **Z** are implementor-specific. Associated with each *deviceattribute* are one or more values which are held in the *ssvn* **^$DEVICE**.

The **^***routineref* alternative is a user-defined *mnemonicspace* and associates the *routine* named in *routineref*$_1$ with the locatlon of code to be executed when a *controlmnemonic* is used.

The user-defined *mnemonicspace* command routine is the *routine* defined in *routineref*$_2$, or if absent in *routineref*$_1$. It associates this *routine* with the location of code to be executed when a *command* is used in conjunction with the *mnemonicspace*.

If an implementation does not provide for the use of a specific *mnemonicspace* then that implementation shall provide a mechanism by which to associate a *routineref* with this *mnemonicspace*. All subsequent references to this *mnemonicspace* are handled as if this were a user-defined *mnemonicspace*.

8.2.24 Quit

	[*sp*]
q[uit] *postcond*	*sp* *expr*
	sp **@**∈{*expratom* → *expr*}

Quit terminates execution of an argumentless **do** command, *doargument*, *xargument*, *exfunc*, *exvar*, or **for** command.

Encountering the end-of-routine mark *eor* is equivalent to an unconditional argumentless **quit**.

The effect of executing **quit** in the scope of **for** is fully discussed in 8.2.13. Note the *eor* never occurs in the scope of **for**.

If an executed **quit** is not in the scope of **for**, then it is in the scope of some argumentless **do** command, *doargument*, *xargument*, *exfunc*, or *exvar* if not explicitly then implicitly, because the initial activation of a process, including that due to execution of a *jobargument*, may be thought of as arising from execution of a **do** naming the first executed routine of that process.

The effect of executing a **quit** in the scope of an argumentless **do** command, *doargument*, *xargument*, *exfunc*, or *exvar* is to restore the previous variable environment (if necessary), restore the value of **$test** (if necessary), restore the previous execution level, and continue execution at the location of the invoking argumentless **do** command, *doargument*, *xargument*, *exfunc*, or *exvar*.

If the *expr* is present in the **quit** and the return is not to an *exfunc* or *exvar*, an error condition occurs with *ecode*=**"M16"**. If the *expr* is not present and the return is to an *exfunc* or *exvar*, an error condition occurs with *ecode*=**"M17"**.

The following discussion uses terms defined in the Variable Handling (see 7.1.3.2) and

Process-Stack (see 7.1.3.3) models and, like those subclauses, does not imply a required implementation technique.

Execution of a **quit** occurs as follows:

a If an *expr* is present, evaluate it. If the resulting value is a value of data type *oref*, do not coerce this value into a value of data type *mval*. This value becomes the value of the invoking *exfunc* or *exvar*.

b Remove the frame on the top of the *process-stack*. If no such frame exists, then execute an implicit **halt**.

c If the *process-stack* frame's linked list of *context-structure*s contains *new name-table*s, process them in last-in-first-out order from their creation. If the *context-structure* is exclusive, make all entries in the currently active local variable *name-table* point to empty *data-cell*s. In all cases, the *new name-table*s are copied to the currently active *name-table*s. Note that, in the model, **quit** never encounters any restart *context-structure*s in the linked list because they must have been removed by **tcommit**s or *rollback*s for the **quit** to reach this point in its execution.

d If the frame contains formal list information, extract the *formallist* and process each *name* in the list with the following steps:

  1 Search the *name-table* for an entry containing the name. If no such entry is found, processing of this *name* is complete. Otherwise, proceed to step 2.

  2 Delete the *name-table* entry for this *name*.

  Finally, copy all *name-table* entries from this frame into the *name-table*.

  Processing of this frame is complete, continue at step *b*.

e If the frame is a **tstart** frame and **$tlevel** is greater than zero, **quit** generates an error with *ecode*=**"M42"**. If the frame is a **tstart** frame and **$tlevel** is zero, then the frame is discarded.

f If the frame is from an *exfunc* or *exvar* or from an argumentless **do** command, set the value of **$test** to the value saved in the frame.

g Restore the execution level and continue execution at the location specified in the frame. However, if this location is in a routine which has been modified or made inaccessible by the execution of a **rsave** command (subsequent to the placing of the frame on the *process-stack*), unspecified behavior may result.

8.2.25 READ

R[EAD] *postcond* *sp* *L* *readargument*

readargument ::=
| *strlit*
| *format*
| *glvn* [*readcount*] [*timeout*]
| * *glvn* [*timeout*]
| @ *expratom* *V* *L* *readargument*

readcount `::=` `#` *intexpr*

The *readargument*s are executed, one at a time, in left-to-right order. **@**∈{*expr* → *actualname*}

The forms *strlit* and *format* cause output operations to the current device; the forms *glvn* and *****_glvn_ cause input from the current device to the named variable (see 7.1.3.4 for a description of the value assignment operation). If no *timeout* is present, execution will be suspended until the input message is terminated, either explicitly or implicitly with a *readcount*. (See 8.2.34 for a definition of *current device.*)

If a *timeout* is present, it is interpreted as a `t`-second timeout, and execution will be suspended until the input message is terminated, but in any case no longer than `t` seconds. If `t'>0`, `t=0` is used.

When a *timeout* is present, **`$TEST`** is affected as follows. If the input message has been terminated at or before the time at which execution resumes, **`$TEST`** is set to `1`; otherwise, **`$TEST`** is set to `0`.

When the form of the argument is ***** *glvn* **[** *timeout*], the input message is by definition one character long, and it is explicitly terminated by the entry of one character, which is not necessarily from the ASCII set. The value given to *glvn* is an integer; the mapping between the set of input characters and the set of integer values given to *glvn* may be defined by the implementor in a device-dependent manner. If *timeout* is present and the timeout expires, *glvn* is given the value `-1`.

When the form of the argument is *glvn* [*timeout*], the input message is a string of arbitrary length which is terminated by an implementor-defined procedure, which may be device-dependent. If *timeout* is present and the timeout expires, the value given to *glvn* is the string entered prior to expiration of the timeout; otherwise, the value given to *glvn* is the entire string.

When the form of the argument is *glvn* **#** *intexpr* [*timeout*], let `n` be the value of *intexpr*. If `n'>0` an error condition occurs with *ecode*=`"`**`M18`**`"`. Otherwise, the input message is a string whose length is at most `n` characters, and which is terminated by an implementor-defined, possibly device-dependent procedure, which may be the receipt of the `n`th character. If *timeout* is present and the timeout expires prior to the termination of the input message by either mechanism just described, the value given to *glvn* is the string entered prior to the expiration of the timeout; otherwise, the value given to *glvn* is the string just described.

When it has been specified that the current device is able to send control-sequences according to some *mnemonicspace*, the **`READ`** will be terminated as soon as such a control-sequence has been entered (be it by typing a function-key or by some other internal process within the device). The value of the specified *glvn* will be the same as if instead of the control-sequence the usual terminator-character would have been received before the control-sequence was sent.

When the form of the argument is *strlit*, it is equivalent to **`WRITE`** *strlit*. When the form of the argument is *format*, it is equivalent to **`WRITE`** *format*.

`$X` and **`$Y`** are affected by **`READ`** the same as if the command were **`WRITE`** with the same argument list (except for *timeout*s and *readcount*s) and with each *expr* value in each *writeargument* equal, in turn, to the final value of the respective *glvn* resulting from the **READ**.

Input operations, except when the form of the argument is ***** *glvn* [*timeout*], are affected by the Character Set Profile input-transform. Output operations are affected by the Character Set Profile output-transform. (see 7.1.4.1 **^$CHARACTER**)

8.2.26 RLOAD

RL[OAD] *postcond* *sp* *L* *routineargument*

routineargument ::=
| *routineref* : *glvn* [: *routineparameters*] |
| @ *expratom* *V* *L* *routineargument* |

routineparameters ::=
| *routineparam* |
| ([[*routineparam*] :] ... *routineparam*) |

routineparam ::=
| *routinekeyword* |
| *routineattribute* = *expr* |

routinekeyword ::= *name*

routineattribute ::= *name*

Spellings of *routinekeyword* and *routineattribute* differing only in the use of lowercase and uppercase letters are equivalent. **@**∈{*expr* → *actualname*}

All values of *routinekeyword* and *routineattribute* not starting with the character "**Z**" are reserved for the MDC.

routinekeywords are processed in strict left-to-right order. When multiple equivalent *routinekeywords* are encountered, the last occurrence processed will define the action(s) to be taken.

Assume that *glvn* is represented as $A(i_1,i_2,\ldots,i_x)$ (x'<0).

Then the lines of the *routine* denoted by *routineref* are stored in nodes $A(i_1,i_2,\ldots,i_x,i_{x+1})$. i_{x+1} has a value of *n* for the n^{th} line of the *routine* for all *lines* of the *routine* and no other nodes of A within the subscript range $i_1 \ldots i_{x+1}$ will be affected.

The naked indicator is modified by the reference to *glvn* if it is a *gvn*, but not by the implicit reference to the immediate descendants of *glvn*.

If the *routineref* denotes a non-existent *routine* an error condition occurs with *ecode*="**M88**".

8.2.27 Rsave

rs[ave] *postcond* *sp* *list*{*routineargument*}

Routinekeywords are processed in strict left-to-right order. When multiple equivalent *routinekeywords* are encountered, the last occurrence processed will define the action(s) to be taken.

Assume that *glvn* is represented as $A(i_1,i_2,\ldots,i_x)$ (`x'<0`)

Then the data values of all nodes $A(i_1,i_2,\ldots,i_x,i_{x+1})$ for which the value of **`$data`** is either `1` or `11` are stored as *lines* of the *routine* denoted by *routineref*. The *lines* are taken in the subscript ordering for i_{x+1} as specified in the definition of **`$order`** (7.1.6.12)

If *glvn* is undefined or if no node $A(i_1,i_2,\ldots,i_x,i_{x+1})$ with a **`$data`** value of `1` or `11` exists the *routine* denoted by *routineref* is deleted.

If any one of the *lines* denoted by $A(i_1,i_2,\ldots,i_x,i_{x+1})$ does not conform to the definition of a *line* the effect of executing **`rsave`** is unspecified.

At no point during the execution of **`rsave`** will any process be able to see a partially-filed *routine*.

Execution of **`rsave`** where *routineref* names the currently-executing *routine* causes an error with *ecode*=`"M25"`, and the *routine* is not modified.

The naked indicator is modified by the reference to *glvn* if it is a *gvn*, but not by the implicit reference to the immediate descendants of *glvn*.

8.2.28 Set

`s[et]` *postcond* *sp* *list*{*setargument*}

setargument ::= | *setdestination*=*expr*
| **@**∈{*expratom* → *setargument*}

setdestination ::= | *setleft*
| **(***list*{*setleft*}**)**

setleft ::= | *leftrestricted*
| *leftexpr*
| *glvn*
| *owproperty*

leftrestricted ::= | **`$d[evice]`**
| **`$k[ey]`**
| **`$r[eference]`**
| **`$x`**
| **`$y`**

leftexpr ::= | *setpiece*
 setextract
 setev
 setqsub
 setdextract
 setdpiece |

setpiece ::= **$p**[**iece**]**(***glvn***,***expr*$_1$[**,***intexpr*$_1$[**,***intexpr*$_2$]]**)**

setextract ::= **$e**[**xtract**]**(***glvn*[**,***intexpr*$_1$[**,***intexpr*$_2$]]**)**

setev ::= | **$ec**[**ode**]
 $et[**rap**] |

setqsub ::= **$qs**[**ubscript**]**(***glvn***,***intexpr***)**

setdextract ::= **$de**[**xtract**]**(***extracttemplate***,***list*[*recordfieldglvn*]**)**

setdpiece ::= **$dp**[**iece**]**(***piecedelimiter***,***list*[*recordfieldglvn*]**)**

recordfieldglvn ::= *glvn***:**[*fieldindex*]

Set is the general means both for explicitly assigning values to variables, and for substituting new values in pieces of a variable. Each *setargument* computes one value, defined by its *expr*. That value is then either assigned to each of one or more variables, or it is substituted for one or more pieces of a variable's current value. Each variable is named by one *glvn*.

Each *setargument* is executed one at a time in left-to-right order. If the portion of the *setargument* to the left of the **=** does not consist of **$x** or **$y** then the execution of a *setargument* occurs in the following order.

- a One of the following two operations is performed:
  - 1 If the portion of the *setargument* to the left of the **=** consists of one or more *glvn*s, the *glvn*s are scanned in left-to-right order and all subscripts are evaluated, in left-to-right order within each *glvn*.
  - 2 If the portion of the *setargument* to the left of the **=** consists of a *setpiece* or a *setextract* or a *setqsub*, the *glvn* that is the first argument of the *setpiece* or a *setextract* or a *setqsub* is scanned in left-to-right order and all subscripts are evaluated in left-to-right order within the *glvn*, and then the remaining arguments of the *setpiece* or a *setextract* or a *setqsub* are evaluated in left-to-right order.
- b The *expr* to the right of the **=** is evaluated. For each *setleft*, if it is a *leftrestricted*, the value to be assigned or replaced is truncated or converted to meet the inherent restrictions for that *setleft* before the assignment takes place. This means that in one **set** *command*, the various *setleft*s may receive different values.
- c One of the following eight operations is performed.
  - 1 If the left-hand side of the set is one or more *glvn*s, the value of *expr* is given to each *glvn*, in left-to-right order. (See 7.1.3.2 for a description of the value assignment operation).
  - 2 For each *setleft* that is a *setpiece*, of the form **$piece(***glvn***,***d***,***m***,***n***)**, the value of

expr replaces the m^{th} through the n^{th} pieces of the current value of the *glvn*, where the value of `d` is the piece delimiter. Note that both `m` and `n` are optional. If neither is present, then `m=n=1`; if only `m` is present, then `n=m`. If *glvn* has no current value, the empty string is used as its current value. Note that the current value of *glvn* is obtained just prior to replacing it. That is, the other arguments of *setpiece* are evaluated in left-to-right order, and the *expr* to the right of the **`=`** is evaluated prior to obtaining the value of *glvn*.

Let `s` be the current value of *glvn*, `k` be the number of occurrences of `d` in `s`, that is, `k=max(0,$length(s,d)-1)`, and `t` be the value of *expr*. The following cases are defined, using the concatenation operator **`_`** of 7.2.1.1:

a `m>n` or `n<1` The *glvn* is not changed and does not change the naked indicator.

b `n'<m-1>k` The value in *glvn* is replaced by `s_F(m-1-k)_t`, where `F(x)` denotes a string of `x` occurrences of `d`, when `x>0`; otherwise, `F(x)=""`. In either case, *glvn* affects the naked indicator.

c `m-1'>k<n` The value in *glvn* is replaced by `$piece(s,d,1,m-1)_F(min(m-1,1))_t`.

d Otherwise The value in *glvn* is replaced by `$piece(s,d,1,m-1)_F(min(m-1,1))_t_d_$piece(s,d,n+1,k+1)`.

3 For each *setleft* that is a *setextract* of the form **`$extract(`***glvn***`,`***m***`,`***n***`)`**, the value of *expr* replaces the m^{th} through the n^{th} characters of the current value of the *glvn*. Note that both `m` and `n` are optional. If neither is present, then `m=n=1`; if only `m` is present, then `n=m`. If *glvn* has no current value, the empty string is used as its current value. Note that the current value of *glvn* is obtained just prior to replacing it. That is, the other arguments of *setextract* are evaluated in left-to-right order, and the *expr* to the right of the **`=`** is evaluated prior to obtaining the value of *glvn*.

Let `s` be the current value of *glvn*, `k` be the number of characters in `s`, that is, `k=$length(s)`, and `t` be the value of *expr*. The following cases are defined, using the concatenation operator **`_`** of 7.2.1.1:

a `m>n or n<1` The *glvn* is not changed and does not change the naked indicator.

b `n'<m-1>k` The value in *glvn* is replaced by `s_$justify("",m-1-k)_t`.

c `m-1'>k<n` The value in *glvn* is replaced by `$extract(s,1,m-1)_t`.

d Otherwise The value in *glvn* is replaced by `$extract(s,1,m-1)_t_$extract(s,n+1,k)`.

In cases *b*, *c*, and *d* the naked indicator is affected.

4 If the left-hand side of the **`set`** is a *setev*, one of the following two operations is performed:

a If the *setev* is **`$ecode`**:

If the value of the *expr* is the empty string:

1 The current value of **`$ecode`** is replaced by the empty string.

2 All forms of the two-argument function **`$stack($stack+n,…)`** return the empty string for all values of *n*>`0`.
3 All forms of the function **`$stack($stack+n)`** return the empty string for all values of *n*>`0`.

If the value of *expr* is not the empty string:

1 If the value of *expr* does not conform to format required in section 7.1.5.10 for **`$ecode`**, the **`set`** of **`$ecode`** to the value of the *expr* is not performed. An `"M101"` error is generated instead.
2 If the value of *expr* does conform to format required in section 7.1.5.10 for **`$ecode`**:
   a The current value of **`$ecode`** is replaced by the value of *expr*.
   b The value of **`$stack($stack,"ECODE")`** is replaced by the value of *expr*.
   c The value of **`$stack($stack,"PLACE")`** is replaced to reflect the **`set`** command which is updating **`$ecode`**.
   d The value of **`$stack($stack,"MCODE")`** is replaced to reflect the **`set`** command which is updating **`$ecode`**.
   a An error trap is invoked.

b If the *setev* is **`$etrap`**, the current value of **`$etrap`** is replaced by the value of *expr*.

5 For each *setleft* that is a *setqsub* of the form **`$qsubscript(nv,m)`**, if the value of *nv* is not a valid *namevalue*, an error condition occurs with *ecode*=`"M90"`. Otherwise, let *t* be the value of *expr* and *nv* in the form *NAME*$(s_1,s_2,\ldots,s_n)$, considering *n* to be zero if there are no subscripts, is modified according to the value of *intexpr m* as follows:

a Values of *m* less than `-1` are reserved for possible future use by the MDC.
b If *m*=`-1`, the *environment* is changed to *t*.
c If *m*=`0`, the *name* is changed to *t*.
d If *m*>n, the intervening *n*+`1` through *m*-`1` subscripts are each set to the empty string and the m^{th} subscript is set to *t*.
e Otherwise, the m^{th} subscript is changed to *t*.

If the resulting value of *nv* is not a valid *namevalue*, an error condition occurs with *ecode*=`"M90"`.

Note that the original and resulting *namevalue*s are not “executed,” and will not modify the naked indicator beyond those modifications described at the end of this clause. Note also that the *namevalue*s, while meeting the syntax of a *namevalue*, might specify a non-existent *environment* or contain a subscript value (such as the empty string or control characters) which do not meet the requirements of Section II Clause 2.3.3 (Values of subscripts).

6 For each *setleft* that is a *setdextract*, the *expr* is used as the starting value, which is partitioned into consecutive **`$extract`** fields using *extracttemplate* (see definition of **`$dextract`** section 7.1.6.4). Each *glvn* is assigned its corresponding field

extracted from *expr*. The values corresponding to omitted *glvn*s are ignored. The *fieldindex* specifies which field is to be assigned to the *glvn*. If omitted, the next successive field index is assigned. Although *recordfieldglvn* is optional, at least one *recordfieldglvn* (not necessarily the first) in the list must be nonempty.

7 For each *setleft* that is a *setdpiece* the *expr* is used as the starting value, which is partitioned into consecutive **`$piece`** fields using *piecedelimiter* (see definition of **`$dpiece`** section 7.1.6.5). Each *glvn* is assigned its corresponding field pieced from *expr*. The value corresponding to omitted *glvn* are ignored. The *fieldindex* specifies which field is to be assigned to the *glvn*. If omitted, the next successive field index is assigned. Although *recordfieldglvn* is optional, at least one *recordfieldglvn* (not necessarily the first) in the list must be nonempty.

8 If the left-hand side of the **`set`** is at *owproperty*, the value of the *expr* is given to the *owproperty*.

If the portion of the *setargument* to the left of the **`=`** is a **`$x`** or a **`$y`** then the execution of the *setargument* occurs in the following order:

a The *intexpr* to the right of the **`=`** is evaluated.

b The value of the *intexpr* is given to the special intrinsic variable on the left of the **`=`** with the following restrictions and affects:

1 The range of values of **`$x`** and **`$y`** are defined in 7.1.5.10. Any attempt to set **`$x`** or **`$y`** outside this range specified in 7.1.5.10 is erroneous (*ecode*=**`"M43"`**) and the value of **`$x`** or **`$y`** will remain unchanged.

2 Setting **`$x`** or **`$y`** changes the value of **`$x`** or **`$y`**, respectively, but it does not cause any input or output operation. The purpose is to allow a program to correct the value of **`$x`** or **`$y`** following input or output operations whose effect on the cursor position may not be reflected in **`$x`** and **`$y`**.

The value of the naked indicator may be modified as a side-effect of the execution of a **`set`** *command*. Events that influence the value of the naked indicator are (in order of evaluation):

1 references to *glvn*s in *expr*s in arguments or subscripts of *setleft*s;

2 references to *glvn*s in the *expr* on the righthand side of the **`=`** sign;

3 references to *glvn*s in the *setdestination*.

8.2.29 `Tcommit`

`tc`[**`ommit`**] *postcond* [*sp*]

If **`$tlevel`** is one, **`tcommit`** performs a *commit* of the *transaction* and sets **`$trestart`** to zero. (See the Transaction Processing subclause for the definition of *commit*).

If **`$tlevel`** is greater than one, **`tcommit`** subtracts one from **`$tlevel`**.

If $**`tlevel`** is zero, **`tcommit`** generates an error with *ecode*=**`"M44"`**.

Using the (model) linked list of *restart context-structure*s for the *transaction*, *tcommit* removes the last created *restart context-structure* from both the *process-stack* linked list and the *transaction* linked list and discards the *restart context-structure*.

8.2.30 Then

th[**en**] [*sp*]

This *command* creates a new *context-structure* consisting of a *new name-table* and attaches it to a linked list of *context-structure*s associated with the current *process-stack* frame, and modifies currently active *name-table*s equivalent to **new** *svn* for the *svn* **$test**. The value of **$test** is restored from this *context-structure* (and the *context-structure* is removed unless otherwise indicated) by any of the following actions (note “current line” refers to the line with the **then** command):

a Execution encounters the *eol* at the end of the current *line*.
b A **quit** *command* is encountered at the current execution level (note this includes *restarts*).
c A **quit** *command* returns execution to the current execution level (but does not remove the *context-structure*).
d An explicit **goto** *command*, located in the current *line*, is executed.

(Note: an Error Processing transfer of control (c.f. 6.3) does not restore the value of **$test**.)

8.2.31 Trestart

tre[**start**] *postcond* [*sp*]

If **$tlevel** is greater than zero, **trestart** performs a *restart*. If **$tlevel** is zero, **trestart** generates an error with *ecode*=**"M44"**.

8.2.32 Trollback

tro[**llback**] *postcond* [*sp*]

If **$tlevel** is greater than zero, a *rollback* is performed, **$tlevel** and **$trestart** are set to zero, and the naked indicator becomes undefined. (See the Transaction Processing subclause for the definition of *rollback*).

If **$tlevel** is zero, **trollback** generates an error with *ecode*=**"M44"**.

8.2.33 TSTART

TS[TART] *postcond*	[*sp*] *sp* *tstartargument*

tstartargument ::=	[*restartargument*] [: *transparameters*] @ *expratom* *V* *tstartargument*

restartargument ::=	*lname* (*L* *lname*) * ( )

transparameters ::=	*tsparam* (*tsparam* [: *tsparam*] ...)

tsparam ::= *tstartkeyword* [= *expr*]

Tstartkeywords that differ only in the use of corresponding upper and lower-case letters are equivalent. The standard defines the following keywords: **@**∈{*expr* → *actualname*}

S[**ERIAL**]

T[**RANSACTIONID**]=*expr*

Z[unspecified][=*expr*]

Unused keywords other than those starting with the letter "**Z**" are reserved for future enhancement of the standard.

tstartkeywords are processed in strict left-to-right order. When multiple equivalent *tstartkeywords* are encountered, the last occurrence processed will define the action(s) to be taken.

After evaluation of *postcond*, if any, and *tstartargument*, if any, **TSTART** adds one to **$TLEVEL**. If, as a result, **$TLEVEL** is one, then **TSTART** initiates a TRANSACTION that is restartable if a *restartargument* is present, or non-restartable if *restartargument* is absent; and serializable independently of **LOCK**s if *transparameters* are present and contain the keywords **SERIAL** or **S**, or dependent on **LOCK**s for serialization if those keywords are absent.

The *tsparam*, **TRANSACTIONID**, provides a means for identifying arbitrary classes of TRANSACTIONs.

The following discussion uses terms defined in the Variable Handling (see 7.1.3.2) and Process-Stack (see 7.1.3.3) models and, like those subclauses, does not imply a required implementation technique. **TSTART** creates a RESTART CONTEXT-STRUCTURE containing the execution location of the **TSTART** *command*, values for **$TEST** and the naked indicator, a copy of the process LOCK-LIST, a RESTART NAME-TABLE and an exclusive indicator. **TSTART** attaches the CONTEXT-STRUCTURE to a linked list of such RESTART CONTEXT-STRUCTUREs for the current TRANSACTION and also to a linked list of CONTEXT-STRUCTUREs associated with the current PROCESS-STACK frame. **TSTART** copies from the currently active NAME-TABLE to the RESTART NAME-TABLE all entries corresponding to the local variable names specified by the *restartargument*. **TSTART** also points the entries in the RESTART NAME-TABLE to copies of VALUE-TABLE tuples containing values that persist unchanged from the point that the

TSTART command created the NAME-TABLE. When the *restartargument* is an asterisk (*****), it specifies all current names and causes the CONTEXT-STRUCTURE to be marked as exclusive.

8.2.34 USE

`U[SE]` *postcond sp L useargument*

```
                    | devn  [ : deviceparameters                          ] |
useargument ::=     |       [ : [ deviceparameters ] : mnemonicspace      ] |
                    | @ expratom V L useargument                            |
```

There is a large overlap in specification between the commands **OPEN**, **USE**, and **CLOSE**. As a side-effect of the alphabetical ordering of the commands, many features are described in clause 8.2.7 **CLOSE**. As a matter of style in this document, these features are not repeated in this clause. See 8.2.23 **OPEN** for *mnemonicspace*. See 8.2.7 for the syntax and interpretation of *devn* and *deviceparameters*. **@**∈{*expr* → *actualname*}

Before a device can be employed in conjunction with an input or output data transfer it must be designated, through execution of a **USE** command, as the *current device*. Before a device can be named in an executed *useargument*, its ownership must have been established through execution of an **OPEN** command.

The specified device remains current until such time as a new **USE** command is executed. As a side effect of employing *expr* to designate a current device, **$IO** is given the value of *expr* contained in *devn* and **$IOR** is given the value of *devn*.

Specification of device parameters, by means of the *expr*s in *deviceparameters*, is normally associated with the process of obtaining ownership; however, it is possible, by execution of a **USE** command, to change the parameters of a device previously obtained.

Distinct values for **$X** and **$Y** are retained for each device. The special variables **$X** and **$Y** reflect those values for the current device. When the identity of the current device is changed as a result of the execution of a **USE** command, the values of **$X** and **$Y** are saved, and the values associated with the new current device are then the values of **$X** and **$Y**.

8.2.35 View

`v[iew]` *postcond* arguments unspecified

View makes available to the implementor a mechanism for examining machine-dependent information. It is to be understood that routines containing the **view** command may not be portable.

8.2.36 WRITE

W[RITE] *postcond* *sp* L *writeargument*

```
                   | format                          |
writeargument ::=  | expr                            |
                   | * intexpr                       |
                   | @ expratom V L writeargument    |
```

The *writearguments* are executed, one at a time, in left-to-right order. Each form of argument defines an output operation to the current device. @∈{*expr* → *actualname*}

When the form of argument is *format*, processing occurs in left-to-right order.

```
format ::=  | positionformat                       |
            | / controlmnemonic [ ( L expr ) ]     |

                    | nlformat ... [ tabformat ]   |
positionformat ::=  | ffformat                     |
                    | tabformat                    |

                     | ?     |  ┌ ident ┐
controlmnemonic ::=  | ident |  └ digit ┘ ...

nlformat ::= !
ffformat ::= #
tabformat ::= ? intexpr
```

The following describes the effect of specific characters when used in a format:

! causes a *new line* operation on the current device. Its effect is the equivalent of writing *cr lf* on a pure ASCII device. In addition, **$X** is set to `0` and `1` is added to **$Y**.

causes a *top of form* operation on the current device. Its effect is the equivalent of writing *cr ff* on a pure ASCII device. In addition, **$X** and **$Y** are set to `0`. When the current device is a display, the screen is blanked and the cursor is positioned at the upper left-hand corner.

? *intexpr*
produces an effect similar to *tab to column* intexpr. If **$X** is greater than or equal to *intexpr*, there is no effect. Otherwise, the effect is the same as writing (*intexpr*-**$X**) spaces. (Note that the leftmost column of a line is column `0`.)

/ *controlmnemonic* [(*expr* [, *expr*] ...)]
produces an effect which is defined by the *mnemonicspace* which has been assumed by default or has been selected in a previous *mnemonicspace* specification with a **USE**

command. The relevant control-function is indicated by means of the *controlmnemonic* which must be defined in the above-mentioned *mnemonicspace*. Possible parameters are given through the optional *expr*s. *Controlmnemonic*s which start with the character `"?"` are implementor-specific.

The implementor may restrict the use of *controlmnemonic*s in a device-dependant way. A reference to an undefined *mnemonicspace* or an undefined *controlmnemonic* is reflected in special variable **`$DEVICE`**.

When the form of argument is *expr*, the value of *expr* is sent to the device. The effect of this string at the device is defined by appropriate device handling.

When the form of the argument is *****_intexpr_, one character, not necessarily from the ASCII set and whose code is the number represented in decimal by the value of *intexpr*, is sent to the device. The effect of this character at the device may be defined by the implementor in a device-dependent manner.

As **`WRITE`** transmits characters one at a time, certain characters or character combinations represent device control functions, depending on the identity of the current device. To the extent that the supervisory function can detect these control characters or character sequences, they will alter **`$X`** and **`$Y`** as follows.

graphic: add `1` to **`$X`**
backspace: `set $X=`*`max`*`($X-1,0)`
line feed: add `1` to **`$Y`**
carriage return: `set $X=0`
form feed: `set $Y=0,$X=0`

When a *format* specification is interpreted and the effect would cause the ‘physical’ external equivalent of **`$X`** and **`$Y`** to be modified, this effect will be reflected as far as possible in the values of the special variables **`$X`** and **`$Y`**.

Output operations, except when the form of the argument is *****_intexpr_, are affected by the Character Set Profile output-transform.

8.2.37 Xecute

`x[ecute]` *postcond* *sp* *list*{*xargument*}

xargument `::=` | *expr* *postcond* |
| **`@`**∈{*expratom* → *xargument*} |

`Xecute` provides a means of executing M code which arises from the process of expression evaluation.

Each *xargument* is evaluated one at a time in left-to-right order. If the *postcond* in the *xargument* is present and its *tvexpr* is false, the *xargument* is not executed. Otherwise, if the value of *expr* is *x*, execution of the *xargument* is executed in a manner equivalent to execution of `do` *y*, where *y* is the spelling of an otherwise unused *label* attached to the following two-line subroutine considered to be a part of the currently executing routine:

y *ls* *x* *eol*

ls `quit` *eol*

8.2.38 Z

`z[unspecified]` arguments unspecified

All *commandwords* in a given implementation which are not defined in the standard are to begin with the letter **`z`**. This convention protects the standard for future enhancement.

8.3 Device parameters

8.3.1 Output time out

For any *mnemonicspace* the implementation may define a device parameter that causes an error condition when an output-producing argument of a **`read`** or **`write`** *command* fails to complete execution within a specified time. If it is defined, the device parameter shall conform to this clause and to the related sections of 7.1.4.2.

This device parameter shall have the following form:

deviceparam `::=` **`OUTTIMEOUT=`***numexpr*

Numexpr shall be interpreted as the value of a *timeout* (see 8.1.6). Should any subsequent output-producing argument of a **`read`** or **`write`** *command* to the device fail to complete execution within that time, then

a the **`OUTSTALLED`** member of **`^$device`**, described in 7.1.4.2, shall assume the value **`1`**, and

b an error with *ecode*=`"`**`M100`**`"` shall occur.

Output time out shall not apply to a device when

a no **`OUTTIMEOUT`** *deviceparam* has executed for the device, or

b the value of *numexpr* in the most recent **`OUTTIMEOUT`** is non-positive.

An execution of an **`OUTTIMEOUT`** *deviceparam* shall replace any previous **`OUTTIMEOUT`** *deviceparam* for the device.

The **`close`** command shall

a set the value of the **`OUTTIMEOUT`** *deviceparam* to **`0`**;

b set the value of the **`OUTTIMEOUT`** member of **`^$device`** to `0`;

c set the value of the **`OUTSTALLED`** member of **`^$device`** to `0`**`.`**

Note: this is an exception to the general specification of device parameters in 8.2.7.

Note: output time out applies to the execution of **`read`** or **`write`** arguments, not to the delivery of data to a device.

9 Character set profile charset

A *charset* is a definition of the valid characters and their characteristics available to a process. The required characteristics for a fully defined *charset* are:

a The character codes and their meaning
b The definition of which character codes are valid in names
c The available *patcode*s and their definitions
d The collation order of character strings.

Note: a charset definition is not necessarily tied to any (natural) language and could be an arbitrary set of characters or a repertoire from another set, such as ISO 10646.

```
   charset ::= descriptor
             | %     |            | ident |
descriptor ::= |       | [descsep]  |       | ...
             | ident |            | digit |

             | - | (note: hyphen)
             | _ | (note: underscore)
             | % |
             | * |
             | . |
   descsep ::= | / |
             | + |
             | : |
             | $ |
             | ! |
             | @ |
```

The definition of the contents of standardized *charset*s is in Annex A. Unused *charset* names beginning with the initial letter **Y** are available for usage by M programmers; those beginning with the initial letter **Z** are reserved for implementor-defined *charset*s; all other *charset* names are reserved for future enhancement of the *Standard*.

Section 2
M portability requirements

Introduction

Section 2 highlights, for the benefit of implementors and application programmers, aspects of the language that must be accorded special attention if M program transferability (i.e., portability of source code between various M implementations) is to be achieved. It provides a specification of limits that must be observed by both implementors and programmers if portability is not to be ruled out. To this end, implementors *must meet or exceed* these limits, treating them as a minimum requirement. Any implementor who provides definitions in currently undefined areas must take into account that this action risks jeopardizing the upward compatibility of the implementation, upon subsequent revision of the M Language Specification. Application programmers striving to develop portable programs must take into account the danger of employing “unilateral extensions” to the language made available by the implementor.

The following definitions apply to the use of the terms *explicit limit* and *implicit limit* within this document. An explicit limit is one which applies directly to a referenced language construct. Implicit limits on language constructs are second-order effects resulting from explicit limits on other language constructs. For example, the explicit command line length restriction places an implicit limit on the length of any construct which must be expressed entirely within a single command line.

1 Character set

The character set used for routines and data is restricted to the Character Set Profile `M` (as defined in Annex A).

2 Expression elements

2.1 Names

Portable *name* length is limited to thirty-one (`31`) characters. All characters in a *name* are significant in determining uniqueness. Therefore the length restriction places an implicit limit on the number of unique names on an implementation. If a name’s length exceeds an implementor's limit an error condition occurs with *ecode*=`"M56"`.

2.2 External routines and names

The *externalroutinename* namespace is unspecified, as this is a function of the binding, although at the present time, a maximum of twenty-four (24) characters allowed is placed upon *externalroutinenames* to be treated uniquely, although this should be viewed as a minimum number that needs to be handled rather than as the maximum number that can be used. Any number of characters, from one to the maximum number shall be valid as *externalroutinename*s. Any additional external mapping between these *name*s and any actually used by an external package is an implementation issue.

2.3 Local variables

2.3.1 Number of local variables

The number of local variable names in existence at any time is not explicitly limited. However, there are implicit limitations due to the storage space restrictions (Clause 8).

2.3.2 Number of subscripts

There is no explicit limit on the number of distinct local variable nodes which may be defined, but there is an implicit limit based on the number of subscripts that may be defined for any local variable reference. The number of subscripts in a local variable is limited in that, in a local array reference, the total length of the array reference must not exceed 510 characters. The length of an array reference is calculated as follows:

assuming an array reference in the form

$name(i_1,i_2,\ldots,i_n)$,

N=$length($name$)

I=$length($i_1$)+$length(i_2)+…+$length($i_n$)

where each subscript (i_1 through i_n) is either a *numlit* or a *sublit*

$L=n$

then the total length of an array reference is

N+I+(2*L)+15.

2.3.3 Values of subscripts

Local variable subscript values are nonempty strings which shall only contain characters from the M printable character subset. The length of individual subscripts is limited to 255 characters; in addition, a complete variable name reference is limited according to the restrictions specified in 2.3.2. When the subscript value satisfies the definition of a numeric data value (See 7.1.5.3 of Section 1), it is further subject to the restrictions of number range given in 2.6. The use of subscript values which do not meet these criteria is undefined, except for the use of the empty string as the last subscript of a starting reference in the context of data transversal functions such as **$order** and **$query**.

2.4 Global variables

2.4.1 Number of global variables

There is no explicit limit on the number of distinct global variable names in existence at any time.

2.4.2 Number of subscripts

The number of subscripts in a global variable is limited in that, in a global array reference, the total length of the array reference must not exceed `510` characters. The length of an array reference is calculated as follows:

assuming an array reference in the form

$\texttt{^|}\mathit{environment}\texttt{|}\mathit{name}\texttt{(}i_1\texttt{,}i_2\texttt{,}\ldots\texttt{,}i_n\texttt{),}$

$E\texttt{=\$length(}\mathit{environment}\texttt{)}$

$N\texttt{=\$length(}\mathit{name}\texttt{)}$

$I\texttt{=\$length(}i_1\texttt{)+\$length(}i_2\texttt{)+}\ldots\texttt{+\$length(}i_n\texttt{)}$

where each subscript (i_1 through i_n) is either a *numlit* or a *sublit*

$L\texttt{=}n$

then the total length of an array reference is

$E\texttt{+3+}N\texttt{+}I\texttt{+(2*}L\texttt{)+15.}$

2.4.3 Values of subscripts

The restrictions imposed on the values of global variable subscripts are identical to those imposed on local variable subscripts (see 2.3.3).

2.4.4 Number of nodes

There is no explicit limit on the number of distinct global variable nodes which may be defined.

2.5 Data types

The M Language Specification defines a single data type, namely, variable length character strings. Contexts which demand a numeric, integer, or truth value interpretation are satisfied by unambiguous rules for mapping a string datum into a number, integer, or truth value.

The implementor is not limited to any particular internal representation. Any internal representation(s) may be employed as long as all necessary mode conversions are performed automatically and all external behavior agrees with the M Language Specification. For example, integers might be stored as binary integers and converted to decimal character strings whenever an operation requires a string value.

2.6 Number range

All values used in arithmetic operations or in any context requiring a numeric interpretation are within the inclusive intervals $[-10^{25},-10^{-25}]$ or $[10^{-25},10^{25}]$, or are zero.

If the result of any mathematical operation is too large (either positive or negative) for the implementation to represent it to the accuracy specified earlier in this clause, an error will occur with *ecode*=**"M92"**.

Implementations shall represent numeric quantities with at least 15 significant digits. The error introduced by any single instance of the arithmetic operations of addition, subtraction, multiplication, division, integer division, or modulo shall not exceed one part in 10^{15}. The error introduced by exponentiation shall not exceed one part in 10^{7}.

Programmers should exercise caution in the use of noninteger arithmetic. In general, arithmetic operations on noninteger operands or arithmetic operations which produce noninteger results cannot be expected to be exact. In particular, noninteger arithmetic can yield unexpected results when used in loop control or arithmetic tests.

2.7 Integers

The magnitude of the value resulting from an integer interpretation is limited by the accuracy of numeric values (see 2.6). The values produced by integer valued operators and functions also fall within this range (see 7.1.5.6 of Section 1 for a precise definition of integer interpretation).

2.8 Character strings

Character string length is limited to 32,767 characters for local variables, 510 characters for global variables, 32,767 characters for structured system variables. If a string's length exceeds an implementor's limit, an error condition occurs with *ecode*=**"M75"**.

The characters permitted within character strings must include those defined in the *ASCII Standard* (*ANSI X3.4-1986*).

2.9 Special variables

The special variables **$x** and **$y** are nonnegative integers (see 2.7). The effect of incrementing **$x** and **$y** past the maximum allowable value is undefined. (For a description of the cases in which the values of **$x** and **$y** may be altered see 8.2.36 of Section 1; for a description of the type of values **$x** and **$y** may have see 7.1.5.10 of Section 1). The value of **$system** as provided by an implementor must conform to the requirements for a local variable subscript (see 2.3.3).

3 Expressions

3.1 Nesting of expressions

The number of levels of nesting in expressions is not explicitly limited. The maximum string length does impose an implicit limit on this number (see 2.8).

3.2 Results

Any final result which does not satisfy the constraints on character strings (see 2.8) is erroneous. Any intermediate result which does not satisfy the constraints on local variable character strings (see 2.8) is erroneous. Furthermore, integer results are erroneous if they do not satisfy the constraints on integers (see 2.7).

3.3 External references

External references are not portable.

4 Routines and command lines

4.1 Command lines

A command line (*line*) must satisfy the constraints on global variable character strings (see 2.8). The length of a command line is the number of characters in the *line* up to but not including the *eol*.

The characters within a command line are restricted to the 95 ASCII printable characters. The character set restriction places a corresponding implicit restriction upon the value of the argument of the indirection delimiter (Clause 7).

4.2 Number of command lines

There is no explicit limit on the number of command lines in a routine, subject to storage space restrictions (Clause 8).

4.3 Number of commands

The number of commands per line is limited only by the restriction on the maximum command line length (see 4.1).

4.4 Labels

A label of the form *name* is subject to the constraints on names (see 2.1), with the exception that the first `31` characters are uniquely distinguished. Labels of the form *intlit* are subject to the same length constraints.

4.5 Number of labels

There is no explicit limit on the number of labels in a routine. However, the following restrictions apply:

a A command line may have only one label.

b No two lines may be labeled with equivalent (not uniquely distinguishable) labels.

4.6 Number of routines

There is no explicit limit on the number of routines. The number of routines is implicitly limited by the name length restriction (see 2.1).

5 External routine calls

When the external routine called is not within the current default M environment, all variables should be assumed to be scalars (i.e., *a* refers to the value associated with *a*, but does not refer to any descendants *a* might have such as *a*`(1)`, etc.). No prohibition against non-scalar extensions should be inferred, only that they may not be portable. It should be noted that no all-encompassing implied guarantee of the number of routines supported by an external package exists.

6 Character set profiles

Character Set Profiles are registered through the MUMPS Development Committee (ANSI X11). New Character Set Profile Definitions are approved through the standard procedures of the MUMPS Development Committee.

Routines and data created using a registered Character Set Profile are portable to all implementations which support that Character Set Profile.

The list of MDC registered Character Set Profiles is included in Annex A.

Note that subscript-string length (see 2.3.2, 2.3.3, 2.4.2, 2.4.3) is either the length of the value of the subscript, or the length of the computed Character Set Profile collation value, whichever is larger.

Collation values are not portable between implementations unless the value is explicitly stated in the definition of the Character Set Profile.

7 Indirection

The values of the argument of indirection and the argument of the `xecute` command are subject to the constraints on character string length (see 2.8). They are additionally restricted to the character set limitations of command lines (see 4.1).

8 Storage space restrictions

The size of a single routine must not exceed `20,000` characters. The size of a routine is the sum of the sizes of all the lines in the routine. The size of each line is its length (as defined in 4.1) plus two.

Note: in comparison to previous versions of the standard, there is no specification of local variable storage. Like global variable storage, local variable storage can be arbitrarily large. The implementation's conformance statement must specify the minimum guaranteed to be available.

9 Process-stack

Systems will provide a minimum of `127` levels in the PROCESS-STACK. The actual use of all these levels may be limited by storage restrictions (Clause 8).

Nesting within an expression is not counted in this limit. Expression nesting is not explicitly limited; however, it is implicitly limited by the storage restriction (Clause 8).

10 Formats

Device control may be effected through the `read` and `write` commands using the `/`*controlmnemonic* syntax in a specification of a *format*. In general, portability of routines containing such syntax is only possible in cases which meet several criteria, most obviously

a the devices to be used at the receiving facility must have all the capabilities required by the `/`*controlmnemonic* occurrences in the routines;

b the implementors of the systems at both the originating and the receiving facilities have implemented each combination of *mnemonicspace* and *controlmnemonic* in compatible ways.

As a result of these limitations, “blind interchange” will only be dependent upon the *device*s at the receiving site.

However, the following advice to both implementors and programmers will increase the number of cases in which “informed interchange” will be possible.

Note: user-defined *mnemonicspace*s, together with their associated *controlmnemonic*s, are inherently portable provided that the M routines are also portable.

10.1 *Mnemonicspace*

For portability, the *mnemonicspace* to be used must be a generally accepted standard, e.g. *ANSI X3.64-199_* or *GKS,* or after such a standard would have been accepted, any other ANSI or ISO standard.

10.2 *Controlmnemonic*

For portability, the *controlmnemonic* must be one of the *controlmnemonic*s assigned to a control-function specified in the chosen *mnemonicspace* and interpretation of the *format* specification must lead to the effect described in the *mnemonicspace*. There should be no other (side-)effects on the device.

With regard to the status of the process, the value of some special variables may change, e.g. with some control-functions **`$x`** and **`$y`** would have to receive proper values. Apart from these documented effects, no other effects may be caused by any implementation.

An implementation needs not to allow for all *controlmnemonic*s in all *mnemonicspace*s.

10.3 Parameters

A *format* containing **`/`***controlmnemonic* may contain one or more parameters, specified as *list*{*expr*}, in which case each *expr* specifies a parameter of the control-function. The *expr*s must appear in the same order and number as the parameters in the corresponding *mnemonicspace*. The value of each *expr* should meet the limitations of 2.6 through 2.8.

11 Transaction processing

11.1 Number of modifications in a *transaction*

The sum of the lengths of the *namevalue*s and values of global variable tuples modified within a *transaction* must not exceed `57,343` characters.

11.2 Number of nested `tstarts` within a *transaction*

A single *transaction* must not contain more than `126` **`tstart`**s after the **`tstart`** that initiates the *transaction*.

12 Event processing

12.1 Number of timers

The number of concurrently running timers must not exceed one (`1`) per process or sixteen (`16`) per system, whichever is smaller.

12.2 Depth of event queues

The per-process event queues (one each for synchronous and asynchronous events) must not contain more than one event.

12.3 Resolution of timers

Timers must not use a resolution finer than one second.

13 Other portability requirements

Programmers should exercise caution in the use of noninteger values for the **`hang`** command and in **`TIMER`** events and timeouts. In general, the period of actual time which elapses upon the execution of a **`hang`** command or which elapses before a **`TIMER`** event cannot be expected to be exact. In particular, relying upon noninteger values in these situations can lead to unexpected results.

Implementations may restrict access to *ssvn*s that contain default *environment*s of processes other than the one referring to the *ssvn*. Therefore, portable programs shall not rely on the *ssvn*s defined in 7.1.4.5.7 when *processid* is not their own **`$job`**.

Section 3
X3.64 binding

Introduction

ANSI X3.64 is a functional standard for additional control functions for data interchange with two-dimensional character-imaging input and/or output devices. It is an ANSI standard, but also an ISO standard with roughly similar characteristics exists (*ISO 2022*). As such, it has been implemented in many devices worldwide. It is expected that M can be easily adapted to these implementations.

The standard defined as *ANSI X3.64* defines a format for device-control. No physical device is required to be able to perform all possible control-functions. In reality, as some functions rely on certain physical properties of specific devices, no device will be able to perform all functions. The standard, however, does not specify which functions a device should be able to do, but if it is able to perform a function, how the control-information for this function is to be specified.

This binding is to the functional definitions included in *X3.64.* The actual dialogue between the M implementation and the device is left to the implementor.

1 The binding

ANSI X3.64 is accessed from the M language by making use of *mnemonicspace*s. A *controlmnemonic* from *X3.64* may be accessed as follows:

/*controlmnemonic* [(*expr* [, *expr*] ...)]

where the relevant *controlmnemonic* equalling the generic function and *expr*s the possible applicable parameters. The use of a *controlmnemonic* produces the effect defined in *ANSI X3.64* for the control-function with the same name as the *controlmnemonic* specified.

Some *controlmnemonic*s return a value, or a collection of values. It is perfectly legal to issue these *controlmnemonic*s with either a **READ** or **WRITE** *command*. If a **READ** *command* is used, the argument list in the statement(s) must be ordered to correctly accept the returned values. If a **WRITE** *command* is used the values returned may be read by a single, or series of, **READ** *command*s. These **READ** *command*s must be correctly ordered to match the returned values, however there may be intermediate calculations utilizing some of the returned values before reading the remaining values in the list. Reading the return list of values may be terminated without error by issuing another *controlmnemonic*. In this case, all returned values not assigned to a variable will be lost to the application program.

All *controlmnemonic*s have the same name in M as in *X3.64.*

Unless explicitly mentioned, the use of *X3.64 controlmnemonic*s has no side-effects on special variables such as **$X**, **$Y**, **$KEY** and **$DEVICE**.

1.1 Control-functions with an effect on `$X` or `$Y` or both

Below follows a list of control-functions (*X3.64*) or *controlmnemonic*s (M) that have an effect on the special variables **`$X`** or **`$Y`** or both. Since some definitions in *X3.64* are fairly open-ended, the exact effect may be implementation dependent in some cases. In section 3.4 these open-ended definitions are listed resolution of possible ambiguities are stated.

The relevant *controlmnemonic*s are:

`/CBT(n)`	`$X`	`/CUP(y,x)`	`$X, $Y`	`/NEL`	`$X, $Y`
`/CHA(x)`	`$X`	`/CUU(n)`	`$Y`	`/PLD`	`$Y`
`/CHT(n)`	`$X`	`/CVT(n)`	`$Y`	`/PLU`	`$Y`
`/CNL(n)`	`$X, $Y`	`/HPA(x)`	`$X`	`/REP(n)`	`$X, $Y`
`/CPL(n)`	`$X, $Y`	`/HPR(n)`	`$X`	`/RI`	`$Y`
`/CUB(n)`	`$X`	`/HTJ`	`$X`	`/RIS`	`$X=0, $Y=0`
`/CUD(n)`	`$Y`	`/HVP(y,x)`	`$X, $Y`	`/VPA(y)`	`$Y`
`/CUF(n)`	`$X`	`/IND`	`$Y`	`/VPR(n)`	`$Y`

The control-function **`REP`** repeats the previous character or function as many times as indicated by its argument. Hence, the side-effects of this function do not depend on this function itself, but rather on the character or function that is being repeated.

1.2 Control-functions with an effect on `$KEY`

Currently only one *controlmnemonic* may have a side-effect on special variable **`$KEY`**: **`/DSR`** (device status report). The side-effect depends on the value of the parameter of this function: parameter-value `0` or `5` will cause a status report to be returned, parameter-value **`6`** will cause the active cursor-position to be returned. The format of the value returned is:

`$CHAR(27,91)_`*`REPORT`*`_$CHAR(110)`

or

`$CHAR(27,91)_`*`Y`*`_$CHAR(59)_`*`X`*`_$CHAR(82)`

where *`REPORT`* is a code for the status reported, *`Y`* is the value of the current Y-coordinate and *`X`* is the value of the current X-coordinate.

The values described will be reported in special variable **`$KEY`** as a side-effect of the first **`READ`** *command* that is executed after the control-function has been issued.

1.3 Control-functions with an effect on `$DEVICE`

All *controlmnemonic*s will have a side-effect on special variable **`$DEVICE`**. The most common situation will be that **`$DEVICE`** will receive the value:

`"0,,X3-64"`

in order to reflect the correct processing of a *controlmnemonic*.

In certain situations a status has to be indicated. Status codes for **$DEVICE** relating to *X3.64* are as follows:

Code	American English Description
1	mnemonicspace not found
2	invalid mnemonic
3	parameter out of range
4	hardware error
5	mnemonic not available for this device
6	parameter not available for this device
7	attempt to move outside boundary—not moved
8	attempt to move outside boundary—moved to boundary
9	auxiliary device not ready

1.4 Open-ended definitions

Under some conditions, the behavior specified by *X3.64* is either ambiguous or optional. The following clarifies the behavior to ensure consistency:

CBT Move the cursor to the last horizontal tabulator-stop in the previous line. If no such tabulator-stop exists, do not move the cursor.

CHA When a location outside the available horizontal range is specified, move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value greater than current position) or leftmost (parameter value less than current position) position.

CHT When no further forward horizontal tabulator-stops have been defined in the current line, move the cursor to the first horizontal tabulator-stop in the next line. If no such tabulator-stop exists, do not move the cursor.

CNL When the cursor is moved forward beyond the last line on the device, do not move the cursor. If the output device is a CRT-screen, scroll up one line.

CPL When the cursor is moved backward beyond the first line on the device, do not move the cursor. If the output device is a CRT-screen, scroll down one line.

CUB When the cursor is moved backward beyond the first position on a line, do not move the cursor.

CUD When the cursor is moved downward beyond the last line on a device, do not move the cursor.

CUF When the cursor is moved forward beyond the last position on a line, do not move the cursor.

CUP When a location outside the available horizontal or vertical ranges is specified, do not move the cursor.

CUU When the cursor is moved upward beyond the last line on a device, do not move the cursor.

CVT When no further forward vertical tabulator-stops have been defined on the device, move the cursor to the first vertical tabulator-stop in the next page. If no such tabulator-stop exists, do not move the cursor.

`HPA` When a location outside the available horizontal range is specified, move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value greater than current position) or leftmost (parameter value less than current position) position.

`HPR` When a location outside the available horizontal range is specified, move the cursor in the direction suggested by the parameter-value to either the rightmost (parameter value positive) or leftmost (parameter value negative) position.

`HTJ` When no further forward horizontal tabulator-stops have been defined in the current line, move the cursor to the first horizontal tabulator-stop in the next line. If no such tabulator-stop exists, do not move the cursor.

`HVP` When a location outside the available horizontal or vertical ranges is specified, do not move the cursor.

`IND` When the cursor is moved downward beyond the last line on a device, move the cursor to the corresponding horizontal position in the first line on the next page.

`NEL` When the cursor is moved downward beyond the last line on a device, move the cursor to the first position on the first line on the next page.

`PLD` This function may or may not be similar to `CUD` or `IND`. The effect of two successive `PLD` operations may or may not be equal to the effect of one single `CUD` or `IND` operation; this function will be identical to `CUD`. The effect of `PLD` and `PLU` will be complementary, i.e. .`PLD` immediately followed by `PLU` will effectively not move the cursor.

`PLU` This function may or may not be similar to `CUU` or `RI`. The effect of two successive `PLU` operations may or may not be equal to the effect of one single `CUU` or `RI` operation; this function will be identical to `CUU`. The effect of `PLD` and `PLU` will be complementary, i.e. .`PLU` immediately followed by `PLD` will effectively not move the cursor.

`RI` When the cursor is moved upward beyond the first line on a device, move the cursor to the corresponding horizontal position in the last line on the previous page.

`VPA` When a location outside the vertical range is specified, move the cursor in the direction suggested by the parameter-value to either the bottommost (parameter value greater than current position) or topmost (parameter value less than current position) position.

`VPR` When a location outside the vertical range is specified, move the cursor in the direction suggested by the parameter-value to either the bottommost (parameter value positive) or topmost (parameter value negative) position.

The following functions shall not cause the cursor to move: `ICH`, `JFY`, `MC`, `NP`, `DL` and `PP`.

The following functions shall move the cursor so that it will point to the same character in the new projection of the information: `SD`, `SL`, `SR` and `SU`. Boundary conditions will be similar to `CUD`, `CUB`, `CUF`, and `CUU` respectively.

2 Portability issues

2.1 Implementation

Any implementation of this binding will accept all *controlmnemonic*s specified. However, in most cases all *controlmnemonic*s will not be supported for all devices. The appropriate error code will be return in `$DEVICE` to indicate if a particular *controlmnemonic* is supported for the current device.

2.2 Application

Several *controlmnemonic*s specified in *X3.64* are ambiguous and usage of these will likely have different meaning between different devices and implementations. Usage of these will not be portable.

Control-mnemonic	Control Function
`APC`	Application Program Command
`DA`	Device Attributes
`DCS`	Device Control String
`FNT`	Font Selection
`INT`	Interrupt
`OSC`	Operating System Command
`PLD`	Partial Line Down (`CUD` recommended; see 1.4)
`PLU`	Partial Line Up (`CUU` recommended; see 1.4)
`PM`	Privacy Message
`PU1`	Private Use One
`PU2`	Private Use Two

Control-mnemonic	Control Function
`SGR`	Select Graphic Rendition for the following:
`10`	primary font
`11`	first alternative font
`12`	second alternative font
`13`	third alternative font
`14`	forth alternative font
`15`	fifth alternative font
`16`	sixth alternative font
`17`	seventh alternative font
`18`	eighth alternative font
`19`	ninth alternative font
`SS2`	Single Shift Two
`SS3`	Single Shift Three

3 Conformance

Each implementation must supply a list of the *controlmnemonic*s and arguments that are supported for each device.

Annex A · Character set profiles
(normative)

The definition of a Character Set Profile requires the definition of four elements—the names of the characters in the character set and the internal codes which are used to represent them, the definitions of which characters match which pattern codes, the collation scheme used, and the definition of which characters may be used in *name*s.

Note that the *patcode*s **`A`**, **`C`**, **`E`**, **`L`**, **`N`**, **`P`**, and **`U`** are applicable for all character set profiles; in addition *patcode* **`E`** matches any character, not just those listed in any specific *charset*.

Two collation schemes are provided which only require a properly defined table of characters for the Character Set associated with the specific Character Set Profile.

`STRING` COLLATION

Determining the Collation Ordering for a Character Set Profile requires the collation value(s) for each character within the character set be accessible as a group of values presented as an *n*-tuple. Each column of the definition table provides one value of the tuple in the specified order. When no value is present in any column, the corresponding character id value is used in its place. Note that certain characters may be represented with more than one value entry line in the table; in these cases the entries are taken one at a time and treated as if they represented separate characters in the original string (e.g. the character `M` in `ISO-Latin-1` (id# `198`) would be treated as a form of the string `"`**`AE`**`"`).

Let s be any non-empty string. Define the numeric function $CV_n(s)$ to return the nth -order collation value for string s: unless otherwise specified this value is determined by evaluating the value in the nth-column of each collation tuple for each character in the string examined in left-to-right order and combining those tuples together. Note: selected collation-tuple columns may optionally be designated for right-to-left evaluation.

The Collation Ordering function *`CO`* determines relative ordering for a character set. The exact value of this function is not specified here, however, the values formed by any implementation must satisfy the following rules when comparing two non-equal strings:

Let t also be any non-empty string, not equal to s. The *`STRING`* Collation Ordering function *`CO`* is defined as:

a `CO("",s)=s`.

b $CO(s,t)=t$ if, and only if, $CV_j(n)>CV_j(m)$ and for all i, i=1…j-1, $CV_i(t)=CV_i(s)$; otherwise, $CO(s,t)=s$.

`M` COLLATION

The *`M`* Collation Ordering function *`CO`* for uses the definition of $CV_n(s)$ specified in *`STRING`* Collation and is otherwise different only with respect to numbers:

Let s be any non-empty string, let m and n be strings satisfying the definition of numeric data values (see I.7.1.5.3), and *u* and *v* be non-empty strings which do not satisfy that definition.

a *CO("",s)=s*

b *CO(m,n)=n* if *n>m*; otherwise, *CO(m,n)=m*

c *CO(m,u)=u*

d *CO(u,v)=v* if, and only if, $CV_j(v)>CV_j(u)$ and for all `i`, `i`=1...`j`-1, $CV_i(v)=CV_i(u)$; otherwise, *CO(u,v)=u*.

1 Charset `M`

The *charset* `M` is defined using the table A.1. The values in the columns headed *Character ID* and *Character Symbol* are taken from ASCII (*X3.4-1990*). The column headed patcode defines which characters match the *patcodes* **A**, **C**, **E**, **L**, **N**, **P**, and **U**. The characters in the table with a *patcode* of **A** are defined as *ident*s. The collation rule used is *M* collation, using the collation order values presented in the table.

2 Charset `ASCII`

The *charset* `ASCII` is defined using the table A.1. The values in the columns headed *Character ID* and *Character Symbol* are taken from ASCII (*X3.4-1990*). The column headed patcode defines which characters match the *patcodes* **A**, **C**, **E**, **L**, **N**, **P**, and **U**. The characters in the table with a *patcode* of **A** are defined as *ident*s. The collation rule used is *STRING* collation, using the collation order values presented in the table.

Table A.1—`ASCII` character set table

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
0	*NUL*	C,E	0		
1	*SOH*	C,E	1		
2	*STX*	C,E	2		
3	*ETX*	C,E	3		
4	*EOT*	C,E	4		
5	*ENQ*	C,E	5		
6	*ACK*	C,E	6		
7	*BELL*	C,E	7		
8	*BS*	C,E	8		
9	*HT*	C,E	9		
10	*LF*	C,E	10		
11	*VT*	C,E	11		
12	*FF*	C,E	12		
13	*CR*	C,E	13		
14	*SO*	C,E	14		
15	*SI*	C,E	15		
16	*DLE*	C,E	16		
17	*DC1*	C,E	17		

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
18	*DC2*	C,E	18		
19	*DC3*	C,E	19		
20	*DC4*	C,E	20		
21	*NAK*	C,E	21		
22	*SYN*	C,E	22		
23	*ETB*	C,E	23		
24	*CAN*	C,E	24		
25	*EM*	C,E	25		
26	*SUB*	C,E	26		
27	*ESC*	C,E	27		
28	*FS*	C,E	28		
29	*GS*	C,E	29		
30	*RS*	C,E	30		
31	*US*	C,E	31		
32	*SP* (space)	P,E	32		
33	!	P,E	33		
34	"	P,E	34		
35	#	P,E	35		
36	$	P,E	36		
37	%	P,E	37		
38	&	P,E	38		
39	' (apostrophe)	P,E	39		
40	(	P,E	40		
41	)	P,E	41		
42	*	P,E	42		
43	+	P,E	43		
44	, (comma)	P,E	44		
45	- (hyphen)	P,E	45		
46	.	P,E	46		
47	/	P,E	47		
48	0	N,E	48		
49	1	N,E	49		
50	2	N,E	50		
51	3	N,E	51		
52	4	N,E	52		
53	5	N,E	53		
54	6	N,E	54		
55	7	N,E	55		
56	8	N,E	56		
57	9	N,E	57		
58	:	P,E	58		
59	;	P,E	59		
60	<	P,E	60		
61	=	P,E	61		
62	>	P,E	62		
63	?	P,E	63		
64	@	P,E	64		
65	A	A,U,E	65		
66	B	A,U,E	66		
67	C	A,U,E	67		
68	D	A,U,E	68		
69	E	A,U,E	69		
70	F	A,U,E	70		
71	G	A,U,E	71		

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
72	H	A,U,E	72		
73	I	A,U,E	73		
74	J	A,U,E	74		
75	K	A,U,E	75		
76	L	A,U,E	76		
77	M	A,U,E	77		
78	N	A,U,E	78		
79	O	A,U,E	79		
80	P	A,U,E	80		
81	Q	A,U,E	81		
82	R	A,U,E	82		
83	S	A,U,E	83		
84	T	A,U,E	84		
85	U	A,U,E	85		
86	V	A,U,E	86		
87	W	A,U,E	87		
88	X	A,U,E	88		
89	Y	A,U,E	89		
90	Z	A,U,E	90		
91	[	P,E	91		
92	\	P,E	92		
93	]	P,E	93		
94	^	P,E	94		
95	_ (underscore)	P,E	95		
96	`	P,E	96		
97	a	A,L,E	97		
98	b	A,L,E	98		
99	c	A,L,E	99		
100	d	A,L,E	100		
101	e	A,L,E	101		
102	f	A,L,E	102		
103	g	A,L,E	103		
104	h	A,L,E	104		
105	i	A,L,E	105		
106	j	A,L,E	106		
107	k	A,L,E	107		
108	l	A,L,E	108		
109	m	A,L,E	109		
110	n	A,L,E	110		
111	o	A,L,E	111		
112	p	A,L,E	112		
113	q	A,L,E	113		
114	r	A,L,E	114		
115	s	A,L,E	115		
116	t	A,L,E	116		
117	u	A,L,E	117		
118	v	A,L,E	118		
119	w	A,L,E	119		
120	x	A,L,E	120		
121	y	A,L,E	121		
122	z	A,L,E	122		
123	{	P,E	123		
124	|	P,E	124		
125	}	P,E	125		
126	~	P,E	126		

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
127	*DEL*	C,E	127		

Note: 2nd and 3rd order collation values happen to be blank (i.e. not needed) for this Character Set Profile definition; the 1st order collation value happens to be unique across all the characters in this profile.

3 Charset JIS90

The *charset* JIS90 supports an encoding of Japanese characters. The specification for this was developed by the MUMPS Development Coordinating Committee—Japan and is described in *JIS X0201-1990* and *JIS X0208-1990.* The English translation is partially reproduced in Annex G for information purposes. The reader should refer to *JIS X0201-1990* and *JIS X0208-1990* for full definition.

(Note that Annex G is informational.)

4 Charset ISO-8859-1-USA

The charset ISO-8859-1-USA is defined using the table A.2. The values in the columns headed *Character ID* and *Character Symbol* are taken from ISO-8859-1 (ISO Latin 1). The column headed *patcode* defines which characters match the *patcode*s **A**, **C**, **E**, **I**, **L**, **N**, **P**, and **U**. The characters in the table with a patcode of **A** are defined as *ident*s. The collation rule used is *STRING* collation, using the collation order values provided in the table: note that all collation is left-to-right precedence. Note also that the *patcode* **I** matches any non-ASCII characters (id# greater than 127), not just those listed in this *charset*.

5 Charset ISO-8859-1-USA/M

The charset ISO-8859-1-USA/M is defined using the table A.2. The values in the columns headed *Character ID* and *Character Symbol* are taken from ISO-8859-1 (ISO Latin 1). The column headed *patcode* defines which characters match the *patcode*s **A**, **C**, **E**, **I**, **L**, **N**, **P**, and **U**. The characters in the table with a *patcode* of **A** are defined as *ident*s. The collation rule used is *M* collation, using the collation order values provided in the table: note that all collation is left-to-right precedence. Note also that the *patcode* **I** matches any non-ASCII characters (id# greater than 127), not just those listed in this *charset*.

Table A.2—ISO-8859-1-USA character set table

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
0	*NUL*	C,E	0		
1	*SOH*	C,E	1		
2	*STX*	C,E	2		
3	*ETX*	C,E	3		
4	*EOT*	C,E	4		
5	*ENQ*	C,E	5		
6	*ACK*	C,E	6		
7	*BELL*	C,E	7		
8	*BS*	C,E	8		
9	*HT*	C,E	9		
10	*LF*	C,E	10		
11	*VT*	C,E	11		
12	*FF*	C,E	12		
13	*CR*	C,E	13		
14	*SO*	C,E	14		
15	*SI*	C,E	15		
16	*DLE*	C,E	16		
17	*DC1*	C,E	17		
18	*DC2*	C,E	18		
19	*DC3*	C,E	19		
20	*DC4*	C,E	20		
21	*NAK*	C,E	21		
22	*SYN*	C,E	22		
23	*ETB*	C,E	23		
24	*CAN*	C,E	24		
25	*EM*	C,E	25		
26	*SUB*	C,E	26		
27	*ESC*	C,E	27		
28	*FS*	C,E	28		
29	*GS*	C,E	29		
30	*RS*	C,E	30		
31	*US*	C,E	31		
32	*SP* (space)	P,E	32		
33	!	P,E	33		
34	"	P,E	34		
35	#	P,E	35		
36	$	P,E	36		
37	%	P,E	37		
38	&	P,E	38		
39	' (apostrophe)	P,E	39		
40	(	P,E	40		
41	)	P,E	41		
42	*	P,E	42		
43	+	P,E	43		
44	, (comma)	P,E	44		
45	- (hyphen)	P,E	45		
46	.	P,E	46		
47	/	P,E	47		
48	0	N,E	48		
49	1	N,E	49		
50	2	N,E	50		
51	3	N,E	51		
52	4	N,E	52		

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
53	5	N,E	53		
54	6	N,E	54		
55	7	N,E	55		
56	8	N,E	56		
57	9	N,E	57		
58	:	P,E	58		
59	;	P,E	59		
60	<	P,E	60		
61	=	P,E	61		
62	>	P,E	62		
63	?	P,E	63		
64	@	P,E	64		
65	A	A,U,E	65		
66	B	A,U,E	66	1	1
67	C	A,U,E	67	1	1
68	D	A,U,E	68	1	1
69	E	A.U.E	70	1	1
70	F	A,U,E	71	1	1
71	G	A,U,E	72	1	1
72	H	A,U,E	73	1	1
73	I	A,U,E	74	1	1
74	J	A,U,E	75	1	1
75	K	A,U,E	76	1	1
76	L	A,U,E	77	1	1
77	M	A,U,E	78	1	1
78	N	A,U,E	79	1	1
79	O	A,U,E	80	1	1
80	P	A,U,E	81	1	1
81	Q	A,U,E	82	1	1
82	R	A,U,E	83	1	1
83	S	A,U,E	84	1	1
84	T	A,U,E	85	1	1
85	U	A,U,E	86	1	1
86	V	A,U,E	87	1	1
87	W	A,U,E	88	1	1
88	X	A,U,E	89	1	1
89	Y	A,U,E	90	1	
90	Z	A,U,E	91	1	1
91	[	P,E	93		
92	\	P,E	94		
93	]	P,E	95		
94	^	P,E	96		
95	_ (underscore)	P,E	97		
96	’	P,E	98		
97	a	A,L,E	65	0	1
98	b	A,L,E	66	0	1
99	c	A,L,E	67	0	1
100	d	A,L,E	68	0	1
101	e	A,L,E	70	0	1
102	f	A,L,E	71	0	1
103	g	A,L,E	72	0	1
104	h	A,L,E	73	0	1
105	i	A,L,E	74	0	1
106	j	A,L,E	75	0	1
107	k	A,L,E	76	0	1

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
108	l	A,L,E	77	0	1
109	m	A,L,E	78	0	1
110	n	A,L,E	79	0	1
111	0	A,L,E	80	0	1
112	p	A,L,E	81	0	1
113	q	A,L,E	82	0	1
114	r	A,L,E	83	0	1
115	s	A,L,E	84	0	1
116	t	A,L,E	85	0	1
117	u	A,L,E	86	0	1
118	v	A,L,E	87	0	1
119	w	A,L,E	88	0	1
120	x	A,L,E	89	0	1
121	y	A,L,E	90	0	1
122	z	A,L,E	91	0	1
123	{	P,E	99		
124	|	P,E	100		
125	}	P.E	101		
126	~	P,E	102		
127	***DEL***	C,E	103		
128		C,E,I	104		
129		C,E,I	105		
130		C,E,I	106		
131		C,E,I	107		
132	***IND***	C.E.I	108		
133	***NEL***	C,E,I	109		
134	***SSA***	C,E,I	110		
135	***ESA***	C,E,I	111		
136	***HIS***	C,E,I	112		
137	***HTJ***	C,E,I	113		
138	***VTS***	C,EJ	114		
139	***PLD***	C,E,I	115		
140	***PLU***	C,E,I	116		
141	***RI***	C,E,I	117		
142	***SS2***	C,E,I	118		
143	***SS3***	C,E,I	119		
144	***DCS***	C,E,I	120		
145	***PU1***	C,E,I	121		
146	***PU2***	C,E,I	122		
147	***STS***	C,E,I	123		
148	***CCH***	C,E,I	124		
149	***MW***	C,E,I	125		
150	***SPA***	C,E,I	126		
151	***EPA***	C,E,I	127		
152		C,E,I	128		
153		C,E,I	129		
154		C,E,I	130		
155	***CSI***	C,E,I	131		
156	***ST***	C,E,I	132		
157	***OSC***	C,E,I	133		
158	***PM***	C,E,I	134		
159	***APC***	C,E,I	135		
160	***NBSP***	C,E,I	136		
161	¡	P,E,I	137		
162	¢	P,E,I	138		

Character			Collation Table		
ID	**Symbol**	**patcode**	**1st Order**	**2nd Order**	**3rd Order**
163	£	P,E,I	139		
164	¤	P,E,I	140		
165	¥	P,E,I	141		
166	¦	P,E,I	142		
167	§	P,E,I	143		
168	¨	P,E,I	144		
169	©	P,E,I	145		
170	ª	P,E,I	146		
171	«	P,E,I	147		
172	¬	P,E,I	148		
173	*SHY*	P,E,I	149		
174	®	P,E,I	150		
175	¯	P,E,I	151		
176	°	P,E,I	152		
177	±	P,E,I	153		
178	²	P,E,I	154		
179	³	P,E,I	155		
180	´	P,E,I	156		
181	µ	P,E,I	157		
182	¶	P,E,I	158		
183	·	P,E,I	159		
184	¸	P,E,I	160		
185	¹	P,E,I	161		
186	º	P,E,I	162		
187	»	P,E,I	163		
188	¼	P,E,I	164		
189	½	P,E,I	165		
190	¾	P,E,I	166		
191	¿	P,E,I	167		
192	À	A,U,E,I	65	1	3
193	Á	A,U,E,I	65	1	2
194	Â	A,U,E,I	65	1	4
195	Ã	A,U,E,I	65	1	6
196	Ä	A,U,E,I	65	1	5
197	Å	A,U,E,I	65	1	10
			65	1	1
198	Æ	A.U.E.I	70	1	0
199	Ç	A,U,E,I	67	1	13
200	È	A,U,E,I	70	1	3
201	É	A,U,E,I	70	1	2
202	Ê	A,U,E,I	70	1	4
203	Ë	A,U,E,I	70	1	5
204	Ì	A,U,E,I	74	1	3
205	Í	A,U,E,I	74	1	2
206	Î	A,U,E,I	74	1	4
207	Ï	A,U,E,I	74	1	5
208	Ð	A,U,E,I	69	1	1
209	Ñ	A,U,E,I	79	1	6
210	Ò	A,U,E,I	80	1	3
211	Ó	A,U,E,I	80	1	2
212	Ô	A,U,E,I	80	1	4
213	Õ	A,U,E,I	80	1	6
214	Ö	A,U,E,I	80	1	5
215	×	P,E,I	168		
216	Ø	A,U,E,I	80	1	16

Character			Collation Table		
ID	Symbol	patcode	1st Order	2nd Order	3rd Order
217	Ù	A,U,E,I	86	1	3
218	Ú	A,U,E,I	86	1	2
219	Û	A,U,E,I	86	1	4
220	Ü	A,U,E,I	86	1	5
221	Ý	A.U.E.I	90	1	2
222	Þ	A.U.E.I	92	1	1
			84	0	1
223	ß	A,L,E,I	84	0	0
224	à	A,L,E,I	65	0	3
225	á	A,L,E,I	65	0	2
226	â	A,L,E,I	65	0	4
227	ã	A,L,E,I	65	0	6
228	ä	A.L,E,I	65	0	5
229	å	A,L,E,I	65	0	10
			65	0	1
230	æ	A,L,E,I	70	0	0
231	ç	A,L,E,I	67	0	13
232	è	A,L,E,I	70	0	3
233	é	A,L,E,I	70	0	2
234	ê	A.L.E.I	70	0	4
235	ë	A,L,E,I	70	0	5
236	ì	A,L,E,I	74	0	3
237	í	A,L,E,I	74	0	2
238	î	A,L,E,I	74	0	4
239	ï	A,L,E,I	74	0	5
240	ð	A,L,E,I	69	0	1
241	ñ	A,L,E,I	79	0	6
242	ò	A,L,E,I	80	0	3
243	ó	A,L,E,I	80	0	2
244	ô	A,L,E,I	80	0	4
245	õ	A,L,E,I	80	0	6
246	ö	A,L,E,I	80	0	5
247	÷	P,E,I	169		
248	ø	A,L,E,I	80	0	16
249	ù	A,L,E,I	86	0	3
250	ú	A,L,E,I	86	0	2
251	û	A,L,E,I	86	0	4
252	ü	A,L,E,I	86	0	5
253	ý	A,L,E,I	90	0	2
254	þ	A,L,E,I	92	0	1
255	ÿ	A,L,E,I	90	0	5

Note: unique collation requires that no two rows of this table have identical collation order columns.

Annex B · Error code translations
(informative)

`M1`	naked indicator undefined
`M2`	invalid combination with `P` *fncodatom*
`M3`	`$random` seed less than `1`
`M4`	no true condition in `$select`
`M5`	*lineref* less than zero
`M6`	undefined *lvn*
`M7`	undefined *gvn*
`M8`	undefined *svn*
`M9`	divide by zero
`M10`	invalid pattern match range
`M11`	no parameters passed
`M12`	invalid *lineref* (negative offset)
`M13`	invalid *lineref* (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 into itself
`M20`	*line* must have *formallist*
`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
`M25`	cannot `rsave` currently-executing routine
`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 *ssvn* not allowed by implementation
`M30`	reference to *glvn* with different collating sequence within a collating algorithm
`M31`	*controlmnemonic* used for device without a *mnemonicspace* selected
`M32`	*controlmnemonic* used in user-defined *mnemonicspace* which has no associated *line*
`M33`	`set` or `kill` to `^$routine` when *routine* exists
`M35`	device does not support *mnemonicspace*
`M36`	incompatible *mnemonicspace*s
`M37`	`read` from device identified by the empty string
`M38`	invalid *ssvn* subscript
`M39`	invalid `$name` argument
`M40`	call-by-reference in `job` *actual*

`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
`M47`	invalid attribute value
`M56`	maximum name length exceeded
`M57`	more than one defining occurence of *label* in *routine*
`M58`	too few formal parameters
`M60`	undefined *ssvn*
`M75`	maximum string length exceeded
`M88`	cannot `rload` non-existent routine
`M90`	invalid *namevalue*
`M92`	numeric overflow
`M94`	zero to the power of zero
`M95`	complex result with imaginary part
`M96`	no write access to *ssvn*
`M97`	non-existent *routine* for user-defined *ssvn*
`M98`	resource unavailable
`M99`	invalid operation for context
`M100`	output time out expired
`M102`	incompatible event processing
`M103`	invalid event
`M104`	`etrigger` invalid job number
`M105`	inaccessible *object*
`M106`	invalid *service*
`M107`	no default value
`M108`	not an *object*
`M109`	undefined device attribute or keyword
`M110`	event ID cannot be registered (resource unavailable)

`S0`	general syntax error

Annex C · Metalanguage element dictionary (informative)

`::=`	definition
[ ]	optional element
| |	group of alternate choices
...	optional indefinite repetition
`@`∈{ → }	value (evaluates to)
actual	actual argument
actualkeyword	actual argument keyword
actuallist	actual argument list
actualname	actual argument name
algoref	algorithm reference
alternation	alternation
argument	argument of a command
assignargument	**`assign`** argument
assigndestination	**`assign`** destination
assignleft	**`assign`** left
binaryop	binary operator
charset	character set
charsetexpr	character set expression
charspec	character specification
closeargument	**`close`** argument
command	command
commands	commands separated by *cs*
commandword	command word
comment	comment
control	control character
controlmnemonic	control mnemonic
cr	carriage return character
cs	command separator
descriptor	character set name
descsep	character set name separator
device	device
deviceattribute	device attribute
devicekeyword	device keyword
deviceparam	device parameter
deviceparameters	device parameters
devicexpr	device expression
devn	device name
digit	decimal digit character
dlabel	indirect label (evaluated label)
doargument	**`do`** argument
ecode	error code
einfoattribute	event information attribute
einforef	event information reference
emptystring	empty string
entryref	entry reference
environment	set of distinct names
eoffset	error offset
eol	end-of-line
eor	end-of-routine
erspec	event restricted specification
espec	event specification
especref	event specification reference
evclass	event class
eventexpr	event expression
evid	event ID
exfunc	extrinsic function
exp	exponent
expr	expression
expratom	expression atom
expritem	expression item
exprtail	expression tail
externalroutinename	external routine name
externref	external reference
extid	external identifier
extractfields	**`$extract`** fields
extracttemplate	**`$extract`** template
extsyntax	external syntax
exttext	external text
exvar	extrinsic variable
fieldindex	**`$extract`** field index

fieldwidth	`$extract` field width
fncodatom	`$fnumber` code atom
fncode	`$fnumber` code
fncodexpr	`$fnumber` code expression
fncodp	`$fnumber` code `P`
fncodt	`$fnumber` code `T`
ff	form feed character
ffformat	form feed format
formalline	formal line (line with formal list)
formallist	formal argument list
format	I/O format code
forparameter	`for` argument
fservice	free *service*
function	intrinsic function
glvn	global or local variable name
gnamind	global name indirection
gotoargument	`goto` argument
graphic	graphic character (character with visible representation)
gvn	global variable name
gvnexpr	global variable name expression
hangargument	`hang` argument
ident	identification character
ifargument	`if` argument
initialrecordvalue	initial data-record value
intexpr	expression, value interpreted as an integer
intlit	integer literal
iocommand	i/o command
jobargument	`job` argument
jobenv	`job` environment
jobparameters	`job` parameters
killargument	`kill` argument
list{}	list (list of)
label	label of a line
labelref	label reference
leftexpr	left expression
leftrestricted	left restricted
levelline	level line (line without formal list)
lf	line feed character
li	level indicator
libdatatype	library data type
library	library
libraryelement	library element
libraryelementdef	library element definition
libraryelementexpr	library element expression
libraryexpr	library expression
libraryopt	library option
libraryparam	library parameter
libraryref	library reference
libraryresult	library result
line	line in routine
linebody	line body
lineref	line reference
lname	local name
lnamind	local name indirection
lockargument	`lock` argument
logicalop	logical operator
ls	label separator
lvn	local variable name
lvnexpr	local variable name expression
mant	mantissa
mergeargument	`merge` argument
mnemonicspace	mnemonic space
mnemonicspacename	mnemonic space name
mnemonicspec	mnemonic space specifier
mumpsreturn	`$mumps` return value
mval	value of data type *mval*
name	name
namedactual	named actual argument
namedactuallist	named actual argument list
namevalue	name value
newargument	`new` argument
newsvn	`new` special variable name
nlformat	new line format
noncomma	non-comma character
noncommasemi	non-comma, non-semicolon character
nonquote	non-quote character (any

	graphic character not equal to quote)
nref	name reference
nrefind	name reference indirection
numexpr	expression, value interpreted numerically
numlit	numeric literal
object	*`object`* expression atom
openargument	**`open`** argument
openparameters	**`open`** parameters
oref	value of data type *`oref`*
owmethod	*`object`* with *`method`*
owproperty	*`object`* with *`property`*
owservice	*`object`* with *`service`*
packagename	package name
patatom	pattern atom
patcode	pattern code
patnony	pattern non-**`Y`** character
patnonyz	pattern non-**`Y`**-or-**`Z`** character
patnonz	pattern non-**`Z`** character
patsetdest	pattern **`set`** destination
patstr	pattern string literal
pattern	pattern
piecedelimiter	**`$piece`** delimiter
place	place
positionformat	position format
postcond	post condition
processid	process identifier
processparameters	process parameters
readargument	**`read`** argument
readcount	**`read`** count
recordfieldglvn	data-record field global or local variable name
recordfieldvalue	data-record field value
relation	relational operator
repcount	repeat count in pattern atom
restartargument	restart argument
rexpratom	restricted expression atom
rgvn	restricted global variable name

rlvn	restricted local variable name
rnref	restricted name reference
routine	routine
routineargument	**`rload`**/**`rsave`** argument
routineattribute	**`rload`**/**`rsave`** attribute
routinebody	routine body
routinehead	routine head
routinekeyword	**`rload`**/**`rsave`** keyword
routinename	routine name
routineparam	**`rload`**/**`rsave`** parameter
routineparameters	**`rload`**/**`rsave`** parameters
routineref	routine reference
routinexpr	routine expression
rssvn	restricted structured system variable name
servicename	*`service`* name
setargument	**`set`** argument
setdestination	**`set`** destination
setdextract	**`set $dextract`**
setdpiece	**`set $dpiece`**
setev	**`set`** error variable
setextract	**`set $extract`**
setleft	**`set`** left
setpiece	**`set $piece`**
sp	space character
ssvn	structured system variable name
ssvname	name portion of structured system variable name
ssvnamind	structured system variable name indirection
stackcode	**`$stack`** code
stackcodexpr	**`$stack`** code expression
strconst	string constant
strlit	string literal
sublit	subscript literal
subnonquote	subscript non-quote character
svn	special variable name
system	system
systemexpr	system expression

tabformat	tab format
textarg	**`$text`** argument
timeout	time-out specification
transparameters	transaction parameters
truthop	truth operator
tsparam	**`tstart`** parameter
tstartargument	**`tstart`** argument
tvexpr	expression, value interpreted as a truth-value
unaryop	unary operator
useargument	**`use`** argument
wevclass	windowing API event class
writeargument	**`write`** argument
xargument	**`xecute`** argument

Annex D · Embedded SQL
(informative)

SQL2 provides a capability for supporting embedded SQL M programs. The specification for this is described in *ANSI X3.135* (*ISO/IEC 9075, 1992*) and is partially reproduced here for information purposes. The reader should refer to *ANSI X3.135* Section 19, "Embedded SQL" for the full definition.

"19.1 `<embedded SQL host program>`

. . .

Syntax Rules

1 An **`<embedded SQL host program>`** is a compilation unit that consists of programming language text and SQL text. The programming language text shall conform to the requirements of a specific standard programming language. The SQL text shall consist of one or more **`<embedded SQL statement>`**s and, optionally, one or more **`<embedded SQL declare section>`**s, as defined in this standard.

2 An **`<embedded SQL statement>`**, **`<embedded SQL begin declare>`**, or **`<embedded SQL end declare>`** that is contained in an **`<embedded SQL MUMPS program>`** shall contain an **`<SQL prefix>`** that is **`"<ampersand>SQL<open paren>"`**. There shall be no **`<separator>`** between the **`<ampersand>`** and **`"SQL"`** nor between **`"SQL"`** and the **`<open paren>`**.

. . .

3 . . .

An **`<embedded SQL statement>`**, **`<embedded SQL begin declare>`**, or **`<embedded SQL end declare>`** that is contained in an **`<embedded SQL MUMPS program>`** shall contain an **`<SQL terminator>`** that is a **`<close paren>`**.

4 The **`<token>`**s comprising an **`<SQL prefix>`**, **`<embedded SQL begin declare>`**, or **`<embedded SQL end declare>`** shall be separated by **`<space>`** characters and be specified on one line. Otherwise, the rules for the continuation of lines and tokens from one line to the next and for the placement of host language comments are those of the programming language of the containing **`<embedded SQL host program>`**.

. . .

19.7 `<embedded SQL MUMPS program>`

Function

Specify an **`<embedded SQL MUMPS program>`**

Format

```
<embedded SQL MUMPS program> ::= !! See the Syntax Rules.


<MUMPS host identifier> ::= !! See the Syntax Rules.
```

```
<MUMPS variable definition> ::=
   { <MUMPS numeric variable> | <MUMPS character variable> } <semicolon>

<MUMPS character variable> ::=
   VARCHAR <MUMPS host identifier> <MUMPS length specification>
   [ { , <MUMPS host identifier> <MUMPS length specification> }... ]

<MUMPS length specification> ::=
   <open paren> <length> <close paren>

<MUMPS numeric variable> ::=
   { INT
   | DEC [ { <precisions> [ , <scale> ] ) ]
   | REAL }
     <MUMPS host identifier> [ { , <MUMPS host identifier> }... ]
```

Syntax Rules

1 An **`<embedded SQL MUMPS program>`** is a compilation unit that consists of MUMPS text and SQL text. The MUMPS text shall conform to standard MUMPS. The SQL text shall consist of one or more **`<embedded SQL statement>`**s and, optionally, one or more **`<embedded SQL declare section>`**s.

2 A **`<MUMPS host identifier>`** is any valid MUMPS variable name. A **`<MUMPS host identifier>`** shall be contained in an **`<embedded SQL MUMPS program>`**.

3 An **`<embedded SQL statement>`** may be specified wherever a MUMPS command may be specified.

4 A **`<MUMPS variable definition>`** defines one or more host variables.

5 The **`<MUMPS character variable>`** defines a variable-length string. The equivalent SQL data type is **`VARCHAR`** whose maximum length is the **`<length>`** of the **`<MUMPS length specification>`**.

6 **`INT`** describes an exact numeric variable. The equivalent SQL data type is **`INTEGER`**.

7 **`DEC`** describes an exact numeric variable. The **`<scale>`** shall not be greater than the **`<precision>`**. The equivalent SQL data type is **`DECIMAL`** with the same **`<precision>`** and **`<scale>`**.

8 **`REAL`** describes an approximate numeric variable. The equivalent SQL data type is **`REAL`**.

9 An **`<embedded SQL MUMPS program>`** shall contain either a variable named **`SQLCODE`** defined with a datatype of **`INT`** or a variable named **`SQLSTATE`** defined with a datatype that is **`VARCHAR`** with length **`5`**, or both.

Note: *`SQLSTATE` is the preferred status parameter. The `SQLCODE` status parameter is a deprecated feature that is supported for compatibility with earlier versions of this standard. See Annex D, "Deprecated Features".*

. . . ”

Annex E · Transportability of M software systems
(informative)

The transfer of *routine*s between machine environments is affected by numerous machine and operating systems factors. A standard transfer format for both routines and data stored within globals cannot at the same time easily cope with the simple and the complex case efficiently, in addition to dealing with the environmental idiosyncrasies. Therefore, the responsibility for the detailed format is left to the transferor.

1 Routine transfer format

The routine loader routine shall have a form that will load the routines from the transfer medium and will save it in internal format. The save routine creating the transfer medium shall produce the following routine transfer format:

```
header-line-1 eol
header-line-2 eol
routinehead
routine-line eol
.
.
.
eol
routinehead
routine-line eol
.
.
.
eol
[ ***RTN END*** ] eol
```

In the above structure, *routine-line* is a string in a format as returned by **$TEXT**. The two *header-line*s shall be free text and may contain any message the sender wishes to convey to the receiver.

Note: Each routine is separated by a blank line (an *eol*) from the following one. Optionally, either two successive blank lines or the string **"***RTN END***"** denotes the end of the file. *Eol* is defined to be a logical end-of-line record as mutually defined by the sending and receiving environments.

2 Global dump formats

The global loader shall read and the global dumper shall produce on the transfer medium the following transfer format:

```
header-line-1 eol
header-line-2 eol
full-global-reference eol
data-contents eol
.
.
.
full-global-reference eol
data-contents eol
eol
[ ***GBL END*** ] eol
```

Eol is defined to be a logical end-of-line record as mutually defined by the sending and receiving environments.

The `full-global-reference` shall conform to a global variable name specification as defined by *ANSI/MDC X11.1-1995* section 1, subclause 7.1.3.4. When `data-contents` contains ASCII control characters, decimal **`0`**–**`31`** or **`127`**, the user shall be responsible for handling the accurate reconstruction of the data string in the host environment. Subscripts in `full-global-reference` shall not contain the ASCII control characters decimal **`0`**–**`31`** or **`127`**. Optionally, either two successive blank lines or the string **`"***GBL END***"`** denotes the end of the file.

3 Transfer media

If the medium is magnetic tape, it should preferably be ½" industry standard 9 track (unlabelled and with the ASCII character set). The eighth bit (i.e., most significant bit) shall be set to zero when transferring 7 bit data. The physical block size shall be preferably 1024 characters but may be any clearly designated integer multiple of this size and preferably use the *ANSI X3.27-1987* “D” (unspanned variable length records) format. The reel size should preferably be 7".

Tapes recorded at either 800 bpi, NRZI (*ANSI X3.22-1990*) or 1600 or 6250 bpi phase-encoded (*ANSI X3.39-1992*) are recommended for current systems.

Annex F · X3.64 controlmnemonics
(informative)

Control-mnemonic	Control Function
APC	Application Program Command
CBT	Cursor Backward Tabulation
CCH	Cancel Character
CHA	Cursor Horizontal Absolute
CHT	Cursor Horizontal Tabulation
CNL	Cursor Next Line
CPL	Cursor Preceding Line
CPR	Cursor Position Report
CTC	Cursor Tabulation Control
CUB	Cursor Backward
CUD	Cursor Down
CUF	Cursor Forward
CUP	Cursor Position
CUU	Cursor Up
CVT	Cursor Vertical Tabulation
DA	Device Attributes
DAQ	Define Area Qualification
DCH	Delete Character
DCS	Device Control String
DL	Delete Line
DMI	Disable Manual Input
DSR	Device Status Report
EA	Erase in Area
ECH	Erase Character
ED	Erase in Display
EF	Erase in Field
EL	Erase in Line
EMI	Enable Manual Input
EPA	End of Protected Area
ESA	End of Selected Area
FNT	Font Selection
GSM	Graphic Size Modification
GSS	Graphic Size Selection
HPA	Horizontal Position Absolute
HPR	Horizontal Position Relative
HTJ	Horizontal Tab with Justify
HTS	Horizontal Tabulation Set
HVP	Horizontal and Vertical Position
ICH	Insert Character
IL	Insert Line
IND	Index
INT	Interrupt
JFY	Justify
MC	Media Copy
MW	Message Waiting
NEL	Next Line
NP	Next Page
OSC	Operating System Command
PLD	Partial Line Down
PLU	Partial Line Up
PM	Privacy Message
PP	Preceding Page
PU1	Private Use One
PU2	Private Use Two
QUAD	QUAD
REP	Repeat
RI	Reverse Index
RIS	Reset to Initial State
RM	Reset Mode
SEM	Select Editing Extent Mode
SGR	Select Graphic Rendition
SL	Scroll Left
SM	Set Mode
SPA	Start of Protected Area
SPI	Spacing Increment
SR	Scroll Right
SS2	Single Shift Two
SS3	Single Shift Three
SSA	Start of Selected Area
ST	String Terminator
STS	Set Transmit State
SU	Scroll Up
TBC	Tabulation Clear
TSS	Thin Space Specification
VPA	Vertical Position Absolute
VPR	Vertical Position Relative
VTS	Vertical Tabulation Set

Annex G · charset `JIS90`
(informative)

(This is a partial English reproduction of the **`JIS90`** *charset*. The reader should refer to *JIS X0201-1990* and *JIS X0208-1990* for the full definition.)

1 Charset `JIS90`

The *charset* **`JIS90`** is defined using the *JIS X0201-1990 8-bit Code* and the *JIS X0208-1990 2-Byte Code for Information Interchange.*

2 *JIS X0201-1990*

In *JIS X0201-1990,* the values of decimal and character are the same as those from ASCII (*X3.4-1990*) in the range between decimal **`0`**–**`127`**, except decimal **`92`** which represents **`"¥"`** (yen) instead of **`"\"`** and Decimal **`126`** which represents **`"_"`** (overline) instead of **`"~"`** (tilde).

The *patcode*s defined in *charset* **`M`** as **`A`**, **`C`**, **`E`**, **`L`**, **`N`**, **`P`**, and **`U`** apply in the same way in the range of decimal **`0`**–**`127`**.

In the decimal range between **`161`** and **`223`**, the values represent 8-bit katakana characters.

3 *JIS X0208-1990*

In *JIS X0208-1990,* the relation of decimal and character is obtained as following. Let c_1 and c_2 be the decimal values of the first byte and the second byte code for character, then the range of decimal code for both c_1 and c_2 is **`[33,127]`** and the decimal value of the character is c_1**`*256+`**c_2**`.`** Let *n* be a decimal and if there is no character assigned for *n* in *JIS X0208,* then **`$C(`***`n`***`)`** is a space as exemplified by **`$C(8481)`**.

4 Pattern codes

*Patcode*s **`E`** and (*ka,* **`$C(182)`**) apply for the characters in the decimal range **`161`**–**`223`**. *Patcode*s **`E`** and (*`zen,`* **`$C(16692)`**) apply for the characters in the decimal range **`8481`**–**`32382`**.

5 Characters used in names

Characters in the *charset* **`JIS90`** except **`$C(8481)`** may be defined as *ident*.

6 Collation

The collation scheme of *charset* `JIS90` is ordered by the `$A` value of the character, within each of *JIS X0201-1990* and *X0208-1990*.

Annex H · Sockets binding
(informative)

1 Introduction

Sockets are used to represent and manage a communication channel between two entities on a network. The channel can be connection-oriented, in which the two entities establish a session for the duration of the conversation, or it can be connectionless, in which messages are simply sent out to the intended recipient.

2 General

Socket communications are accessed by the use of *controlmnemonic*s and *deviceparameters*. This binding uses the **SOCKET** *mnemonicspace*.

Socket identifiers (referred to simply as “sockets”) are used by the implementation to identify the “socket handle” used by the underlying implementation. The actual mapping between socket identifiers and the underlying sockets is implementation-specific.

Sockets are accessed and manipulated via a socket device. The socket device can contain a collection of sockets. At any one time, a single socket from the collection is the current socket. Any socket in the collection can be designated to be the current socket. Furthermore, sockets can be attached (added) and detached (deleted) from the collection of sockets.

3 Commands and Deviceparameters

For the **SOCKET** *mnemonicspace*, the following *deviceattributes* are defined. All *deviceattribute*s and *devicekeyword*s beginning with the letter **Z** (or **z**) are reserved for the implementation. All others are reserved. Names differing only in the use of corresponding upper and lower case letters are equivalent.

Once a device is successfully **open**ed, the structured system variable **^$device** reflects the current settings of the *deviceattribute*s.

An attempt to modify a socket when none is current will result in an error with *ecode*=**"M99"** (invalid operation for context). An attempt to specify an invalid argument to a *deviceattribute* will result in an error with *ecode*=**"M47"** (invalid attribute value).

3.1 Open and use commands

The **open** and **use** commands allow sockets to be associated with devices after specifying a *mnemonicspace* equal to **"SOCKET"**.

The following *deviceattribute*s are valid on an **open** or **use** command.

3.1.1 Attach

attach=*expr*

Expr specifies an implementation-specific socket identifier. It specifies an existing socket that should be added to this device's collection of sockets. If the socket is attached to any other process or device, the **attach** will fail with *ecode*=**"M98"** (resource unavailable). Otherwise, if the operation is successful, the attached socket will become the current socket for the device.

3.1.2 Connect

connect=*expr*

Expr specifies implementation-specific connection information. A client connection will be established with a server, using the connection information to locate the server. A new socket will be allocated for the client connection and will become the current socket for the device.

3.1.3 Delimiter

delimiter= | *expr*
 (*list*{*expr*}**)** |

Expr specifies an I/O delimiter. Each usage of this *deviceattribute* replaces the existing set of I/O delimiters with a new set (which may be empty). The set is empty if all *expr*s have the value of the empty string.

If no **delimiter** is specified, the initial set of I/O delimiters for the socket is empty.

3.1.4 Ioerror

ioerror=*expr*

expr specifies the I/O error trapping mode.

A value equal to **"notrap"** specifies that I/O errors on a device do not raise error conditions. A value equal to **"trap"** specifies that I/O errors on a device do raise error conditions with an *ecode* value associated with the error. Values beginning with **Z** (or **z**) are reserved for the implementation. All other values are reserved. Values differing only in the use of corresponding upper and lower case letters are equivalent.

If no **ioerror** is specified, the initial I/O error trapping mode for a socket is **"notrap"**.

3.1.5 `Listen`

`listen=`*expr*

Expr specifies implementation and protocol specific information. This command causes the device to allocate a new socket and prepare it for listening for incoming requests for connection to a server. The new socket is made the current socket for the device. Requests for connections will not be accepted until a **`write /`***controlmnemonic* is issued.

The following *deviceattribute*s are valid on the **`use`** but not the **`open`** command.

3.1.6 `Detach`

`detach=`*expr*

Expr specifies an implementation-specific socket identifier. The specified socket is detached from the device without affecting the socket's existing connection. The socket may then be attached to another socket device using the **`attach`** *deviceattribute*.

3.1.7 `Socket`

`socket=`*expr*

Expr specifies an implementation-specific socket identifier. The specified socket becomes the current socket.

3.2 `Close command`

The following *deviceattribute*s are valid on the **`close`** command.

3.2.1 `Socket`

`socket=expr`

Expr specifies an implementation-specific socket identifier which is the socket associated with the device that is to be closed. If this *deviceattribute* is specified then any other sockets associated with the device are not closed and the device is not released.

If the **`socket`** deviceattribute is omitted then all sockets associated with the device are closed and the device is released.

3.3 `Read command`

The **`read`** command may be used to obtain data from a socket.

A **read** operation will terminate if any of the following are detected, in the order specified:

- Error condition. **$device** reflects the error, **$key** is assigned the empty string. The value returned by the **read** command is implementation specific.
- **Read** timeout. **$key** is assigned the empty string. The **read** command returns data received up to the timeout.
- **Read** delimiter. **$key** is assigned the delimiter string which terminated the **read**. The **read** command returns data received up to, but not including, the delimiter.
- Fixed-length **read** requirements are satisfied. This occurs only after the specified number of characters are received. **$key** is assigned the empty string. The **read** command returns the characters received.
- For a stream-oriented protocol, when the buffer is empty the **read** waits. When there is at least one character, the **read** command returns available characters, up to the maximum string length for the implementation. Note that the number of characters returned is not predictable except to be within the range from one to the maximum string length. **$key** is assigned the empty string.
- For a message-oriented protocol, when a complete message is received, **read** returns the message. **$key** is assigned the empty string.

For multi-character I/O delimiters, the possibility exists due to the stream nature of transmissions, that characters which would otherwise match an I/O delimiter may actually be spread across multiple “packets.” In the event that the last *n* characters received (*n*>0) match a prefix of one or more I/O delimiters, the implementation must determine if any of the additionally expected characters complete the match with the I/O delimiter(s). One implementation would be to internally issue a timed **read**. A timeout of this internally issued timed **read** does not affect **$test**. The time associated with this internal timed **read** is implementation specific and is not included in the timeout which may have been optionally specified on the actual **read** command.

3.4 Write command

The **write** command may be used to send data to a socket.

Data being transmitted is sent using the urgency mode currently in effect for the socket. The definition and usage of urgency mode is implementation-specific.

Write ! appends the first I/O delimiter (see 3.1.3), if specified, to the internal output buffers for the current device. The process then immediately transfers the internally buffered output data to the underlying binding services. This command does not affect internally buffered input data. **$x** is set to **0**, **$y** is incremented by 1.

Write # causes the process to immediately transfer any internally buffered output data for the current device to the underlying binding services. No I/O delimiters are implicitly added to the internal output buffer. This command does not affect internally buffered input data. **$x** and **$y** are set to 0.

4 Controlmnemonics

controlmnemonic names differing only in the use of corresponding upper and lower case letters are equivalent.

4.1 Listen

listen[**(***expr***)**]

The use of this *controlmnemonic* causes the process to establish a queue depth for incoming client connections.

If *expr* is omitted then the queue depth established will take on an implementation specific value.

4.2 Wait

wait[**(***numexpr***)**]

Numexpr is a timeout value.

If the optional *numexpr* is present, the value must be nonnegative. If it is negative, the value `0` is used. *numexpr* denotes a *t*-second timeout, where *t* is the value of *numexpr*. If *t*=`0`, the condition is tested. If *t* is positive, execution is suspended until the connection is made, but in any case no longer than *t* seconds.

The use of this *controlmnemonic* causes the process to wait for an event to occur on any socket associated with the device, subject to timeout. When this operation completes, **$key** contains a value identifying the event that occurred.

In the event of a timeout or an error the empty string is returned in **$key**.

If a listening server socket receives a connection request, **$key** will contain the value "**CONNECT**". A new socket will be allocated to handle the connection with the client, and the new socket will become the current socket of the device.

If a message is received by a connectionless protocol, **$key** will contain the value **read**. The socket which received the message will become the current socket of the device.

5 ^$device

The following nodes are defined in **^$device** for the **SOCKET** mnemonicspace:

^$device(*device***,"SOCKET")=***intexpr*

Each device has a collection of sockets associated with it. Each new socket is identified by a socket identifier which is assigned an index number in the collection of sockets. This node of **^$device** defines the index number of the current socket.

^$device(*device***,"SOCKET",***index***,"DELIMITER")=***intexpr*

This provides the number of I/O delimiters, as defined using the **delimiter** *deviceattribute*, in effect for the device/socket. (See 3.1.3)

^$device(*device***,"SOCKET",***index***,"DELIMITER",***n***)=***expr*

This provides the *n*th I/O delimiter string. (See 3.1.3)

^$device(*device***,"SOCKET",***index***,"IOERROR")=***expr*

I/O error trapping mode. (See 3.1.4)

^$device(*device***,"SOCKET",***index***,"LOCALADDRESS")=***expr*

This provides the local network node address of the connection.

^$device(*device***,"SOCKET",***index***,"PROTOCOL")=***expr*

This provides the network protocol used for the connection.

^$device(*device***,"SOCKET",***index***,"REMOTEADDRESS")=***expr*

This provides the remote network node address of the connection.

^$device(*device***,"SOCKET",***index***,"SOCKETHANDLE")=***expr*

The value of this node is an implementation-specific string that provides the socket identifier of the indicated socket.

Annex I · Mumps standard library sample code (informative)

Note: The Mumps code that approximates any function in the following definitions in this annex only serves as an example of a possible implementation of that library function. Implementors are encouraged to provide implementations that offer better efficiency as well as greater accuracy.

1 CHARACTER library sample code

1.1 $%COLLATE^CHARACTER

```
COLLATE(A,CHARMOD)  ;
  new x set x=""
  if $get(CHARMOD)'="" do
  .  if $extract(CHARMOD,1)=""" do
  .  .  set x=$extract(CHARMOD,2,$length(CHARMOD))
  .  .  if x'="" set x=$get(^$global(x,"COLLATE"))
  .  if x="" set x=$get(^$character(CHARMOD,"COLLATE")) if x="" set
      x=^$job($job,"COLLATE")
  set x=@(x_"("_A_")")
  quit x
```

1.2 $%COMPARE^CHARACTER

```
COMPARE(A,B,CHARMOD)  ;
  new x,y
  ; assume current collation, i.e. ]] , if no CHARMOD specified
  if $get(CHARMOD)="" quit $select(A=B:0,A]]B:1,1:-1)
  ; otherwise need to override it and do string compare
  ; on collation value
  set x=$%COLLATE(A,CHARMOD),y=$%COLLATE(B,CHARMOD)
  quit $select(x=y:0,x]y:1,1:-1)
```

2 MATH library sample code

2.1 $%ABS^MATH

```
ABS(X)  quit $translate(+X,"-")
```

2.2 $%ARCCOS^MATH

```
ARCCOS(X)  ;
  ; Comment: This version of the function is optimized for speed, not for
      precision. The 'precision' parameter is not supported, and the precision
      is at best 2 in 10**-8.
  new A,N,R,SIGN,XX
  if X<-1 set $ecode=",M28,"
  if X>1 set $ecode=",M28,"
  set SIGN=1 set:X<0 X=-X,SIGN=-1
  set A(0)=1.5707963050,A(1)=-0.2145988016,A(2)=0.0889789874
  set A(3)=-0.0501743046,A(4)=0.0308918810,A(5)=-0.0170881256
  set A(6)=0.0066700901,A(7)=-0.0012624911
  set R=A(0),XX=1 for N=1:1:7 set XX=XX*X,R=A(N)*XX+R
  set R=$%SQRT^MATH(1-X,11)*R
  quit R*SIGN

ARCCOS(X,PREC)  ;
  new L,LIM,K,SIG,SIGS,VALUE
  if X<-1 set $ecode=",M28,"
  if X>1 set $ecode=",M28,"
  set PREC=$get(PREC,11)
  if $translate(X,"-")=1 quit 0
  set SIG=$select(X<0:-1,1:1),VALUE=1-(X*X)
  set X=$%SQRT^MATH(VALUE,PREC)
  if $translate(X,"-")=1 do  quit VALUE
  .  set VALUE=$%PI^MATH()/2*X
  .  quit
  if X>0.9 do  quit VALUE
  .  set SIGS=$select(X<0:-1,1:1)
  .  set VALUE=1/(1/X/X-1)
  .  set X=$%SQRT^MATH(VALUE,PREC)
  .  set VALUE=$%ARCTAN^MATH(X,PREC)*SIGS
  .  quit
  set (VALUE,L)=X
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 do  quit:$translate(L,"-")<LIM
  .   set L=L*X*X*(K-2)/(K-1)*(K-2)/K,VALUE=VALUE+L
  .   quit
  quit $select(SIG<0:$%PI^MATH()-VALUE,1:VALUE)
```

2.3 $%ARCCOSH^MATH

```
ARCCOSH(X,PREC)  ;
  if X<1 set $ecode=",M28,"
  new SQ
  set PREC=$get(PREC,11)
  set SQ=$%SQRT^MATH(X*X-1,PREC)
  quit $%LOG^MATH(X+SQ,PREC)
```

2.4 $%ARCCOT^MATH

```
ARCCOT(X,PREC)  ;
  set PREC=$get(PREC,11)
  set X=1/X
  quit $%ARCTAN^MATH(X,PREC)
```

2.5 $%ARCCOTH^MATH

```
ARCCOTH(X,PREC)  ;
  new LI,L2
  set PREC=$get(PREC,11)
  set L1=$%LOG^MATH(X+1,PREC)
  set L2=$%LOG^MATH(X-1,PREC)
  quit L1=L2/2
```

2.6 $%ARCCSC^MATH

```
ARCCSC(X,PREC)  ;
  set PREC=$get(PREC,11)
  set X=1/X
  quit $%ARCSIN^MATH(X,PREC)
```

2.7 $%ARCSEC^MATH

```
ARCSEC(X,PREC)  ;
  set PREC=$get(PREC,11)
  set X=1/X
  quit $%ARCCOS^MATH(X,PREC)
```

2.8 $%ARCSIN^MATH

```
ARCSIN(X)  ;
```

```
  ; Comment: This version of the function is optimized for speed, not for
      precision. The “precision' parameter is not supported, and the precision
      is at best 2 in 10**-8.
  new A,N,R,SIGN,XX
  if X<-1 set $ecode=",M28,"
  if X>1 set $ecode=",M28,"
  set SIGN=1 set:X<0 X=-X,SIGN=-1
  set A(0)=1.5707963050,A(1)=-0.2145988016,A(2)=0.0889789874
  set A(3)=-0.0501743046,A(4)=0.0308918810,A(5)=-0.0170881256
  set A(6)=0.0066700901,A(7)=-0.0012624911
  set R=A(0),XX=1 for N=1:1:7 set XX=XX*X,R=A(N)*XX+R
  set R=$%SQRT^MATH(1-X,11)*R
  set R=$%PI^MATH()/2-R
  quit R*SIGN

ARCSIN(X,PREC)  ;
  new L,LIM,K,SIGS,VALUE
  set PREC=$get(PREC,11)
  if $translate(X,"-")=1 do  quit VALUE
  .  set VALUE=$%PI^MATH()/2*X
  .  quit
  if X>0.99999 do  quit VALUE
  .  set SIGS=$select(X<0:-1,1:1)
  .  set VALUE=1/(1/X/X-1)
  .  set X= $%SQRT^MATH(VALUE,PREC)
  .  set VALUE=$%ARCTAN^MATH(X,PREC)*SIGS
  .  quit
  set (VALUE,L)=X
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 do  quit:$translate(L,"-")<LIM
  .   set L=L*X*X*(K-2)/(K-1)*(K-2)/K,VALUE=VALUE+L
  .   quit
  quit VALUE
```

2.9 $%ARCSINH^MATH

```
ARCSINH(X,PREC)  ;
  if X<1 set $ecode=",M28,"
  new SQ
  set PREC=$get(PREC,11)
  set SQ=$%SQRT^MATH(X*X+1,PREC)
  quit $%LOG^MATH(X+SQ,PREC)
```

2.10 $%ARCTAN^MATH

```
ARCTAN(X,PREC)  ;
  new FOLD,HI,L,LIM,LO,K,SIGN,SIGS,SIGT,VALUE
  set PREC=$get(PREC,11)
  set LO=0.0000000001,HI=9999999999
  set SIGT=$select(X<0:-1,1:1),X=$translate(X,"-")
  set X=$select(X<LO:LO,X>HI:HI,1:X)
  set FOLD=$Select(X'<1:0,1:1)
  set X=$select(FOLD:1/X,1:X)
  set L=X,VALUE=$%PI^MATH()/2-(1/X),SIGN=1
  if X<1.3 do  quit VALUE
  .  set X=$select(FOLD:1/X,1:X),VALUE=1/((1/X/X)+1)
  .  set $%SQRT^MATH(VALUE,PREC)
  .  if $translate(X,"-")=1 do  quit
  .  .  set VALUE=$%PI^MATH()/2*X
  .  .  quit
  .  if X>0.9 do  quit
  .  .  set SIGS=$select(X<0:-1,1:1)
  .  .  set VALUE=1/(1/X/X-1)
  .  .  set X=$%SQRT^MATH(VALUE)
  .  .  set VALUE=$$ARCTAN(X,10)
  .  .  set VALUE=VALUE*SIGS
  .  .  quit
  .  set (VALUE,L)=X
  .  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  .  for K=3:2 do  quit:($translate(L,"-")<LIM)
  .  .   set L=L*X*X*(K-2)/(K-1)*(K-2)/K,VALUE=VALUE+L
  .  .   quit
  .  set VALUE=$select(SIGT<1:-VALUE,1:VALUE)
  .  quit
  set LIM=$select((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 do  quit:$translate(1/L,)<LIM
  .   set L=L*X*X,VALUE=VALUE+(1/(K*L)*SIGN)
  .   set SIGN=SIGN*-1
  .   quit
  set VALUE=$select(FOLD:$%PI^MATH()/2-VALUE,1:VALUE)
  set VALUE=$select(SIGT<1:-VALUE,1:VALUE)
  quit VALUE
```

2.11 $%ARCTANH^MATH

```
ARCTANH(X,PREC)  ;
  if X<-1 set $ecode=",M28,"
  if X>1 set $ecode=",M28,"
  set PREC=$get(PREC,11)
  quit $%LOG^MATH(1+X/(1-X),PREC)/2
```

2.12 $%CABS^MATH

```
CABS(Z)  ;
  new ZRE,ZIM
  set ZRE=+Z,ZIM=+$piece(Z,"%",2)
  quit $%SQRT^MATH(ZRE*ZRE+(ZIM*ZIM))
```

2.13 $%CADD^MATH

```
CADD(X,Y)  ;
  new XRE,XIM,YRE,YIM
  set XRE=+X,XIM=+$piece(X,"%",2)
  set YRE=+Y,YIM=+$piece(Y,"%",2)
  quit XRE+YRE_"%"_(XIM+YIM)
```

2.14 $%CCOS^MATH

```
CCOS(Z,PREC)  ;
  new E1,E2,IA
  set PREC=$get(PREC,11)
  set IA=$%CMUL^MATH(Z,"0%1")
  set E1=$%CEXP^MATH(IA,PREC)
  set IA=-IA_"%"_(-$piece(IA,"%",2))
  set E2=$%CEXP^MATH(IA,PREC)
  set IA=$%CADD^MATH(E1,E2)
  quit $%CMUL^MATH(IA,"0.5%0")
```

2.15 $%CDIV^MATH

```
CDIV(X,Y)  ;
  new D,IM,RE,XIM,XRE,YIM,YRE
  set XRE=+X,XIM=+$piece(X,"%",2)
  set YRE=+Y,YIM=+$piece(Y,"%",2)
  set D=YRE*YRE+(YIM*YIM)
  set RE=XRE*YRE+(XIM*YIM)/D
  set IM=XIM*YRE-(XRE*YIM)/D
  quit RE_"%"_IM
```

2.16 $%CEXP^MATH

```
CEXP(Z,PREC)  ;
  new R,ZIM,ZRE
  set PREC=$get(PREC,11)
  set ZRE=+Z,ZIM=+$piece(Z,"%",2)
  set R=$%EXP^MATH(ZRE,PREC)
  quit R*$%COS^MATH(ZIM,PREC)_"%"_(R*$%SIN^MATH(ZIM,PREC))
```

2.17 $%CLOG^MATH

```
CLOG(Z,PREC)  ;
  new ABS,ARG,ZIM,ZRE
  set PREC=$get(PREC,11)
  set ABS=$%CABS^MATH(Z)
  set ZRE=+Z,ZIM=+$piece(Z,"%",2)
  set ARG=$%ARCTAN^MATH(ZIM,ZRE,PREC)
  quit $%LOG^MATH(ABS,PREC)_"%"_ARG
```

2.18 $%CMUL^MATH

```
CMUL(X,Y)  ;
  new XIM,XRE,YIM,YRE
  set XRE=+X,XIM=+$piece(X,"%",2)
  set YRE=+Y,YIM=+$piece(Y,"%",2)
  quit XRE*YRE-(XIM*YIM)_"%"_(XRE*YIM+(XIM*YRE))
```

2.19 $%COMPLEX^MATH

```
COMPLEX(X)  quit +X_"%0"
```

2.20 $%CONJUG^MATH

```
CONJUG(Z)  ;
  new ZIM,ZRE
  set ZRE=+Z,ZIM=+$piece(Z,"%",2)
  quit ZRE_"%"_(-ZIM)
```

2.21 $%COS^MATH

```
COS(X)  ;
  ; Comment: This version of the function is optimized for speed, not for
      precision. The 'precision' parameter is not supported and the precision
      is at best 1 in 10**-9. Note that this function does not accept its
      parameter in degrees, minutes and seconds.
  new A,N,PI,R,SIGN,XX
  ;
  ; This approximation only works for 0≤x≤π/2
  ; so reduce angle to correct quadrant.
  ;
  set PI=$%PI^MATH(),X=X#(PI*2),SIGN=1 set:X>PI X=2*PI-X
  set:X*2>PI X=PI-X,SIGN=-1
  ;
  set XX=X*X,A(1)=-0.4999999963,A(2)=0.0416666418
  set A(3)=-0.0013888397,A(4)=0.0000247609,A(5)=-0.0000002605
  set (X,R)=1 for N=1:1:5 set X=X*XX,R=A(N)*X+R
  quit R*SIGN

COS(X,PREC)  ;
  ; Comment: The official description does not mention than the function may
      also be called with the first parameter in degrees, minutes and seconds.
  new L,LIM,K,SIGN,VALUE
  set:X[":" X=$%DMSDEC^MATH(X)
  set PREC=$Get(PREC,11)
  set X=X#(2*$%PI^MATH())
  set (VALUE,L)=1,SIGN=-1
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=2:2 do  quit:$translate(L,"–")<LIM set SIGN=SIGN*-1
  .   set L=L*X*X/(K-1*K),VALUE=VALUE+(SIGN*L)
  .   quit
  quit VALUE
```

2.22 $%COSH^MATH

```
COSH(X)  ;
  quit $%EXP^MATH(X)+$%EXP^MATH(-X)/2

COSH(X,PREC)  ;
  new E,F,I,P,R,T,XX
  set PREC=$get(PREC,11)+1
```

```
 set @("E=1E-"_PREC)
 set XX=X*X,F=1,(P,R,T)=1,I=1
 for  set T=T*XX,F=I+1*I*F,R=T/F+R,P=P-R/R,1=1+2 if -E<P,P<E quit
 quit R
```

2.23 $%COT^MATH

```
COT(X,PREC) ;
 ; Comment: The official description does not mention than the function may
   also be called with the first parameter in degrees, minutes and seconds.
 new C,L,LIM,K,SIGN,VALUE
 set:X[":" X=$%DMSDEC^MATH(X)
 set PREC=$get(PREC,11)
 set (VALUE,L)=1,SIGN=-1
 set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
 for K=2:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
 .  set L=L*X*X/(K-1*K),VALUE=VALUE+(SIGN*L)
 .  quit
 set C=VALUE
 set X=X#(2*$%PI^MATH())
 set (VALUE,L)=X,SIGN=-1
 set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
 for K=3:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
 .  set L=L/(K-1)*X/K*X,VALUE=VALUE+(SIGN*L)
 .  quit
 if 'VALUE quit "INFINITE"
 quit VALUE=C/VALUE
```

2.24 $%COTH^MATH

```
COTH(X,PREC)  ;
 new SINH
 if 'X quit "INFINITE"
 set PREC=$get(PREC,11)
 set SINH=$%SINH^MATH(X,PREC)
 if 'SINH quit "INFINITE"
 quit $%COSH^MATH(X,PREC)/SINH
```

2.25 $%CPOWER^MATH

```
CPOWER(Z,N,PREC)  ;
 new AR,NIM,NRE,PHI,PI,R,RHO,TH,ZIM,ZRE
```

```
  set PREC=$get(PREC,11)
  set ZRE=+Z,ZIM=+$piece(Z,"%",2)
  set NRE=+N,NIM=+$piece(N,"%",2)
  if 'ZRE,'ZIM,'NRE,'NIM set $ecode=",M28,"
  if 'ZRE,'ZIM quit "0%0"
  set PI=$%PI^MATH()
  set R=$%SQRT^MATH(ZRE*ZRE+(ZIM*ZIM),PREC)
  if ZRE set TH=$%ARCTAN^MATH(ZIM/ZRE,PREC)
  else  set TH=$select(ZIM>0:PI/2,1:-PI/2)
  set RHO=$%LOG^MATH(R,PREC)
  set AR=$%EXP^MATH(RHO*NRE-(TH*NIM),PREC)
  set PHI=RHO*NIM+(NRE*TH)
  quit AR*$%COS^MATH(PHI,PREC)_"%"_(AR*$%SIN^MATH(PHI,PREC))
```

2.26 $%CSC^MATH

```
CSC(X,PREC)  ;
  ; Comment: The official description does not mention than the function may
      also be called with the first parameter in degrees, minutes and seconds.
  new L,LIM,K,SIGN,VALUE
  set:X[":" X=$%DMSDEC^MATH(X)
  set PREC=$get(PREC,11)
  set X=X#(2*$%PI^MATH())
  set (VALUE,L)=X,SIGN=-1
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
  .   set L=L/(K-1)*X/K*X,VALUE=VALUE+(SIGN*L)
  .   quit
  if 'VALUE quit "INFINITE"
  quit 1/VALUE
```

2.27 $%CSCH^MATH

```
CSCH(X,PREC)
  quit 1/$%SINH^MATH(X,$get(PREC,11))
```

2.28 $%CSIN^MATH

```
CSIN(Z,PREC)  ;
  new IA,E1,E2
  set PREC=$get(PREC,11)
  set IA=$%CMUL^MATH(Z,"0%1")
  set E1=$%CEXP^MATH(IA,PREC)
  set IA=-IA_"%"_(-$piece(IA,"%",2))
  set E2=$%CEXP^MATH(IA,PREC)
  set IA=$%CSUB^MATH(E1,E2)
  set IA=$%CMUL^MATH(IA,".5%0")
  quit $%CMUL^MATH("0%-1",IA)
```

2.29 $%CSUB^MATH

```
CSUB(X,Y)  ;
  new XIM,XRE,YIM,YRE
  set XRE=+X,XIM=+$piece(X,"%",2)
  set YRE=+Y,YIM=+$piece(Y,"%",2)
  quit XRE-YRE_"%"_(XIM-YIM)
```

2.30 $%DECDMS^MATH

```
DECDMS(X,PREC)  ;
  set PREC=$get(PREC,5)
  set X=X#360*3600
  set X=+$justify(X,0,$select((PREC-$length(X\1))'<0:PREC-$length(X\1),1:0))
  quit X\3600_":"_(X\60#60)_":"_(X#60)
```

2.31 $%DEGRAD^MATH

```
DEGRAD(X)  quit X*3.14159265358979/180
```

2.32 $%DMSDEC^MATH

```
DMSDEC(X)  ;
  quit $piece(X,":")+($piece(X,":",2)/60)+($piece(X,":",3)/3600)
```

2.33 $%E^MATH

```
E()  quit 2.71828182845905
```

2.34 $%EXP^MATH

```
EXP(X,PREC)  ;
  new L,LIM,K,VALUE
  set PREC=$get(PREC,11)
  set L=X,VALUE=X+1
  set LIM=$select((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=2:1 set L=L*X/K,VALUE=VALUE+L quit:($translate(L, "-" )<LIM)
  quit VALUE
```

2.35 $%LOG^MATH

```
LOG(X,PREC)  ;
  new L,LIM,M,N,K,VALUE
  if X'>0 set $ecode=",M28,"
  set PREC=$get(PREC,11)
  set M=1
  for N=0:1 quit:X/M<10  set M=M*0.1
  if X<1 for N=0:-1 quit:X/M>0.1  set M=M*0.1
  set X=X/M
  set X=(X-1)/(X+1),(VALUE,L)=X
  set LIM=$select((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 set L=L*X*X,M=L/K,VALUE=M+VALUE set:M<0 M=-M quit:M<LIM
  set VALUE=VALUE*2+(N*2.30258509298749)
  quit VALUE
```

2.36 $%LOG10^MATH

```
LOG10(X,PREC)  ;
  new L,LIM,M,N,K,VALUE
  if X'>0 set $ecode=",M28,"
  set PREC=$get(PREC,11)
  set M=1
  for N=0:1 quit:X/M<10  set M=M*10
  if X<1 for N=0:-1 quit:X/M>0.1  set M=M*0.1
  set X=X/M
  set X=(X-1)/(X+1),(VALUE,L)=X
  set LIM=$select((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 set L=L*X*X,M=L/K,VALUE=M+VALUE set:M<0 M=-M quit:M<LIM
  set VALUE=VALUE*2+(N*2.30258509298749)
  quit VALUE/2.30258509298749
```

2.37 $%MTXADD^MATH

```
MTXADD(A,B,R,ROWS,COLS)  ; add A[ROWS,COLS] to B[ROWS,COLS],
  ; result goes to R[ROWS,COLS]
  if $data(A)<10 quit 0
  if $data(B)<10 quit 0
  if $get(ROWS)<1 quit 0
  if $get(COLS)<1 quit 0
  ;
  new ROW,COL,ANY
  for ROW=1:1:ROWS for COL=1:1:COLS do
  .   kvalue R(ROW,COL) set ANY=0
  .   set:$data(A(ROW,COL))#2 ANY=1
  .   set:$data(B(ROW,COL))#2 ANY=1
  .   set:ANY R(ROW,COL)=$get(A(ROW,COL))+$get(B(ROW,COL))
  . quit
  quit 1
```

2.38 $%MTXCOF^MATH

```
MTXCOF(A,I,K,N)  ; Compute cofactor for element [i,k] in matrix A[N,N]
  new T,R,C,RR,CC
  set CC=0
  for C=1:1:N do:C'=K
  .   set CC=CC+1,RR=0
  .   for R=1:1:N set:R*=I RR=RR+1,T(RR,CC)=$get(A(R,C))
  .   quit
  quit $%MTXDET^MATH(.T,N-1)
```

2.39 $%MTXCOPY^MATH

```
MTXCOPY(A,R,ROWS,COLS)  ; copy A[ROWS,COLS] to R[ROWS,COLS]
  if $data(A)<10 quit 0
  if $get(ROWS)<1 quit 0
  if $get(COLS)<1 quit 0
  ;
  new ROW,COL
  for ROW=1:1:ROWS for COL=1:1:COLS do
  .   kvalue R(ROW,COL)
  .   set:$data(A(ROW,COL))#2 R(ROW,COL)=A(ROW,COL)
  .   quit
  quit 1
```

2.40 $%MTXDET^MATH

```
MTXDET(A,N)  ; compute determinant of matrix A[N,N]
  if $data(A)<10 quit ""
  if N<1 quit ""
```

```
 ;
 ; First the simple cases
 ;
 if N=1 quit $get(A(1,1))
 if N=2 quit $get(A(1,1))*$get(A(2,2))-($get(A(1,2))*$get(A(2,1)))
 new DET,I,SIGN
 ;
 ; Det A = sum (k=1:n) element (i,k) *cofactor [i,k]
 ;
 set DET=0,SIGN=1
 for I=1:1:N do
 . set DET=$get(A(1,I))*$%MTXCOF^MATH(.A,1,I,N)*SIGN+DET
 . set SIGN=-SIGN
 . quit
 quit DET
```

2.41 $%MTXEQU^MATH

```
MTXEQU(A,B,R,N,M)  ; solve matrix equation A[M,M]*R[M,N]=B[M,N]
 if $get(M)<1 quit ""
 if $get(N)<1 quit ""
 if 'MTXDET(.A) quit 0
 ;
 new I,I1,J,J1,J2,K,L,T,TI,T2,TEMP,X
 ;
 set X=$%MTXCOPY^MATH(.A,.T,N,N)
 set X=$%MTXCOPY^MATH(.B,.R,N,M)
 ;
 ; reduction of matrix A
 ; steps of reduction are counted by index K
 for K=1:1:N-1 do
 . ;
 . ; search for largest coefficient of T (denoted by
 . ; TEMP) in first column of reduced system
 . set TEMP=0,J2=K
 . for J1=K:1:N do
 . . quit:$translate($get(T(J1,K)),"-")'>=$translate(TEMP,"-")
 . . set TEMP=T(J1,K),J2=J1
 . . quit
 . ;
 . ; exchange row number K with row number J2, if
 . ; necessary
 .   ;
 .   do:J2'=K
 .   .  ;
 .   .  for J=K:1:N do
 .   .  .   set T1=$get(T(K,J)),T2=$get(T(J2,J)) kill T(K,J),T(J2,J)
 .   .  .   if T1'="" set T(J2,J)=T1
 .   .  .   if T2'="" set T(K,J)=T2
 .   .  .   quit
 .   .  for J=1:1:M do
 .   .  .   set T1=$get(R(K,J)),T2=$get(R(J2,J)) kill R(K,J),R(J2,J)
 .   .  .   if T1'="" set R(J2,J)=T1
 .   .  .   quit
 .   .  if T2'=" set R(K,J)=T2
 .   .  .  quit
 .   .  quit
 .   ;
 .   ; actual reduction
 .   ;
 .   for I=K+1:1:N do
 .   .   for J=K+1:1:N do
 .   .   .   quit:'$get(T(K,K))
 .   .   .   set T(I,J)=-$get(T(K,J))*$get(T(I,K))/T(K,K)+$get(T(I,J))
 .   .   .   quit
 .   .   for J=1:1:M do
 .   .   .   quit:'$get(T(K,K))
 .   .   .   set R(I,J)=-$get(R(K,J))*$get(T(I,K))/T(K,K)+$get(R(I,J))
 .   .   .   quit
 .   .   quit
 .   quit
 ;
 ; backsubstitution
 ;
 for J=1:1:M do
 .   if $get(T(N,N)) set R(N,J)=$get(R(N,J))/T(N,N)
 .   if N-1>0 for I1=1:1:N-1 do
 .   .  set I=N-I1
 .   .  for L=I+1:1:N do
 .   .  .   set R(I,J)=-$get(T(I,L))*$get(R(L,J) )+$get(R(I,J))
 .   .  .   quit
 .   .  .   if $get(T(I,I)) set R(I,J)=$get(R(I,J))/$get(T(I,I))
 .   .  quit
 .   quit
```

```
  quit $%MTXDET^MATH(.R)
```

2.42 $%MTXINV^MATH

```
MTXINV(A,R,N)  ; invert A[N,N], result goes to R[N,N]
  if $data(A)<10 quit 0
  if $get(N)<1 quit 0
  ;
  new T,X
  set X=$%MTXUNIT^MATH(.T,N)
  quit $%MTXEQU^MATH(.A,.T,.R,N,N)
```

2.43 $%MTXMUL^MATH

```
MTXMUL(A,B,R,M,L,N)  ; multiply A[M,L] by B[L,N], result goes to R[M,N]
  if $data(A)<10 quit 0
  if $data(B)<10 quit 0
  if $get(L)<1 quit 0
  if $get(M)<1 quit 0
  if $get(N)<1 quit 0
  ;
  new I,J,K,SUM,ANY
  for I=1:1:M for J=1:1:N do
  .   set (SUM,ANY)=0
  .   kvalue R(I,J)
  .   for K=1:1:L do
  .   .   set:$data(A(I,K))#2 ANY=1
  .   .   set:$data(B(K,J))#2 ANY=1
  .   .   set SUM=$get(A(I,K))*$get(B(K,J))+SUM
  .   .   quit
  .   set:ANY R(I,J)=SUM
  .   quit
  quit 1
```

2.44 $%MTXSCA^MATH

```
MTXSCA(A,R,ROWS,COLS,S)  ; multiply A[ROWS,COLS] with the scalar S,
  ; result goes to R[ROWS,COLS]
  if $data(A)<10 quit 0
  if $get(ROWS)<1 quit 0
  if $get (COLS)<1 quit 0
  if '($data(S)#2) quit 0
  ;
  new ROW,COL
  for ROW=1:1:ROWS for COL=1:1:COLS do
  .   kvalue R(ROW,COL)
  .   set:$data(A(ROW,COL))#2 R(ROW,COL)=A(ROW,COL)*S
  .   quit
  quit 1
```

2.45 $%MTXSUB^MATH

```
MTXSUB(A,B,R,ROWS,COLS)  ; subtract B[ROWS,COLS] from A[ROWS,COLS],
  ; result goes to R[ROWS,COLS]
  if $data(A)<10 quit 0
  if $data(B)<10 quit 0
  if $get(ROWS)<1 quit 0
  if $get(COLS)<1 quit 0
  ;
  new ROW,COL,ANY
  for ROW=1:1:ROWS for COL=1:1:COLS do
  .   kvalue R(ROW,COL) set ANY=0
  .   set:$data(A(ROW,COL))#2 ANY=1
  .   set:$data(B(ROW,COL))#2 ANY=1
  .   set:ANY R(ROW,COL)=$get(A(ROW,COL))-$get(B(ROW,COL))
  .   quit
  quit 1
```

2.46 $%MTXTRP^MATH

```
MTXTRP(A,R,M,N)  ; transpose A[M,N], result goes to R[N,M]
  if $data(A)<10 quit 0
  if $get(M)<1 quit 0
  if $get(N)<1 quit 0
  ;
  new I,J,K,D1,VI,D2,V2
  for I=1:1:M+N-1 for J=1:1:I+1\2 do
  .   set K=I-J+1
  .   if K=J do  quit
  .   .  set V1=$get(A(J,J)),D1=$data(A(J,J))#2
  .   .  if J'>N,J'>M kvalue R(J,J) set:D1 R(J,J)=V1
  .   .  quit
  .   ;
  .   set V1=$get(A(K,J)),D1=$data(A(K,J))#2
  .   set V2=$get(A(J,K)),D2=$data(A(J,K))#2
  .   if K'>M,J'>N kvalue R(K,J) set:D2 R(K,J)=V2
  .   if J'>M,K'>N kvalue R(J,K) set:D1 R(J,K)=V1
  .   quit
  quit 1
```

2.47 $%MTXUNIT^MATH

```
MTXUNIT(R,N,SPARSE)  ; create a unit matrix R[N,N]
  if $get(N)<1 quit 0
  ;
  new ROW,COL
  for ROW=1:1:N for COL=1:1:N do
  .   kvalue R(ROW,COL)
  .   if $get(SPARSE) quit:ROW'=COL
  .   set R(ROW,COL)=$select(ROW=COL:1,1:0)
  .   quit
  quit 1
```

2.48 $%PI^MATH

```
PI()  quit 3.14159265358979
```

2.49 $%RADDEG^MATH

```
RADDEG(X)  quit X*180/3.14159265358979
```

2.50 $%SEC^MATH

```
SEC(X,PREC)  ;
  new L,LIM,K,SIGN,VALUE
  ; Comment:The official description does not mention than the function may
      also be called with the first parameter in degrees, minutes and seconds.
  set:X[":" X=$%DMSDEC^MATH(X)
  set PREC=$get(PREC,11)
  set X=X#(2*$%PI^MATH())
  set (VALUE,L)=1,SIGN=-1
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=2:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
  .   set L=L*X*X/(K-1*K),VALUE=VALUE+(SIGN*L)
  .   quit
  if 'VALUE quit "INFINITE"
  quit 1/VALUE
```

2.51 $%SECH^MATH

```
SECH(X,PREC)  quit 1/$%COSH^MATH(X,$get(PREC,11))
```

2.52 $%SIGN^MATH

```
SIGN(X)  quit $select(X<0:-1,X>0:1,1:0)
```

2.53 $%SIN^MATH

```
SIN(X)  ;
  ; Comment: This version of the function is optimized for speed, not for
      precision. The 'precision' parameter is not supported, and the precision
      is at best 1 in 10**-9. Note that this function does not accept its
      parameter in degrees, minutes and seconds.
  new A,N,PI,R,SIGN,XX
  ;
  ; This approximation only works for 0 <> x £ n/2
  ; so reduce angle to correct quadrant.
  ;
  set PI=$%PI^MATH(),X=X#(PI*2),SIGN=1
  set:X>PI X=2*PI-X,SIGN=-1
  set:X*2<PI X=PI-X
  ;
  set XX=X*X,A(1)=-0.4999999963,A(2)=0.0416666418
  set A(3)=-0.0013888397,A(4)=0.0000247609,A(5)=-0.0000002605
  set (X,R)=1 for N=1:1:5 set X=X*XX,R=A(N)*X+R
  quit R*SIGN

SIN(X,PREC)  ;
  ; Comment: The official description does not mention than the function may
      also be called with the first parameter in degrees, minutes and seconds.
  new L,LIM,K,SIGN,VALUE
  set:X[":" X=$%DMSDEC^MATH(X)
  set PREC=$get(PREC,11)
  set X=X#(2*$%PI^MATH())
  set (VALUE,L)=X,SIGN=-1
  set LIM=$select((PREC+3)'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=3:2 do  quit:$translate(L,)<LIM  set SIGN=SIGN*-1
  .   set L=L/(K-1)*X/K*X,VALUE=VALUE+(SIGN*L)
```

```
 .  quit
 quit VALUE
```

2.54 $%SINH^MATH

```
SINH(X)  ;
 quit $%EXP^MATH(X)-$%EXP^MATH(-X)/2

SINH (X,PREC)  ;
 new E,F,I,P,R,T,XX
 set PREC=$get(PREC,11)+1
 set @("E=1E-"_PREC)
 set XX=X*X,F=1,I=2,(P,R,T)=X
 for  set T=T*XX,F=I+1*I*F,R=T/F+R,P=P-R/R,I=1+2 if -E<P,P<E quit
 quit R
```

2.55 $%SQRT^MATH

```
SQRT(X,PREC)  ;
 if X<0 set $ecode=",M28,"
 if X=0 quit 0
 set PREC=$get(PREC,11)
 if X<1 quit 1/$%SQRT^MATH(1/X,PREC)
 new P,R,E
 set PREC=$get(PREC,11)+1
 set @("E=1E-"_PREC)
 set R=X
 for  set P=R,R=X/R+R/2,P=P-R/R if -E<P,P<E quit
 quit R
```

2.56 $%TAN^MATH

```
TAN(X,PREC)  ;
 ; Comment: The official description does not mention than the function may
   also be called with the first parameter in degrees, minutes and seconds.
 new L,LIM,K,S,SIGN,VALUE
 set:X[":" X=$%DMSDEC^MATH(X)
 set PREC=$get(PREC,11)
 set X=X#(2*$%PI^MATH())
 set (VALUE,L)=X,SIGN=-1
 Set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
 for K=3:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
```

```
  .   set L=L/(K-1)*X/K*X,VALUE=VALUE+(SIGN*L)
  .   quit
  set S=VALUE
  set X=X#(2*$%PI^MATH())
  set (VALUE,L)=1,SIGN=-1
  set LIM=$select(PREC+3'>11:PREC+3,1:11),@("LIM=1E-"_LIM)
  for K=2:2 do  quit:$translate(L,"-")<LIM  set SIGN=SIGN*-1
  .   set L=L*X*X/(K-1*K),VALUE=VALUE+(SIGN*L)
  .   quit
  if 'VALUE quit "INFINITE"
  quit S/VALUE
```

2.57 $%TANH^MATH

```
TANH(X,PREC)  ;
  set PREC=$get(PREC,11)
  quit $%SINH^MATH(X,PREC)/$%COSH^MATH(X,PREC)
```

3 STRING library sample code

3.1 $%CRC16^STRING

```
CRC16(string,seed)  ; CRC-16
  ; Polynomial=x^16+x^15+x^2+x^0
  new I,J,R
  if '$data(seed) set R=0
  else  if seed'<0,seed'>65535 set R=seed\1
  else  set $ecode=",M28,"
  for I=1:1:$length(string) do
  . set R=$$XOR($ascii(string,I),R,8)
  . for J=0:1:7 do
  . . if R#2 set R=$$XOR(R\2,40961,16)
  . . else  set R=R\2
  quit R
  ;
XOR(a,b,w)  ;
  new I,M,R
  set R=b
  set M=1
  for I=1:1:w set:a\M#2 R=R+$select(R\M#2:-M,1:M) set M=M+M
  quit R
```

3.2 $%CRC32^STRING

```
CRC32(string,seed)  ; CRC-32
  ; Polynomial=x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x^0
  new I,J,R
  if '$data(seed) set R=4294967295 ; 0xFFFFFFFF = 2^32 - 1
  else  if seed'<0,seed'>4294967295 set R=4294967295-seed
  else  set $ecode=",M28,"
  for I=1:1:$length(string) do
  . set R=$$XOR($ascii(string,I),R,8)
  . for J=0:1:7 do
  . . if R#2 set R=$$XOR(R\2,3988292384,32)
  . . else  set R=R\2
  quit 4294967295-R ; 32-bit ones complement
  ;
XOR(a,b,w)  ;
  new I,M,R
  set R=b
  set M=1
  for I=1:1:w set:a\M#2 R=R+$select(R\M#2:-M,1:M) set M=M+M
  quit R
```

3.3 $%CRCCCIT^STRING

```
CRCCCITT(string,seed)  ; CRC-CCITT
  ; Polynomial=x^16+x^12+x^5+x^0
  new I,J,R
  if '$data(seed) set R=65535 ; FFFF = 2^16 -1
  else  if seed'<0,seed'>65535 set R=seed\1
  else  set $ecode=",M28,"
  for I=1:1:$length(string) do
  . set R=$$XOR($ascii(string,I)*256,R,16)
  . for J=0:1:7 do
  . . set R=R+R
  . . quit:R<65536  ; (2^16)
  . . set R=$$XOR(4129,R-65536,13)
  quit R
  ;
XOR(a,b,w)  ;
  new I,M,R
  set R=b
  set M=1
  for I=1:1:w set:a\M#2 R=R+$select(R\M#2:-M,1:M) set M=M+M
  quit R
```

3.4 $%FORMAT^STRING

```
FORMAT(V,L)  ; FORMAT Library Function ;06:58 PM 5 Sep 1995; RCR
V  ; version 0.9 ; CHRIS.RICHARDSON@FORUM.SAIC.COM
  ;
  ; The routine contains an initiation segment which will be setup the
  ; first time the module is executed and ignored from that point on.
  ; The global structures, ^$system, and ^$format may be modified to
  ; reflect the cultural bias of the host system. This module is meant
  ; to be an initial attempt at the implementation of this function.
  ;
  new C,CD,CH,CS,DP,E,EX,FL,FM,FO
  new GL,GV1,GV2,GVL,GVH,GX,I,J,K,ST,TV,TY,V1,V2,VP
  ;
  ; Load up format Directives from ^$format or ^system("FORMAT")
  do:'$data(^$format) INFORM
  set (FM,K)="",EX=0,EXS="EXS"
  ;
  ; Extract the working values from the command string
  do PRELOAD
  ;
  ; Process the directives
  do EVALU8
  ; Error Handling
  do:EX ERROR
  xecute:$length(EXS) "K "_EXS
  quit K
  ; -----------------------------------------------------
  ; CM  - Command Array
  ; CS  - Command String
  ; DP  - Decimal Pointer
  ; EX  - Exit Flag
  ; EXS - KILL Exit String
  ; FL  - Field Length
  ; FM  - Format String
  ; FO  - Format Option Array
  ; K   - Return Output String
  ; L   - List of Directives
  ; ST  - String Extraction String
```

```
 ; V   - Input Value
 ; -----------------------------------------------
PRELOAD  ; LOAD THE DEFAULTS PRIOR TO THE APPLICATION OF DIRECTIVES
 set K=""
 ; Load System Defaults
 for  set K=$O(^$system("FORMAT",K)) quit:K="" do
 . set FO(K)=^$system("FORMAT",K)
 . quit
 ; Load Process Defaults
 for  set K=$get(^$format(K)) quit:K=""  set FO(K)=^$format(K) set
     (CS,L)=$get(L)
 ; Load foargument Overrides from the List of Directives
 ; 1) Tokenize the Laterals
 do:L[""""
 . set CS=""
 . for J=2:2:$length(L,"""") do
 . . set ST=$get(ST)+1,ST(ST)=$piece(L,"""",J)
 . . set:ST(ST)="" ST(ST)=""""
 . . set CS=CS_$piece(L,"""",J-1)_"%%"_ST_"%%"
 . . quit
 . set CS=CS_$piece(L,"""",J+1)
 . quit
 ; 2) Evaluate the Directives
 new C,L,X
 for J=1:1:$length(CS,":") do  quit:EX
 . set CD=$piece(CS,":",J)
 . set X=$piece(CD,"="),TV=$piece(CD,"=",2,999)
 . if X="" set EX=1 quit
 . ; Uppercase Symbol Names Only
 . set TY=$translate(X,"abcdefghijklm","ABCDEFGHIJKLM")
 . set TY=$translate(TY,"nopqrstuvwxyz","NOPQRSTUVWXYZ")
 . do
 . . ; To bullet proof, ESTABLISH AN ERROR TRAP TO ERREX
 . . if $length(TV,"%%")>1 do LOADTV set FC(TV)=TV quit
 . . if TV=""
 . . ; 3) Set the Directives in the FO array
 . . set FO(TY)=@TV
 . . quit
 . quit
 ; 4) Construct a KILL Exit String for directives not in default list
 new C,E
 set (C,E)=""
```

```
  for  set C=$order(FO(C)) quit:C=""  set @C=FO(C),E=E_C_","
  set EXS=E "EXS"
  set:$data(FM) FL=$length(FM)
  quit
  ; --------------------------------------------------
  ; DC  - Decimal Character
  ; DP  - Decimal Position
  ; EX  - Abnormal Condition Exit
  ; FL  - Format Mask
  ; GV1 - Integer Portion
  ; GV2 - Fractional Portion
  ; K   - Output Buffer
  ; NOD - No Decimal
  ; SV  - Sign Value (1 – Positive, 0 – Negative)
  ; V   - Input Value
EVALU8  ; Evaluate the input for loading into the output string
  new NOD
  set SV=1 ; ,NOD=$piece(FM,"d",2)=""
  set NOD='(FM("d"))
  set:V<0 SV=0
  set V1=$piece(V,"."),V2=$piece(V,".",2),(GV1,GV2)="" do:FM'=""
  . set FL=$length(FM),DP=$find(FM,"d")-1
  . set:DP<1 DP=$length(FM)
  . quit
  new C
  do GETV1,GETV2:'(NOD!EX)
  quit:EX
  ;
  set K=GV1,DC=GV2
  set:NOD K=GVT
  if $get(FC)'="" set:K[" " K=$translate(K," ",FC)
  set:$length(K)'=FL EX=1
  quit
  ; ------------------------------------------
GETV1  ; Get the integer portion of the value and lay it in GV1
  new CP,SP
  if $get(SL)'="" new SC set SC=$length(SL)
  set GVM=$piece(FM,"d"),GVL=$length(GVM),GL=0       ; 1)
  set:$E(V1)="-" V1=$extract(V1,2,999)              ; 2)
  set GV1=$justify("",GVL),VP=$length(V1),(CP,SP)=1 ; 3)
  ;
  : Rounding of Integer (NO DECIMAL PORTION)
  set:$piece(FM,"d",2)="" V1=$extract((V+.5)\1,1,$length(V1))
  for L=GVL:-1:1 set C=$extract(GVM,L) do  quit:EX    ; 4)
  . set GX=0
  . do TRANSV1
  . quit:GX'=EX
  . ;
  . set:GC'=" " $extract(GV1,L)=GC
  . quit
  set:VP EX=1
  quit
  ; -----------------------------------------
  ; GETV1;
  ; 1) Set the integer Portion of the Mask (GVM) and Length (GVL)
  ; 2) Get the absolute value of V1
  ; 3) Establish Blank Mask, GV1
  ; 4) Extract value for each position in the mask and set it
  ;
GETV2  Get the fractional portion of the value and lay it in GV2
  new CP,SP
  if $get(SR)'="" new SC set SC=SR
  set GVM=$piece(FM,"d",2),GVL=$length(GVM),GL=0,SP=1
  do:GVL<$length(V2)  ; Rounding of Decimal
  . new J,N
  . set N=$extract($translate($justify("",GVL),"
      ",0)_5_$translate($justify("",$L(V2))," ",0),0,$length(V2))
  . set V2=$extract(V2+N,1,$length(V2))
  . quit
  set GV2=$justify("",GVL),VP=1,CP=1
  for L=1:1:GVL set C=$extract(GVM,L) do  quit:EX
  . set GX=0
  . do TRANSV2
  . quit:GX!EX
  . ;
  . set:GC'=" " $extract(GV2,L)=GC
  . quit
  quit
  ; ---------------------------------------------
  ; C - Current Mask Character from the FM
  ; CP - Character Position
  ; L - Position within
  ; VP - Value Position
  ; (integer - Right to Left, fraction - Left to Right)
```

```
TRANSV1  ; Translate the value into the mask
 set (GC,GL)=" "
 quit:"x"[C
 ;
 ; Value Completed, Apply Currency/Float/etc, if requested
 if 'VP do  quit
 . if "c"[C do
 . . set:$get(CP)="" CP=$length(CS)
 . . set GC=$extract(CS,CP),CP=CP-1
 . . set:CP<1 CP=$length(CS)
 . . quit
 . if GVM["f" do  quit
 . . new F,I,LI,LX,X,Q
 . . set X="" L1=L,LX=0
 . . do  ; Identify the Value Represented
 . . . if GVM["+"!(GVM["-"] do  quit
 . . . . set:GVM["+" X="+"
 . . . . set:V<0 X="-"
 . . . . quit
 . . . if GVM["(" do:V<0  quit
 . . . . set X="",LX=1
 . . . . quit
 . . . quit
 . . for I=L:1:GVL set Q=$extract(GV1,I) quit:Q>1N  quit:("("_DC)[Q  do
 . . . set F=$extract(GVM,1),L1=1
 . . . set:"fs("[F $extract(GV1,1)=x
 . . . quit
 . . set BYE=1
 . . set:LX $extract(GV1,LI)="("
 . . quit
 . quit
 if C="+" set GC="+" set:'SV GC="-" set GL=GC quit
 if C="-" set:'SV set GL=GC quit
 if C="(" set:'SV set GL=GC quit
 if C=")" set:'SV set GL=GC quit
 do:VP
 . if C="c" do
 . . set:$get(CP)="" CP=$length(CS)
 . . set GC=$extract(CS,CP),CP=CP-1
 . . set:SP<1 SP=$length(SC)
 . . quit
 . quit
```

```
  quit
  ; -------------------------------------------------
  ; "c"    - Currency
  ; "f"    - Floating
  ; "n"    - Numeric
  ; "s"    - Separator
  ; "+-()" - Sign Representations
  ; ------------------------------------------------
TRANSV2  ; Translate the value into the mask
  set GC=""
  quit:"x"[C
  ;
  do:VP
  . if "f"[C do  quit
  . . set:$get(CP)="" CP=1
  . . set GC=$extract(CS,CP),CP=CP+1
  . . set:CP>$length(CS) CP=1
  . . quit
  . if C="n" do  quit
  . . set GC=$extract(V2,VP),VP=VP+1
  . . set:VP>$length(V2) VP=0
  . . quit
  . if C="s" do  quit
  . . set GC=$extract(SC,SP),SP=SP+1
  . . set:SP<$length(SP) SP=1
  . . quit
  . quit
  if "c"[C do  quit
  . set:$get(CP)="" CP=1
  . set GC=$extract(CS,CP)
  . set:CP>$length(CS) GC=" "
  . set CP=CP+1
  . quit
  if C="+" set GC="+" set:'SV GC="-" set GL=GC quit
  if C="-" set:'SV GC="-" set GL=GC quit
  if C="(" set:'SV GC="(" set GL=GC quit
  if C=")" set:'SV GC=")" set GL=GC quit
  quit
  ; ---------------------------------
ERREX  ; Error Exit point
  do ERROR
  quit K
  ; ------------------------------------
  ; EC - Error Coded String (1 character or longer)
  ; EL - Error Code Length
  ; FL - Field Length
  ; K - Output String, The Error Message goes here.
ERROR  ; ERROR HANDLING
  new C,E,EL,L
  set:$get(FL)<1 FL=$$FLDLNG(0)
  set E=$get(EC),K="",L=1
  set:E="" E="*"
  set EL=$length(E)
  for I=1:1:FL set C=$extract(E,L),L=L+1 set:L>EL L=1 set K=K_C
  quit
  ; ---------------------------------------------
LOADTV  ; DO THE TRANSLATION OF THE TEMP. VALUE WITH THE STRING NX
  set X=""
  for M=1:2:$length(TV,"%%") do
  . set X=X_$piece(TV,"%%",M)
  . set N=$piece(TV,"%%",M+1)
  . set:N X=X_ST(N)
  . quit
  set TV=X
  quit
  ; ---------------------------------------------
FLDLNG(F)  ; FIELD LENGTH Callable from Just About Anywhere
  set F=$get(F)
  quit:F F
  ;
  set F=$length($get(FO("FM")))
  if 'F do
  . set F=$get(FO("FL"))
  . if 'F do
  . . set F=$length($get(^$format("FM")))
  . . if 'F do
  . . . set F=$length($get(^$system("FORMAT","FM")))
  . . . if 'F do
  . . . . set F=$length($get(^$system("FORMAT","FM")))
  . . . . if 'F set F=$get(^$system("FORMAT","FL"))
  . . . . quit
  . . . quit
  . . quit
  . quit
  set:'F F=10
  quit F
  ; --------------------------------------------
  ; vvvvvvvvvvvvvvvvvv
  ; Format Default Load
INFORM  ; Load up the defaults
  new K,X
  set K="",X="FORMAT"
  if '$data(^$format)
  . if '$data(^$system(X)) do QUIT
  . . set ^$format("SC")=" ",^$format("DC")="."
  . . set ^$format("CS")="$",^$format("EC")="*"
  . . quit
  . ; for  set K=$order(^$system(X,K)) quit:K=""  set
      ^$format(K)=^$system(X,K)
  . merge ^$format=^$system("FORMAT")
  . quit
  ; IF ^SYSTEM DOES NOT EXIST, CREATE IT
  do:'$data(^$system(X))
  . ; new K ; If you don't have the MERGE Command
  . ; set K=""
  . ; for  set K=$order(^$format(K)) quit:K=""  set ^$system(X,K)=^$format(K)
  . merge ^$system("FORMAT")=^$format
  . quit
  quit
  ; ---------------------------------------
```

3.5 $%LOWER^STRING

```
LOWER(A,CHARMOD)  ;
  new x,y set x=$get(CHARMOD) if x?1"^"1e.e do
  . set x=$extract(x,2,$length(x))
  . if x?1"|".e do
  . . set x=$reverse($extract(x,2,$length(x))
  . . set y=$reverse($piece(x,"|",2,999))
  . . set x=$reverse($piece(x,"|"))
  . set x=$get(^|y|$global(x,"CHARACTER"))
  . else  set x=$get($global(x,"CHARACTER"))
  if x="" s x=^$job($job,"CHARACTER")
  set x=$get(^$character(x,"LOWER"))
  if x="" quit $translate(A,"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      "abcdefghijklmnopqrstuvwxyz")
  set @("x="_x_"(A)")
  quit x
```

3.6 $%PRODUCE^STRING

```
PRODUCE(IN,SPEC,MAX)  ;
  new VALUE,AGAIN,PI,P2,I,COUNT
  set VALUE=IN,COUNT=0
  for  do  quit:'AGAIN
  .  set AGAIN=0
  .  set I=""
  .  for  set I=$order(SPEC(I)) quit:I=""  do  quit:COUNT<0
  .   . quit:$get(SPEC(I,1))=""
  .   . quit:'($data(SPEC(I,2))#2)
  .   . for  quit:VALUE'[SPEC(I,1)  do  quit:COUNT<0
  .   .   .  set P1=$piece(VALUE,SPEC(I,1),1)
  .   .   .  set P2=$piece(VALUE,SPEC(I,1),2,$length(VALUE))
  .   .   .  set VALUE=P1_SPEC(I,2)_P2,AGAIN=1
  .   .   .  set COUNT=COUNT+l
  .   .   .  if $data(MAX),COUNT>MAX set COUNT=-1,AGAIN=0
  .   .   .  quit
  .   .   quit
  .   quit
  quit VALUE
```

3.7 $%REPLACE^STRING

```
REPLACE(IN,SPEC)  ;
  new L,MASK,K,I,LT,F,VALUE
  set L=$length(IN),MASK=$justify("",L)
  set I="" for  set I=$order(SPEC(I)) quit:I=""  do
  . quit:'($data(SPEC(I,1))#2)
  . quit:SPEC(I,1)=""
  . quit:'($data(SPEC(I,2))#2)
  . set LT=$length(SPEC(I,1))
  . set F=0 for  set F=$find(IN,SPEC(I,1),F) quit:F<1 do
  .   . quit:$extract(MASK,F-LT,F-l)["X"
  .   . set VALUE(F-LT)=SPEC(I,2)
  .   . set $extract(MASK,F-LT,F-l)=$translate($justify("",LT)," ","X")
  .   . quit
  . quit
  set VALUE="" for K=1:1:L do
```

```
 . if $extract(MASK,K)=" " set VALUE=VALUE_$extract(IN,K) quit
 . set:$data(VALUE(K)) VALUE=VALUE_VALUE(K)
 . quit
 quit VALUE
```

3.8 $%UPPER^STRING

```
UPPER(A,CHARMOD)  ;
  new x,y set x=$get(CHARMOD)
  if x?1"^"1e.e do
  . set x=(x,2,$length(x))
  . if x?1"|".e do
  . . set x=$reverse($extract(x,2,$length(x)))
  . . set y=$reverse($piece(x,"|",2,999))
  . . set x=$reverse($piece(x,"|"))
  . . set x=$get(^|y|$global(x,"CHARACTER"))
  . else  set x=$get($global(x,"CHARACTER"))
  if x="" set x=^$job($job,"CHARACTER")
  set x=$get(^$character(x,"UPPER"))
  if x="" quit $translate(A,"abcdefghijklmnopqrstuvwxyz",
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
  set @("x="_x_"(A)")
  quit x
```

Index