Tcl/TOPE Reverse Engineering to MOIS

Version:	Draft
Date:	10-Sep-2003
Author:	Damien Callet
Reference:	ALC-MOIS-RQ-RHEA-0001
Filename:	Converter RequirementsCurrent.doc
Approved by:	Keith Turner Managing Director

Rhea System S.A.
New Tech Center,
Avenue Einstein 2a,
B-1348 Louvain-La-Neuve,
Belgium

TEL : + 32 10 48 72 50
FAX : + 32 10 45 25 07
www.rheagroup.com

DISTRIBUTION

Name	Number of Copies

DOCUMENT STATUS SHEET

Date	Version	Author	Reason for change
09/09/2003	Draft	Damien Callet	Draft

DOCUMENT CHANGE RECORD

Date	Version	Changed Pages/Paragraphs

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Purpose

This document defines the requirements to enable the conversion from a Tcl/TOPE script to a MOIS procedures.

It also provides a set of coding standards, based on the requirements, which can be used by authors of Tcl/TOPE procedures to facilitate conversion to MOIS.

1.2 Overview

The requirements here deal systematically with all of the TOPE extensions. Tcl control structures are also dealt with systematically. Other Tcl commands are dealt with on a case by case basis (according to their perceived utility in a control system). Tcl commands not dealt with here will not be supported.

This document consists of the following sections :-

- Language Analysis in the Context of MOIS – This section details the Tcl/Tope commands and how it should be implemented.
- Assumptions– This section details the assumptions made (generally about changes to be made to MOIS) that underpin the requirements definition.

1.3 Coding Standards – This section contains the coding standards for Tcl/TOPE, derived from the language analysis.Definitions

Tcl

An open source, interpreted scripting language. More information can be found at http://www.tcl.tk/.

Tcl/TOPE

An extension to Tcl for the Herschel / Planck ground control system. This allows flight procedures to be written in Tcl. The extensions include commands to examine telemetry and transmit telecommands.

1.4 Language Definitions

Two types of meta language have been used to describe the Tcl/TOPE. In all cases meta language is written in `courier` font in this document.

The first corresponds to the statement definitions taken from the Tcl package documentation and AD1. The formulation is not completely consistent and is generally used to introduce the Tcl/TOPE statement.

- Bold is used to identify keywords
- Variables may be in normal or italic, and may or may not be contained in angle brackets (< >).
- Question marks are used to bracket elements that are optional (e.g. if `<item>` is an optional argument, it is written `?<item>?`).
- An ellipsis (…) is used to denote iterated items (e.g. if `<item>` is a repeated argument, it is written `<item>` ... or perhaps `<item1>  <item2>` ...).

The second is (hopefully) used rather more rigourously, and appears mainly in the requirements definitions.

- Variables or expressions (or anything that is expanded in more details elsewhere) is contained in angle brackets (< >).
- Question marks (?) are used as a suffix to indicate the preceding item is optional (i.e. zero or one occurrences). E.g. if `<item>` is an optional argument, then it is written `<item>?`.
- Plus signs (+) are used as a suffix to indicate repetition one or more times. E.g. if `<item>` is a repeated argument that appears at least once, it is written `<item>+`
- Asterix signs (*) are used as a suffix to indicate repetition zero or more times. E.g. if `<item>` is an optional repeated argument , it is written `<item>*`
- Normal brackets ( ) are used to denote groups. E.g. if `<first_item>` and `<second_item>` are both optional arguments, but both are either present or absent together, it is written `(<first_item> <second_item>)?`
- The pipe sign (|) is used as a logical or operator. E.g. if `<item>` can be either `<alt_1>` or `<alt_2>`, it is written `<item> = <alt_1> | <alt_2>`
- Other characters stand for themselves. E.g. we could write a Tcl variable reference as `$<var-name>` (Tcl variable references are identified by the variable name prefixed with the dollar sign). Importantly for Tcl, braces { } and square brackets [ ] all appear for themselves.

1.5 Acronyms and Abbreviations

OL	Operations Language
MOIS	Mission Operation Information System

1.6 Applicable Documents

List all the documents with which this document must comply.

ID	Document	Reference
AD1	Herschel Planck Central Checkout System, System User Manual v 2.2	H-P-4-TE-MA-0010

1.7 Reference Documents

List all the documents referred to in this document, other than applicable documents.

ID	Document	Reference
RD1		

2 LANGUAGE ANALYSIS IN THE CONTEXT OF MOIS

2.1 Tcl Control Structures

This section describes if and how the Tcl control structures will be converted into MOIS structures. Each Tcl control structure (taken from the Tcl package documentation) is discussed in the subsections below.

2.1.1 After

```
after <ms>
after <ms> ?script script script ...?
after cancel <id>
after cancel script script script ...
after idle ?script script script ...
after info ?id?
```

Only the first formulation is handled by MOIS. This suspends execution for a number of milliseconds identified by the `<ms>` argument.

This statement will be implemented as a MOIS CTL/PSE statement with the wait period derived from the supplied `<ms>` argument.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter will recognise simple Tcl/TOPE statements of the form '`after <ms>`' described above and convert these to a MOIS CTL/PSE statement.

b. The converter will raise an error and fail the statement conversion if the `<time>` argument (interpreted as a literal) is not a positive integer.

2.1.2 Break

```
break
```

This statement is used to terminate loop execution early.

This statement will be implemented as a directive in MOIS procedure.

The requirements on the MOIS converter are as follows:-

a. The converter will recognose compound Tcl/TOPE statements of the form '`break`' and convert these as a MOIS directive statement.

2.1.3 Catch

```
catch <script> ?<varName>?
```

This statement is used to execute a script and catch any resultant errors. The `<script>` is executed as a Tcl/TOPE and the statement returns a boolean value identifying whether or not the `<script>` executed without errors.

`<varName>` is an optional variable name argument. If the `<script>` executes with errors, the corresponding variable is set to the error description. Otherwise the variable is set to the result of the `<script>` execution.

The statement is implemented as a MOIS function with two arguments :-

- `<script>` - String, contains the Tcl/TOPE to be executed. Mandatory.
- `<varName>` - String, contains a variable name. Optional.

The MOIS function will return a value of Boolean type

The requirements on the MOIS converter are as follows :-

a. The converter will recognise compound Tcl/TOPE statements of the form ‘ `[catch {<script>} <varName>?]`’ and convert these as a MOIS function call to the catch function.

b. The converter will create a MOIS local variable of Boolean type, assuming a variable of this name does not already exist.

c. The converter will raise an error and fail the statement conversion if a MOIS variable already exists with the same name and this is not of the Boolean type.

d. If `<varName>` is specified, the converter will create a MOIS local variable with the name `<varName>` of the String type, assuming a variable of this name does not already exist.

e. If `<varName>` is specified, the converter will raise an error and fail the statement conversion if a MOIS variable already exists with name `<varName>` and this is not of the String type.

2.1.4 Continue

`continue`

This statement is used to terminate the current loop iteration early, and move to the next.

This statement will be implemented as a MOIS directive statement.

The requirements on the MOIS converter are as follows:-

a. The converter will recognise compound Tcl/TOPE statements of the form ‘`continue`’ and convert it to the corresponding MOIS directive statement.

2.1.5 Error

`error` `message ?info? ?code?`

Statement Raises an error – particularly for propagating unhandled errors out of a catch statement script.

This statement will be implemented as a MOIS directive statement.

- `Message` – String containing the explanation of error, it’s a mandatory argument.
- `Info` – Variable containing usefull information of the error, optional.
- `Code` – Variable containing the error code happened, optional.

The requirements on the MOIS converter are as follows:-

The converter will recognise compound Tcl/TOPE statements of the form `'error'` and convert it to the corresponding MOIS directive statement.

The requirements on the MOIS convreter are as follows:-

a. The converter will recognise compound Tcl/TOPE statements of the form `'error message ?info? ?code?'` and convert it to the defined MOIS directive statement.

b. MOIS converter will raise an error and fail the conversion if `'message'` is not there.

2.1.6 Eval

`eval` `arg ?arg ...?`

Executes the arguments (arbitrary length Tcl script) as a script in a new instance of the Tcl interpreter and returns the result or the error code. .

This statement will be implemented as a MOIS function. This defined function should have one mandatory argument and several optional arguments.

The requirements on the MOIS converterare as follows :-

a. The converter will recognise compound Tcl/TOPE statements of the form `'eval {<script>}'`. This statement shall be converted as an FCT statement with the previously defined characteristics.

2.1.7 For

`for` `start test next body`

This is a Tcl 'for loop' structure. The `body` is repeatedly executed while the test evaluates to true. The `start` argument is Tcl code executed prior to the first iteration of the loop (usually initialising the loop control variable). The `next` argument is Tcl code executed at the end of each loop iteration (usually modifying the loop control variable).

This will be converted into a MOIS while loop. The `start` argument is inserted immediately prior to the start of the while loop, the `next` argument is appended to the body of the loop and the `test` argument is used as the while loop condition. The `body` argument is used as the while loop body.

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise compound Tcl/TOPE statements of the form `'for {<start>} {<test>} {<next>} {<body>}'`. Such statements shall be converted as a MOIS While structure.

b. The `<start>` argument shall be treated as Tcl/TOPE code and converted to MOIS procedure steps/statements. These steps/statements shall be inserted immediately prior to the While step.

c. The `<test>` argument shall be treated as a Tcl/TOPE condition and translated according to the requirements in section 2.2. The MOIS version of the condition shall be inserted as the condition for the While step.

d. The `<body>` argument shall be treated as Tcl/TOPE code and converted to MOIS procedure steps/statements. These steps/statements shall be inserted as subordinates to the MOIS loop body step, which is immediately after the While step.

e. The `<next>` argument shall be treated as Tcl/TOPE code and converted to MOIS procedure steps/statements. These steps/statements shall be appended as subordinates to the MOIS loop body step (i.e. these statements are executed in the loop body, but after the code corresponding to the `<body>`.

2.1.8 Foreach

```
foreach varname list body
foreach varlist1 list1 ?varlist2 list2 ...? body
```

The **foreach** command implements a loop where the loop variable(s) take on values from one or more lists. In the simplest case there is one loop variable, *varname*, and one list, *list*, that is a list of values to assign to *varname*. The *body* argument is a Tcl script. For each element of *list* (in order from first to last), **foreach** assigns the contents of the element to *varname* as if the **lindex** command had been used to extract the element, then calls the Tcl interpreter to execute *body*.

In the general case there can be more than one value list (e.g. *list1* and *list2*), and each value list can be associated with a list of loop variables (e.g. *varlist1* and *varlist2*). During each iteration of the loop the variables of each *varlist* are assigned consecutive values from the corresponding *list*. Values in each *list* are used in order from first to last, and each value is used exactly once. The total number of loop iterations is large enough to use up all the values from all the value lists. If a value list does not contain enough elements for each of its loop variables in each iteration, empty values are used for the missing elements.

This Tcl structure can't be mapped to any loop structure in MOIS.However it will be possible to implement it as a MOIS directive with three mandatory arguments.

- `<varname>` – variable into the foreach control structure. It will be a string argument into the directive
- `<varlist1>` – list of variables. It will be a string argument into the directive.
- `<list>` -list of parameter used into the `<foreach>` body.
- `<body>` - A string representing the list of Tcl/TOPE commands to execute by going through the list of parameter previously defined.

a. The requirement on the MOIS converter are as follows:-MOIS converter will recognise the two following formulation:

- `foreach <varname> {list} {body}`
- `foreach <varlist1> {list1} ?<varlist2> {list2 }...? {body}`

b. Converter will raise an error and the conversion will fail if the three mandatory paramters are not correct.

2.1.9 If

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ...
?else? ?bodyN?
```

This construct provides if-then-else functionality, it also effectively provides a switch functionality through the repeated elseif clauses.

MOIS will handle all variants of this structure. Simple if-then or if-then-else structures shall be converted to MOIS IF step. More complex forms, with any number of elseif clauses shall be converted to the MOIS switch statements.

Note that the handling of conditions is defined in this document – see section 2.2.

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise compound Tcl/TOPE statements of the form '`if {<condition>} (then)? {<then-clause>} ((else)? {<else-clause>})?`'. These shall be converted into MOIS if step structures.

b. An MOIS 'if' step shall be created. This shall contain one or more statements to define the condition specified in `<condition>`.

c. A MOIS step shall be inserted after the 'if' step and shall contain the steps / statements converted from the `<then-clause>`.

d. If the `<else-clause>` is specified, a further MOIS step shall be appended and shall contain the steps / statements converted from the `<else-clause>`.

e. The converter shall recognise compound Tcl/TOPE statements of the form '`if {<condition>} (then)? {<then-clause>} (elseif {<condition>} (then)? {<elseif-clause>})+ ((else)? {<else-clause>})?`'. These shall be converted into MOIS switch structures.

f. An MOIS 'switch' step shall be created. This shall contain several sub-steps (one for each then or elseif clause). Each sub-step shall contain one or more statements to define the condition specified in `<condition>` for the clause.

g. A MOIS step shall be appended after the 'switch' step corresponding to each then, elseif or else clause. Each step shall contain the steps / statements converted from the `<then-clause>`, `<elseif-clause>` or `<else-clause>` as appropriate.

2.1.10 Return

`return` `?`**`-code`** *`code`*`? ?`**`-errorinfo`** *`info`*`? ?`**`-errorcode`** *`code`*`? ?`*`string`*`?`

This statement returns control from a procedure to the calling code.

MOIS can't manage a return result of a called procedure. The only behaviour supported by MOIS procedure will be to exit the procedure on a return Tcl statement, which will be implemented as a directive. See section 2.5.11.

2.1.11 Switch

`switch` `?options? string pattern body ?pattern body ...?`

`switch` `?options? string {pattern body ?pattern body ...?}`

The **switch** command matches its *string* argument against each of the *pattern* arguments in order. As soon as it finds a *pattern* that matches *string* it evaluates the following *body* argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last *pattern* argument is **default** then it matches anything. If no *pattern* argument matches *string* and no default is given, then the **switch** command returns an empty string.

The *options* determine the type of pattern matching performed (exact, glob or regexp).

The practicality of implementing a conversion is somewhat dependent on the condition definitions. However, such a conversion would involve mapping the pattern comparisons back to complete conditions in the MOIS switch construct.

- This command is supported by MOIS and will be implemented as a MOIS switch step. Only the default pattern matching will be implemented by MOIS procedure(exact matching).`String` – It will a variable name of type string.
- `Pattern` – The value against which the string argument will be matched.
- `Body` – Part to evaluate if the corresponding pattern matches the string argument.

The requirements on MOIS converter are as follows:-

a. MOIS converter will recognise compound Tcl/TOPE statements of the form: **`'switch`** `?options? string {pattern body ?pattern body ...?}'`

b. For any value of options, the matche type will the default one ie the exact one.

c. If the compound structure is not as the one defined in section a. then the MOIS converter will raise an error and fail the conversion.

2.1.12 Update

`update` `?idletasks?`

This command is used to bring the application 'up to date' by entering the event loop repeatedly until all pending events (including idle callbacks) have been processed. If the **idletasks** keyword is specified as an argument to the command, then no new events or errors are processed; only idle callbacks are invoked. This causes operations that are normally deferred, such as display updates and window layout calculations, to be performed immediately.

This statement will be implemented as a MOIS directive with a single optional argument :-

`idletasks` – optional switch argument

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise simple Tcl/TOPE statements of the form '`update (idletasks)?`' and convert these as MOIS update directives.

2.1.13 Uplevel

`uplevel` `?level? arg ?arg ...?`

This statement allows a Tcl script to be executed within a procedure call scope above the current scope. E.g. if the variable p is defined in procedures x & y and the procedure x calls procedure y, then a call to uplevel in y would allow script to operate on the variable p in the scope of procedure x (instead of the scope of procedure y as would normally be the case).

MOIS converter won't implement this statement.

2.1.14 Vwait

`vwait` `varName`

This command enters the Tcl event loop to process events, blocking the application if no events are ready. It continues processing events until some event handler sets the value of variable *varName*. Once *varName* has been set, the **vwait** command will return as soon as the event handler that modified *varName* completes. *varName* must globally scoped (either with a call to global for the *varName*, or with the full namespace path specification).

This statement should be implemented as a MOIS directive with a single mandatory argument :-

- `varName` – optional switch argument

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise simple Tcl/TOPE statements of the form '`vwait <var-name>`' and convert these as MOIS vwait directives.

2.1.15 While

`while` `test body`

This statement defines a while loop structure. The Tcl script `body` is executed repeatedly until the Tcl script `test` evaluates to false.

The prototype converter converts these Tcl structures directly into the MOIS while construct.

Note that the handling of conditions is defined in section 2.2. The current prototype just converts the Tcl/TOPE condition script as a comment statement.

a. The requirements on the MOIS converter are as follows :-The converter shall recognise compound Tcl/TOPE statements of the form '`while {<condition>} {<body>}`'. These shall be converted into MOIS while structures.

b. A MOIS 'while' step shall be created. This shall contain one or more statements to define the condition specified in `<condition>`.

c. A MOIS step shall be inserted after the 'while' step and shall contain the steps / statements converted from the `<body>`.

2.2 Conditions

The current implementation of MOIS allows conditions to be expressed as logical combinations of VAR and TLM statement types (i.e. checks on the values of either MOIS variables or telemetry parameters). The syntax allows the use of AND and OR operators to combine the condition elements. The syntax also allows the use of brackets to control the evaluation order of the logical expressions.

For Tcl/TOPE, in fact, this formulation could generate arbitrary conditions by using **expr** statements to pre-evaluate complex conditions into Boolean variables which can then be used in simple MOIS condition definition. In order to effect conversion, however, the original Tcl/TOPE would have to be defined in the same way.

The discussion below therefore relates only to the simple VAR / TLM formulation, linked by AND and OR operators.

The definition of the conditions that can be translated is given below :-

`<condition> = \( <condition-element> ( <logical-operator> <condition-element> )* \)`

`<logical-operator> = && | \|\|` (i.e. either && or | |, the two Tcl logical operators)

`<condition-element> = <tm-check> | <var-check>`

`<tm-check> =` see section 2.3.1.1. for the definition of a TM check as used in a condition

`<var-check> = $<var-name> <comparison-operator> ( $<var-name> | <value> | <tm-value>)`

`<comparison-operator> =` a Tcl/TOPE comparison operator equivalent to the comparison operators available to the MOIS VAR statement `( == | != | > | < | >= | <= )`

`<tm-value> = [getengvalue [fetch <tm_name>]]`

The MOIS requirements are as follows :-

a. The MOIS converter shall identify and convert conditions defined according to the definition of `<condition>` above.

b. Each <condition-element> shall be translated as a separate MOIS statement within the same step.

c. The created MOIS statements shall appear in the same order as they appear in the <condition>.

d. In cases where the <condition-element> is associated with a <logical-operator>, the step, in which statements has just been defined, will define the condition in the step expression builder.

e. Each <var-check> shall be translated as a MOIS VAR statement.

f. Each <tm-check> shall be translated as a MOIS TLM statement.

2.3 Parameter Access

MOIS allows verification of TM parameter references against the s/c database, so it is important to ensure that all TM references made within Tcl/TOPE can be translated into the relevant MOIS structures. Any TM references that are not converted will be missed from the automatic validation provided by MOIS.

The MOIS TLM statements, to which the Tcl/TOPE parameter references will be converted, can be used either in a condition (e.g. for an if or while stucture) or as part of a step. In the latter case, the MOIS TLM statement represents a telemetry check with no explicit action as a result of check pass or failure (although the action might be described in a comment).

Tcl/TOPE includes several parameter access structures which will not fit into the MOIS TLM statement model, but for which access via MOIS will be permitted. These are implemented as MOIS directives & functions (as described in section 2.4.2).

2.3.1 Conversion to MOIS TLM Statements

The following subsections deal with the conversion of parameter access statements from Tcl/TOPE to MOIS. These are the preferred structures for dealing with parameter access statements as they effect the generation of the TLM statements in MOIS.

2.3.1.1 As Used in Conditions

When used as part of a condition, the Tcl/TOPE must match the following pattern, which must in turn be recognised by the MOIS converter.

```
{[getengvalue [fetch <tm_name>]] <operator> <eng_value>}
```

where :-

- `<tm_name>` is a literal string corresponding to a TM parameter name
- `<operator>` is a Tcl/TOPE comparison operator equivalent to the comparison operators available to the MOIS TLM statement (== | != | > | < | >= | <=)
- `<eng_value>` is a literal string or numeric value with which the TM parameter is compared

As it was defined in section 2.2, Several coditions can be combined in a same step with the expression builder, which allows the handling of AND and OR operators and the handling of brackets.

The MOIS converter must match patterns of this form, creating a TLM statement corresponding to each and relating to other TLM (and other) statements according to the condition context.

Mois converter requirements :-

a. The converter shall recognise compound Tcl/TOPE statements in the form described above, within the context of a condition definition, and convert these to a MOIS TLM statement within an equivalent condition context in MOIS.

b. The converter shall raise an error and fail the statement conversion if the TM parameter `<tm_name>` is not found in the s/c DB.

c. The converter shall raise an error and fail the conversion if the <eng_value> is inconsistent with the TM parameter as defined in the s/c DB (numeric value out of range, numeric value outside calibration range, text value not matrching a defined alias).

2.3.1.2 As Used Outside Conditions

The converter should be able to recognize the following formulation and then convert it in MOIS procedure as a PERFORM step containing one verify TM statement.

```
if {[getengvalue [fetch <tm_name>]] <operator>
<eng_value>}{ … exit}
```

where :-

- `<tm_name>` is a literal string corresponding to a TM parameter name
- `<operator>` is a Tcl/TOPE comparison operator equivalent to the comparison operators available to the MOIS TLM statement (== | != | > | < | >= | <=)
- `<eng_value>` is a literal string or numeric value with which the TM parameter is compared

Mois converter requirements :-

a. The converter shall recognise compound Tcl/TOPE statements in the form described above and convert these to a MOIS TLM statement.

b. The converter shall raise an error and fail the statement conversion if the TM parameter `<tm_name>` is not found in the s/c DB.

c. The converter shall raise an error and fail the conversion if the <eng_value> is inconsistent with the TM parameter as defined in the s/c DB (numeric value out of range, numeric value outside calibration range, text value not matrching a defined alias).

2.3.2 Conversion To Directive / Function Structures

The detailed requirements here are described in section 2.4, but these introduce some constraints that may not be immediately obvious. These are described here.

2.3.2.1 TM Parameter Access

The parameter access statements in Tcl/TOPE are detailed in sections 2.4.1 to 2.4.5. In all cases the MOIS converter will only handle parameter access conforming to the following pattern.

Tcl/TOPE allows the way to set up the TM parameter attributes using either a `fecth` or `subscribe` whereas MOIS allows to set up the value of TM parameter in a variable.

Then MOIS will recognise Tcl statement corresponding to the patterns described in the relevant section i.e.:-

- `set <var> [getrawvalue[fetch <param-name>]]`
- `set <var> [getengvalue[fetch <param-name>]]`

MOIS will implement it as a MOIS SET Var statement telemetry.

If the TM parameter attributes use `subscribe` then MOIS will implement it as a directive see section 2.4.3.

The `<var>` must be a variable of type depending to the return type of functions called.

The `<param-name>` must be a string literal representing the TM parameter name. Only literals are allowed here so that the TM parameter can be verified against the s/c DB.

The subscribe function will be recognized by converter and will be implemented as a directive – see section 2.4.3.

The `unsubscribe` function can be called at any time (using the literal TM parameter name as argument) to cancel a subscription and will be implemented as a directive.

The key points here are as follows :-

- Obtaining useful access to a TM parameter attribute requires three Tcl/TOPE statements (a `set / fetch` followed by a `set / getPARAMETERdata`)
- This rather rigid structure is required to facilitate the recognition and conversion of the Tcl/TOPE into MOIS.
- The MOIS converter can only handle conversion of simple statements (i.e. those without embedded command substitutions) plus compound statements that conform to specific structures defined in this document.

2.3.2.2 TM Packet Access

The packet access statements in Tcl/TOPE are detailed in sections 2.4.6 to 2.4.8. In all cases the MOIS converter will only handle packet access conforming to the following pattern.

Set up a variable containing the TM packet attributes using a `subscribepacket`. The variable name must be new (not used before in the procedure) or an old one that was originally set by a call to `subscribepacket`.MOIS converter will implement a MOIS wait for packet as soon as the following relevant pattern is detected:-

- `subscribepacket <spid> referby <varname>`

  `waitfor ?-timeout <time>? <varname>`

  `unsubscribepacket <spid>`

The `<spid>` must be a literal representing the TM packet id. Only literals are allowed here so that the TM packet id can be verified against the s/c DB.

Note that once a variable is set up with the TM packet attribute list, it can be used as an argument in a call to `waitfor` as in the previous pattern. Otherwise it can be used as an argument to one of the `getPACKETdata` functions to extract one of the packet attributes and set it to the variable, i.e. :-

- `set <var_1> [<getPACKETdata> <var>]`

Once the final variable `<var_1>` is set up, it can be used in any operation appropriate to its type (corresponds to the return type of the `getPACKETdata` function.

The `unsubscribepacket` function can be called at any time (using the literal TM packet id as argument) to cancel a subscription.

The key points here are as follows :-

- This rather rigid structure is required to facilitate the recognition and conversion of the Tcl/TOPE into MOIS.

- The packet access traces back to the literal packet id as the argument to the `subscribepacket`. Using a literal here ensures that MOIS can validate the TM packet id used against the s/c DB.
- The MOIS converter can only handle conversion of simple statements (i.e. those without embedded command substitutions) plus compound statements that conform to specific structures defined in this document.

2.4 TOPE Statements

This section addresses each of the commands in the TOPE extensions to Tcl. All of the commands are discussed in turn, with a brief description of the command. The implementation of the command in MOIS is detailed along with the requirements on the MOIS converter to achieve the conversion. Any constraints on the conversion or implementation of the command are also covered.

2.4.1 Fetch

```
Fetch <param-name>
```

This statement returns a list of the TM parameter attributes (including current value), which can then be accessed by the GetPARAMETERdata convenience functions.

Fetch command use will depend of the way it will be used. Examples of the use of Fetch command are described in the relevant section i.e.:-

- TCL Code:'`set real1000 [getrawvalue [fetch SSC01000]]`' will be implemented to a MOIS Set Var Statement telemetry having a radix raw.
- TCL code:'`while { [getengvalue [fetch YZS17900]] == "CONNECTED" }`' will be implemented to a MOIS WHILE step with a condition being a MOIS verify TLM statement having an alias radix.

To summarize the fetch command will be implemented in MOIS procedure, either as a MOIS SET variable statement telemtry or as a MOIS verify TLM statement.The requirements on the MOIS converter are as follows:

a. The converter shall recognise Tcl/TOPE compound statements of the form

   Set <var-name> [<fn> [fetch <param-name>] or shall recognize (IF|WHILE) {[<fn> [fetch <param-name>] == <value>}. <fn> should be one of functions defined in next part.

b. The converter shall create a MOIS local variable in case of Set statement with the name <var-name> depending of the return type of <fn> function used.

c. It will raise an error and fail the conversion if MOIS local variable already exists and this is not the appropriate type.

2.4.2 GetPARAMETERdata

This consists of a set of functions for accessing the specific attributes from a list of all the variable attributes (returned from the Fetch command).

- `getname <parval>`
- `getrawvalue <parval>`
- `israwvaluevalid <parval>`
- `getrawvalidity <parval>`
- `getengvalue <parval>`
- `isengvaluevalid <parval>`
- `getengvalidity <parval>`
- `getdefaultvalue <parval>`
- `isdefaultvaluevalid <parval>`
- `getdefaultvalidity <parval>`
- `getextractedvalue <parval>`
- `getsccstate <parval>`
- `getoolstate <parval>`
- `gettimestamp <parval>`

- `<parval>` -It representing the return value of a fecth command.

- `getname` –Since TM name is passed as paramater to fetch TM characteristic, getname is obvious. Return value will be a string.
- `getrawvalue` – As it is defined above it will depend of the context in which it will be used.
- `israwvaluevalid` –The converter shall implement it as a function, which will return a boolean
- `getrawvalidity` – The converter shall implement it as a function, which will return an integer.

`getengvalue` – It will be implemented as getrawvalue, as described above.

- `isengvaluevalid` – The converter will implement it as a function returning a boolean.
- `getengvalidity` – The converter will implement it as a function returnong an integer.
- `getdefaultvalue` – It will be implemented as a function, return value needs to be dfined.

- `isdefaultvaluevalid` –The converter will implement it as a function returning a boolean.
- `getdefaultvalidity` – The converter will implement it as a function returnong an integer.
- getextractedvalue – The converter shall implement it as a function, the return vaue needs to be defined.
- getsccstate – The converter shall implement it as a function, which will return one of the four following values: "SCC_INIT", "SCC_UNINT", "SCC_DISABLE" or "SCC_OFF".
- `getoolstate` – The converter shall implement it as a function returning a string ('NOMINAL', 'WARNING' or 'ALARM').
- `gettimestamp` – The converter shall implement it as a function returning a string.

Each function describes above will be implemented as a MOIS FCT statement except `getrawvalue` and `getengvalue`. These function will take two parameters:

- `fetch` will be a delimeter .
- `<param-name>` will be the name of parameter of type string.

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise compound Tcl/TOPE statements '`[<fn> fetch <param-name>]`' (with `<fn>` corresponding to each of the GetPARAMETERdata statements, which shall be implemented as function) and convert these as MOIS FCT statement. Then a MOIS local variable will be created to get the return value of functions.

b. The converter shall recognise compound Tcl/TOPE statements of the form '`set <var-name> [<fn> fetch <param-name>]`' and convert this as MOIS SET variable statement of set type telemetry. It should be happened when `<fn>` is 'getrawvalue' or 'getengvalue'.

d. The `converter shall recognise compound Tcl/TOPE statements of the form '`<control>([<fn> fetch <param-name>]== <value>`' and convert this as MOIS VERIFY Telemetry statement. `<control>` shall be a step of type decision(IF) or loop(WHILE/REPEAT). It will happenif `<fn>` is 'getrawvalue' or 'getengvalue'.

e. If the converter needs to create a local variable, it shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the appropriate type.

2.4.3 Subscribe

```
subscribe <param-name> referby <var>
```

This statement associates a TM parameter with a variable name similarly to the Fetch statement. However, in this case the variable is updated with the current list of TM parameter attributes each time the TM parameter changes. It can be used in conjunction with the waitfor statement to perform some task whenever the TM value changes.

This statement should be implemented as a MOIS directive with three mandatory arguments:

- `<param-name>` - MOIS can supply value as a TM name.
- `referby` – fixed parameter
- `<var>` - string argument representing a variable name

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise simple Tcl/TOPE statements of the form '`subscribe <param-name> referby <var>`' and convert these as MOIS subscribe directives.

b. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument is not a valid TM name defined in the s/c DB.

2.4.4 Subscribeset

```
subscribeset <param-list> referby <var>
```

This statement is similar to the Subscribe statement. However, in this case a set of TM parameters is associated with a root variable name. In fact a set of variables with names in the form '`<var>_<param>`' are created.

This statement will be implemented by MOIS procedure as a directive . The directive will have the following parameters:

- `<param-list>` String containing the list of parameters.
- `Referby <var>` String being the name of variable with the referby delimiter.

The requirement of MOIS converter are as follows:-

a. The converter shall recognise simple Tcl/TOPE statements of the form '`subscribeset <param-list> referby <var>`' and convert these as MOIS subscribeset directives.

b. The converter shall raise an error and fail the statement conversion if the `<param-list>` argument is not a correct.

2.4.5 Unsubscribe

```
unsubscribe <param-name>

unsubscribe <param-list>

unsubscribe -all
```

This statement cancels a previous subscription to one or more TM parameters.

In this statement, the TM parameter name(s) used as the argument in the first two cases should be validated by MOIS. Due to the complications of handling a list of parameters in the second case, this second form of the statement is not supported by MOIS.The different forms of the statement mean that it must be implemented as a three MOIS directives. The first will be called '`unsubscribe`' with the mandatory argument :-

- `<param-name>` - MOIS can supply value as a TM name.

The second will be called '`unsubscribe`' with the mandatory argument:-

- `<param-list>` – It sill be a string containing the list of parameters.

The third will be called '`unsubscribe -all`' and won't have arguments.

The requirements on the MOIS converter are as follows :-

a. The unsubscribe statements operating on a single parameter shall be implemented as a MOIS directive.

b. The unsubscribe statement operating on all parameters shall be implemented as a separate MOIS directive.

c. The converter shall raise an error and fail the statement conversion if the `<param-name>` or `<param-list>` argument is not a valid TM name defined in the s/c DB.

2.4.6 getPACKETdata

This consists of a set of functions for accessing the attributes of a variable representing a TM packet (returned as a result of calls to the Subscribepacket statement).

- `getrawdata <pktval>`
- `getpusapid <pktval>`
- `getpustype <pktval>`
- `getpussubtype <pktval>`
- `getfilingtime <pktval>`
- `getsrcseqcnt <pktval>`

These functions can't be implemented in MOIS procedure because MOIS can't manage properly the argument `<pktval>`,which should be packet attributes.

2.4.7 Subscribepacket

```
subscribepacket <spid> referby <var>
```

This statement associates a TM packet with a variable name, the variable is updated with the current list of packet attributes each time the packet changes. It can be used in conjunction with the waitfor statement to perform some task whenever the TM packet changes.

This statement should be implemented as a MOIS directive with three mandatory arguments

- `<spid>` - packet id type, such that MOIS can validate the supplied value as a packet name.

- `referby` – fixed parameter
- `<var>` - string argument representing a variable name

The requirements on the MOIS converter are as follows :-

a. The subscribe statement shall be converted as a MOIS directive.

b. The `<var>` argument represents a variable name. The converter shall create an appropriate MOIS local variable corresponding to this argument if it does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS local variable corresponding to the `<var>` argument already exists, but is of an inappropriate type (must be a string).

The converter shall raise an error and fail the statement conversion if the `<spid>` argument is not a valid packet id defined in the s/c DB.

2.4.8 Unsubscribepacket

`unsubscribepacket <spid>`

This statement cancels a previous subscription to a single TM packet.

This statement should be implemented as a MOIS directive with a single mandatory argument :-

- `<spid>` - packet id type, such that MOIS can validate the supplied value as a packet name.

The requirements on the MOIS converter are as follows :-

a. The unsubscribepacket statement shall be implemented as a MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<spid>` argument is not a valid packet id defined in the s/c DB.

2.4.9 Nametospid

`Nametospid <pktName>`

This statement convert packet name from name in the SDB to SCOS2000 packet ID (SPID)

This statement will be implemented as a MOIS function with a single mandatory argument:

- `<pktName>` - packet name, type will be a string. If this packet name doesn't exist then an empty string will be returned.

2.4.10 Spidtoname

`Spidtoname <spid>`

Convert SCOS2000 packet ID (SPID) to packet name stored in the SDB

This statement will be implemented as a MOIS function with a single mandatory argument:

- `<spid>` - packet id, type will be an integer. If this packet id doesn't exist then an empty string will be returned.

2.4.11 Tcsend

```
tcsend <command-name> ?referby <var>? ?<Options>...?
?<Parameters...>?
```

This statement is used to send telecommands to the s/c. The statement requires the `<command-name>` parameter, in order for MOIS to validate this against the available commands, this must be specified as a literal.

The optional `referby <var>` clause is used to associate the command with a variable in order to trace the command progress (PTV, CEV, etc). The variable values should be accessed using the getTC_STATUSdata convenience functions.

The optional `<Parameters>` list defines the command parameter values to be used for this instance of the command. Each parameter is defined by a Tcl list (contained in braces) in the following format.

```
{ <name> <value> ?<format>? ?RAW|ENG|DEFAULT? }
```

Where :-

- `<name>` - Parameter name
- `<value>` - Parameter value.
- `<format>` - Format of the value:
  - `SH` - short
  - `LO` - long
  - `US` - unsigned short
  - `UL` - unsigned long
  - `FL` - float
  - `DO` - double
  - `CH` - char
  - `BO` - Boolean
  - `OC` - octet
  - `ST` – string (default if format is not specified)
  - `BS` - binary string
  - `TI` - time
- `RAW|ENG|DEFAULT`
  - `RAW` - specifies that this is a raw (uncalibrated) value (default if no option is specified).
  - `ENG` - specifies that this is an engineering (calibrated) value.
  - `DEFAULT` - specifies to use the default value from the SDB

Note that :-

- If the `DEFAULT` keyword is used, an empty string should be used for `<value>` and `<format>`. Any specified values will be replaced by the s/c DB defaults for the command parameter.

- Tcl/TOPE allows a file to be referenced for a parameter value by prefixing the <value> with a "@" character (file path follows the "@"). This formulation is not supported by MOIS.
- Multiple parameters are specified by adding further parameter definitions, separated by spaces, e.g. `tcsend X123 {P001 1} {P002 2}` defines the command X123 to be sent with parameters P001 set to 1 and P002 set to 2.

The main complication of the statement lies in the formatting of the optional `<options>`. In general, each of these options consists of a keyword followed by a single argument or an argument list (contained in braces). The keywords, their arguments and a discussion of the function in each case is given below.

- **`releasetime`** `<time>`

  The absolute time when the command shall be released. `<time>` must be an absolute or (positive) relative time in the TOPE/SCOS2000 time format. If unspecified, the default value is ASAP.

- **`executiontime`** `<time>`

  The absolute time when the command shall be executed (i.e. the execution timetag). `<time>` must be an absolute time in the TOPE/SCOS2000 time format. If unspecified, the default value is ASAP (i.e. the command should be executed directly by the spacecraft).

- **`checks`** `{<staticPTVflag> <dynamicPTVflag> <CEVflag>}` or
- **`checks`** `<checkFlag>`

  Defines which of the dynamic & static PTV (Pre Transmission Verification) and CEV (Command Execution Verification) checks should be performed for the current command. There are two formulations, one allowing individual control of the check flags and the other switching all the check flags on or off together. The allowed flag values are given below :-

  - `staticPTVflag = SPTV | SPTV_OFF`
  - `dynamicPTVflag = DPTV | DPTV_OFF`
  - `CEVflag = CEV | CEV_OFF`
  - `checkFlag = ALL | NONE`

  If unspecified, the default value is `ALL`.

- **`ack`** `<ackflags>`

  Specifies the CEV reporting to be applied for the command. <ackflags> consists of a list of one or literals (if more than one, they appear in braces in a space separated list as per normal Tcl rules). The presence of each literal indicates that the corresponding CEV reporting should be performed, absence indicates the reporting should not be performed. The literals are :-

  - ACCEPT
  - START
  - PROGRESS

  - COMPLETE
  - ALL – indicates all the reporting should be performed
  - NONE – indicates no reporting should be performed

  If unspecified, defaults are taken from the s/c DB (presumably from the command definition – TBC).

- **`id`** `<id>`

  This option is used to specify a TC ID (or "Observation ID"). This TC ID will be included in the command history archive of the CCS. The value is an arbitrary positive integer number of default value 0 (this value appears in the archive if the option is not used).

- **`NOCRC`**

  When specified, suppresses the CRC checksum calculation by the CCS.

- **`patch { {`** `<offset> <mask> <value>` **`}.. }`**

  Specifies a list of patches to be applied to the encoded TC packet. This is not dealt with in detail here, as it can't be supported by MOIS. It also seems unlikely that it would be used in flight.

This statement will be implemented as a MOIS CMD statement.

In all cases, the tcsend arguments must be literal values (i.e. not results of command substitutions or variable values) in order to be translated and verified by MOIS.

a. The converter shall attempt to convert all Tcl/TOPE `tcsend` statements to MOIS CMD statements.

b. The converter shall raise an error and fail the statement conversion if the `<command-name>` (when translated as a literal value) does not correspond to a TC in the s/c DB.

c. The converter shall identify the presence of the referby keyword in the statement and identfy the subsequent referby parameter name (as a literal). The parameter shall be created as a MOIS local parameter, type of it needs to be confirmed.

d. The converter shall raise an error and fail the statement conversion if the referby parameter already exists as a MOIS local parameter and is not of the type it should be.

e. The converter shall associate the created MOIS CMD statement with the identified referby parameter (assuming an error was not generated above).

f. The converter shall identify each TC parameter defined in the statement.

g. For each parameter, the converter shall raise an error and fail the statement conversion if the parameter name specified (as a literal) does not match a parameter for the TC defined in the s/c DB.

h. For each parameter, the converter shall raise an error and fail the statement conversion if the specified parameter attributes (value & format) are inconsistent with the parameter definitions for the TC defined in the s/c DB. Inconsistencies include (but are not restricted to) the following :- numeric value out of range, alias value not defined in DB, types not convertible. This does not apply if the parameter is set to DEFAULT (i.e. use s/c DB defaults) – in this case we don't care what the parameter attributes are.

i. For each DEFAULT parameter, the converter shall build the MOIS CMD statement using the s/c DB defaults for the current parameter.

j. For each non DEFAULT parameter, the converter shall build the MOIS CMD statement using the supplied parameter value and attributes.

k. For TC parameters which are required (according to the s/c DB), but for which no parameter data is supplied, the converter shall build the MOIS CMD statement using the s/c DB defaults for the current parameter. All TC parameters should be included in the CMD statement.

l. The converter shall identify occurrences of each of the `tcsend` options in the statement along with any option arguments.

m. The converter shall raise an error and fail the statement conversion if any option has arguments that do not match the expected pattern (defined above).

n. The converter shall validate & copy the `<time>` argument value for the **`releasetime`** option to the command_Uplink_Time. NB – the current implementation restricts this to relative times only (Time tag tab of TC form).

o. The converter shall validate & copy the `<time>` argument value for the **`executiontime`** option to the command_Execution_Time(Time tag tab of TC form).

p. The converter shall set the PTV and CEV flags according to the **`checks`** option arguments described above if the **`checks`** option is defined. Note that dynamicPTV are not supported by MOIS procedure.

q. The converter shall not support the **`ack`** option (see above) if this option is present.

r. The converter shall not support the observation id if this option is present.

s. The converter shall not support the suppress CRC flag if the **`NOCRC`** option is present in the statement.

t. The converter shall NOT support the **`patch`** option. If found, an error shall be raised and the statement conversion failed.

u. The converter shall raise an error and fail the statement conversion if the statement as a whole contains elements which are inconsistent with the `tcsend` statement pattern. E.g. unhandled or unknown options

2.4.12 getTC_STATUSdata

This consists of a set of functions for accessing the attributes of a variable representing a telecommand (returned as a result of calls to the referby option of the Tcsend statement).

- `getrequestid <vval>`
- `getstage <vval>`
- `getstatus <vval>`
- `getstagehistory <vval>`
- `getcompleted <vval>`
- `getupdatetime <vval>`

All these statement will be implemented as MOIS functions with a single mandatory argument :-

`<vval>` - It will be the refer by variable name used in tcsend command.

The return type is dependent on the individual function, these are listed below. Note that some of these are TBC pending some investigation into the exact nature of the return value.

- `getrequestid` – integer, the request id (not the observation id).
- `getstage` – integer, representing the current stage of TC transmission (value represented by a mnemonic)
- `getstatus` – integer, representing the status of the current stage of TC transmission (value represented by a mnemonic)
- `getstagehistory` – string. The string contains a single letter corresponding to each transmission stage representing the status of that stage.
- `getcompleted` – boolean, identifies whether the TC is complete (i.e. no further verification reports expected) or not.
- `getupdatetime` – string, timestamp corresponding to the current verification report.

The requirements on the MOIS converter are as follows :-

a. The converter shall recognise compound Tcl/TOPE compound statements of the form '`set <var-name> [<fn> <vval>]`' (with `<fn>` corresponding to each of the GetTC_STATUSdata statements) and convert these as a MOIS function calls to the appropriate function.

b. The converter shall create a MOIS local variable with the name `<var-name>` of the type appropriate to the return type of the function, assuming a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the appropriate type.

2.4.13 attach

`attach <name>`

Attach to a SCOE or DFE named <*name*>. This is a preliminary for sending commands which are bound for this SCOE/DFE. The sequence remains attached until detach is called or until the test sequence terminates.

This statement should be implemented as a MOIS directive with a single mandatory argument :-

- `<name>` - SCOE/DFE/IS name type, such that MOIS can validate the supplied value as a SCOE/DFE/IS name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) does not correspond to a SCOE/DFE/IS name in the s/c DB.

2.4.14 detach

`detach <name>`

Dettach from a SCOE or DFE named <*name*>.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<name>` - SCOE/DFE/IS name type, such that MOIS can validate the supplied value as a SCOE/DFE/IS name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) does not correspond to a SCOE/DFE/IS name in the s/c DB.

2.4.15 authorise

`authorise ?-revoke? <name>`

Authorise the current sequence for the command `<name>`. The next `tcsend <name>` will be authorised, even if other requests are submitted between `authorise` and `tcsend`. The option `-revoke` causes any previous authorisation of `<name>` to be revoked.

This statement will be implemented as a MOIS directive with two arguments :-

- `-revoke` - Switch argument with a single allowed value ('-revoke'). Optional.
- `<name>` - TC name type, such that MOIS can validate the supplied value as a TC name. Mandatory.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) does not correspond to a SCOE/DFE/IS name in the s/c DB.

2.4.16 connect

`connect <name>`

Instructs the CCS to establish a connection to the SCOE/DFE/IS called <*name*>. Note: These names are configured in the spacecraft database.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<name>` - SCOE/DFE/IS name type, such that MOIS can validate the supplied value as a SCOE/DFE/IS name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) does not correspond to a SCOE/DFE/IS name in the s/c DB.

2.4.17 disconnect

`disconnect <name>`

Instructs the CCS to terminate a connection to the SCOE/DFE/IS called <*name*>.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<name>` - SCOE/DFE/IS name type, such that MOIS can validate the supplied value as a SCOE/DFE/IS name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) does not correspond to a SCOE/DFE/IS name in the s/c DB.

2.4.18 newtmdumpfile

`newtmdumpfile <vcid> ?<dumpname>?`

The `newtmdumpfile` command reopens the dump for Virtual Channel `<vcid>` on a new file, which will contain `<vcid>` and `<dumpname>` in its name. A previously opened dump will be closed automatically by CCS.

This statement will be implemented as a MOIS directive with two arguments :-

- `<vcid>` - integer (>= 0) corresponding virtual channel id for the s/c. Mandatory.

- `<dumpname>` - string (alphanumeric and underscore characters only) corresponding to the dump file name. Optional.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<vcid>` argument (interpreted as a literal) is not an integer >= 0.

c. The converter shall raise an error and fail the statement conversion if the `<dumpname>` argument (interpreted as a literal), if present, is not a string of alphanumeric and underscore characters only.

2.4.19 setparameter

```
setparameter ?-raw? <param-name> <value>
```

Sets a user defined TM parameter (user defined constant) in the control system. The TM parameter `<param-name>` is set to `<value>`. The allowable range of values depends on the s/c DB definition of the engineering value of the parameter.

If option `-raw` is specified, `<value>` is a raw (uncalibrated) value. In this case only unsigned integer values are allowed.

This statement will be implemented as a MOIS directive with three arguments :-

- `-raw` - Switch argument with a single allowed value ('-raw'). Optional.
- `<param-name>` - TM type such that MOIS can validate the supplied value as a TM name. NB the type here should really be further restricted to the user defined TM parameters.
- `<value>` - string which converts to the appropriate engineering type for the TM parameter (or an unsigned integer only if `-raw` is specified). Mandatory.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument (interpreted as a literal) is not a user defined TM parameter as defined in the s/c DB.

c. The converter shall raise an error and fail the statement conversion if the `<value>` argument (interpreted as a literal) does not convert to a valid, in range value for the TM parameter (as defined in the s/c DB) in cases where the `-raw` switch is not specified. Where the `-raw` switch is specified, the error should be raised if the `<value>` is not an unsigned integer or if the value is too large to fit in the parameter.

2.4.20 enableparam

```
enableparam <param-list>
```

Enables the processing of the specified parameter(s) by the ground control system.

This statement will be implemented as a MOIS directive with one mandatory argument :-

- `<param-list>` - A string representing the list of parameters, it will be delimited by braces.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form `enableparam {<param-list>}` and convert these to the corresponding MOIS directive.

2.4.21 inhibitparam

`inhibitparam <param-list>`

Inhibits the processing of the specified parameter(s) by the ground control system.

This statement will be implemented as a MOIS directive with one mandatory argument :-

- `<param-list>` - A string representing the list of parameters, it will be delimited by braces.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form `inhibitparam {<param-list>}` and convert these to the corresponding MOIS directive.

2.4.22 enablepacket

`enablepacket <spid>`

Enable the processing of packet identified by `<spid>`.

This statement should be implemented as a MOIS directive with a single mandatory argument :-

- `<spid>` - packet id type, such that MOIS can validate the supplied value as a packet name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<spid>` argument (interpreted as a literal) is not a valid packet id defined in the s/c DB.

2.4.23 inhibitpacket

`inhibitpacket <spid>`

Inhibit the processing of packet identified by `<spid>`.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<spid>` - packet id type, such that MOIS can validate the supplied value as a packet name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<spid>` argument (interpreted as a literal) is not a valid packet id defined in the s/c DB.

2.4.24 enablegroup

`enablegroup` *`<grpid>`*

Enable the processing of a group of packets or parameters. (Groups of packets or parameters can be defined in the spacecraft database). The group ID `<grpid>` is expressed as a string.This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<grpid>` - group id type, such that MOIS can validate the supplied value as a group name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<grpid>` argument (interpreted as a literal) is not a valid group name defined in the s/c DB.

2.4.25 inhibitgroup

`inhibitgroup` *`<grpid>`*

Inhibit the processing of a group of packets or parameters.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<grpid>` - group id type, such that MOIS can validate the supplied value as a group name.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<grpid>` argument (interpreted as a literal) is not a valid group name defined in the s/c DB.

2.4.26 patchlocation

```
patchlocation <param-name> <spid> <byteoffset> <bitoffset>
```

This statement is used to temporarily modify the extraction of a TM parameter. The packet and location within the packet are both specified.

This statement will be implemented as a MOIS directive with four mandatory arguments :-

- `<param-name>` - TM type such that MOIS can validate the supplied value as a TM name.
- `<spid>` - packet id type, such that MOIS can validate the supplied value as a packet name.
- `<byteoffset>` - integer (>= 0) representing the byte offset of the parameter location
- `<bitoffset>` - integer (in range 0 -> 7) representing the bit offset of the parameter location

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert it to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<spid>` argument (interpreted as a literal) is not a valid packet id defined in the s/c DB.

c. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument (interpreted as a literal) is not a TM parameter defined in the s/c DB.

d. The converter shall raise an error and fail the statement conversion if the `<byteoffset>` argument (interpreted as a literal) is not a positive integer.

e. The converter shall raise an error and fail the statement conversion if the `<bitoffset>` argument (interpreted as a literal) is not an integer in the range 0 -> 7.

2.4.27 patchscript

```
patchscript <param-name> <script>
```

This command is used to modify the expression used to derive the value a synthetic (derived) parameter.

This statement will be implemented as a MOIS directive with two mandatory arguments :-

- `<param-name>` - TM type such that MOIS can validate the supplied value as a TM name (derived parameters only).
- `<script>` - string (arbitrary length), contains the Tcl/TOPE used to derive the parameter value.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument (interpreted as a literal) is not a valid synthetic TM parameter id defined in the s/c DB.

Note that the converter (or MOIS) will not make any attempt to translate or validate the Tcl/TOPE contained in the `<script>` argument.

2.4.28 patchnumericalcurve

```
patchnumericalcurve <calibcurveId> <pointId> <newXval>
<newYval>
```

Statement is used to modify a single point in a linear calibration curve.

This statement will be implemented as a MOIS directive with four mandatory arguments :-

- `<calibcurveId>` - Positive integer, represents a valid calibration curve id (from the CAP_NUMBR field) in the s/c DB.
- `<pointId>` - Positive integer, identifying the point to be changed
- `<newXval>` - Unsigned integer representing the new raw value for the point
- `<newYval>` - Real representing the engineering value for the point

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<calibcurveId>` argument (interpreted as a literal) is not calibration curve id defined in the s/c DB (CAP_NUMBR field).

c. The converter shall raise an error and fail the statement conversion if the values for the other arguments (interpreted as literals) do not correspond with the defined types.

2.4.29 patchtextualcurve

```
patchtextualcurve <calibcurveId> <pointId> <newFrom>
<newTo> <newText>
```

Statement is used to modify a single point in a text calibration.This statement will be implemented as a MOIS directive with five mandatory arguments :-

- `<calibcurveId>` - Positive integer, represents a valid calibration curve id (from the TXP_NUMBR field) in the s/c DB.
- `<pointId>` - Positive integer, identifying the point to be changed
- `<newFrom>` - Unsigned integer representing the low raw value for the range (TBC)
- `<newTo>` - Unsigned integer representing the high raw value for the range (TBC)
- `<newText>` - String representing the text alias for the defined range

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<calibcurveId>` argument (interpreted as a literal) is not calibration curve id defined in the s/c DB (CAP_NUMBR field).

c. The converter shall raise an error and fail the statement conversion if the values for the other arguments (interpreted as literals) do not correspond with the defined types.

2.4.30 patchpolynomialcurve

```
patchpolynomialcurve <calibcurveId> <coefficientId>
<newCoeffValue>
```

Statement is used to modify a single point in a polynomial curve calibration.

This statement will be implemented as a MOIS directive with three mandatory arguments :-

- `<calibcurveId>` - Positive integer, represents a valid calibration curve id (from the TXP_NUMBR field) in the s/c DB.
- `<coefficientId>` - Integer (in range 0 → 4), identifying the polynomial coefficient to be changed.
- `<newCoeffValue>` - Real representing the new coefficient value (TBC)

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<calibcurveId>` argument (interpreted as a literal) is not calibration curve id defined in the s/c DB (MCF_IDENT field).

c. The converter shall raise an error and fail the statement conversion if the values for the other arguments (interpreted as literals) do not correspond with the defined types.

2.4.31 patchcurveused

```
patchcurveused <param-name> <newCurveId>
```

```
patchcurveused <param-name> <pos> <newCurveId>
```

These commands patch the curve used by the referenced parameter for calibration.

In the first variant, the default calibration curve is switched to `<newCurveId>` for TM. In the second variant, the conditional calibration curve at position `<pos>` is switched. In both cases, `<newCurveId>` must refer to a curve of the same type (numerical, textual, polynomial) as the one originally defined in the spacecraft database.

- The formulation here is a bit of a problem, as we have two variants which are only identifiable by the different argument list. Unfortunately, the `<pos>` argument is not at the end (if it was we could just make it optional).

So only the second formulation will be implemented in a MOIS procedures

This statement will be implemented as a MOIS directive with three mandatory arguments :-

- `<param-name>` - TM type such that MOIS can validate the supplied value as a TM name (derived parameters only).

- `<pos>` - Positive integer, represents the `<pos>`. Mandatory.
- `<newCurveId>` - Positive integer, represents the `<newCurveId>`

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument (interpreted as a literal) is not a TM parameter name defined in the s/c DB.

c. The converter shall raise an error and fail the statement conversion if the values for the other arguments (interpreted as literals) do not correspond with the defined types.

2.4.32 patchlimit

```
patchlimit <param-name> <type> <pos> <lowValue>
?<highValue>?
```

This statement patches the limit currently applicable to the specified parameter in the ground control system.

This statement will be implemented as a MOIS directive with five arguments :-

- `<param-name>` - TM type such that MOIS can validate the supplied value as a TM name (derived parameters only).
- `<type>` - Character (‘H’, ‘S’ or ‘D’) representing the limit type (hard, soft or delta). Mandatory.
- `<pos>` - Positive integer, represents the entry in the OCP table to be modified. Mandatory.
- `<lowValue>` - Unsigned integer (TBC) representing the minimum in limit value (or the delta value if this is a delta limit). Mandatory.
- `<highValue>` - Unsigned integer (TBC) representing the maximum in limit value. Not specified if this is a delta limit. Optional.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form described above and convert these to the corresponding MOIS directive.

b. The converter shall raise an error and fail the statement conversion if the `<param-name>` argument (interpreted as a literal) is not a TM parameter id defined in the s/c DB.

c. The converter shall raise an error and fail the statement conversion if the values for the other arguments (interpreted as literals) do not correspond with the defined types.

2.4.33 waittime

```
waittime <time>
```

Suspends execution for a period identified by the `<time>` argument.

This statement will be implemented as a MOIS CTL/PSE statement with the wait period derived from the supplied `<time>` argument (this is specified as a TOPE delta time).

The requirements on the MOIS converter are as follows :-

c. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`waittime` *`<time>`*' described above and convert these to a MOIS CTL/PSE statement.

d. The converter shall raise an error and fail the statement conversion if the `<time>` argument (interpreted as a literal) is not a positive delta time in the TOPE time format.

2.4.34 call

```
call <name> ?arg...?
```

Invokes (synchronously) the test sequence named `<name>` with an optional list of arguments. The sequence is expected to reside in the file `<name>.tcl` in the test sequence source directory. The optional arguments represent the procedure arguments required for the called procedure.

This statement will be implemented as a MOIS FOP statement with the called procedure name set to `<name>`. The synchronous option will be set in MOIS FOP statement.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`call` *`<name> ?arg…?`*' and convert these to a MOIS FOP statement.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) is not a valid procedure name.

c. The converter shall raise an error and fail the statement conversion if procedure arguments are not consistent with procedure arguments in configuration control..

2.4.35 callasync

```
callasync ?-referby var? name ?arg...?
```

Invokes (asynchronously) the test sequence named `<name>` with an optional list of arguments. The sequence is expected to reside in the file `<name>.tcl` in the test sequence source directory. The optional arguments represent the procedure arguments required for the called procedure.

This statement will be implemented as a MOIS FOP statement with the called procedure name set to `<name>`. The asynchronous option will be set in MOIS FOP statement.

NB. The referby option is also an issue, it will not supported by MOIS procedure.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`callasync ?-referby var?` *`<name>`*' and convert these to a MOIS FOP statement.

b. The converter shall raise an error and fail the statement conversion if the `<name>` argument (interpreted as a literal) is not a valid procedure name.

c. If the –referby switch is present, the converter shall raise an error and fail the conversion if no corresponding variable name is specified.

d. If the referby variable is present, the converter will not take it into account.

2.4.36 waitfor

```
waitfor ?-timeout <time>? <var-list> ?<condition>?
waitfor ?-timeout <time>? -all <var-list>
```

In the first formulation, the statement waits for the update of any of the variables given in `<var-list>` (variables used in the `referby` clause of a TM subscription, TC send request, or callasync request). The switch '*`-timeout <time>`*' specifies a maximum amount of time to wait. `<time>` must be a positive delta time.

The optional argument *`<condition>`* is a Tcl code fragment which will be evaluated whenever any of the variables in *`<var-list>`* is updated. If *`<condition>`* evaluates to true, waiting is finished; otherwise, `waitfor` continues to wait for the next update or the maximum time.

In the second formulation, the command statement waits for *all* of the specified variables to be updated at *least once* before returning.

The statement returns 0 (logical false) if the specified timeout has expired or 1 (logical true) otherwise.

The way that this command will be implemented in a MOIS procedure will depend of the way this command will be called. It will be explained in coding standards part::-

- `-timeout` – Optional switch defining whether a timeout should be applied.
- `<time>` - Positive delta time corresponding to the timeout period. The argument is associated with the `-timeout` switch, and is therefore mandatory if the `–timeout` switch is present and must not be included if the `–timeout` switch is not present.
- `<var-list>` - Variable name. Mandatory. List of variables won't be supported by MOIS.
- `<condition>` - String representing a Tcl/TOPE condition definition. Optional.

For the functions, the return value is boolean.

NB – MOIS will perform no validation on the Tcl/TOPE contained in the `<condition>` argument.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`waitfor ?-timeout <time>? <var-list> ?<condition>?`'

and convert these to a MOIS directive statement depending of the way it will be used.

b. The converter shall raise an error and terminate the statement conversion if the `<var-list>` argument (interpreted as a literal) does not correspond to a MOIS local variable.

.

c. The converter shall raise an error and fail the statement conversion if the `<time>` argument (interpreted as a literal) is not a valid positive delta time.

d. The converter shall raise an error if the `<var-list>` is not only one varaiable.

2.4.37 getshared

```
getshared <name>
```

Returns the current value of the shared variable named < *name*>.

In MOIS, shared variables are handled as global variables, which could in principle be used anytime a local variable is used. This creates complications with the Tcl/TOPE construct, as it is not practical to allow substitution of '`[getshared <name>]`' in place of any variable usage.

The solution is to allow the copy of the global variable to a local variable, using the MOIS set statement.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <var-name> [getshared <name>]`' and convert these to a MOIS SET statement (setting local variable `<var-name>` to global variable `<name>`).

b. The converter shall create a MOIS local variable with the name `<var-name>` of the type String, assuming a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the type String.

d. The converter shall create a MOIS global variable with the name `<name>` of the type String, assuming a variable of this name does not already exist.

e. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<name>` and this is not of the type String.

2.4.38 setshared

```
setshared <name> <value>
```

Sets the current value of the shared variable <name> to <value>.

The statement will be implemented as a MOIS SET statement.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`setshared <name> <value>`’ and convert these to a MOIS SET statement (setting global variable `<name>` to literal value `<value>`).

b. The converter shall create a MOIS global variable with the name `<name>` of the type String, assuming a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS global variable already exists with name `<name>` and this is not of the type String.

2.4.39 lockshared

`lockshared ?-timeout <time>? <name>`

Locks the shared variable `<name>`, which prevents other test sequences from setting it. The option -timeout allows to specify a timeout value as a positive delta time. If this option is not specified, `lockshared` will not wait if it cannot obtain the lock. The lock is maintained until `unlockshared` is called or the sequence holding the lock terminates.

The statement returns a Boolean value indicating whether or not the lock was obtained.This statement will be implemented as both a MOIS directive and a function, each with three arguments :-

- `-timeout` – Optional switch defining whether a timeout should be applied
- `<time>` - Positive delta time corresponding to the timeout period. The argument is associated with the `-timeout` switch, and is therefore mandatory if the `-timeout` switch is present and must not be included if the `-timeout` switch is not present. TBC.
- `<name>` - String representing a variable name. Mandatory.

For the function, the return value is boolean.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`lockshared ?-timeout <time>? <name>`’ and convert these to a MOIS directive statement.

b. The MOIS converter shall recognise compound Tcl/TOPE statements of the form ‘`set <return> [lockshared ?-timeout <time>? <name>]`’ and convert these to a MOIS function statement.

c. The converter shall create a MOIS global variable (of type String) corresponding to the `<name>` argument (interpreted as a literal) if a variable of this name does not already exist.

d. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<name>` and this is not of the type String.

e. The converter shall raise an error and fail the statement conversion if the `<time>` argument (interpreted as a literal) is not a valid positive delta time.

f. In the case of a function, the converter shall create a local MOIS variable (Boolean) corresponding to the `<return>` argument (interpreted as a literal), provided a variable of the same name does not already exist.

g. In the case of a function, the converter shall raise an error and fail the statement conversion if a MOIS variable corresponding to `<return>` already exists and is not of Boolean type.

2.4.40 unlockshared

```
unlockshared <name>
```

Unlocks the shared variable `<name>`.This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<name>` - String representing a variable name

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`unlockshared <name>`’ and convert these to a MOIS directive statement.

b. The converter shall create a MOIS global variable (of type String) corresponding to the `<name>` argument (interpreted as a literal) if a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<name>` and this is not of the type String.

2.4.41 Tellsequence

```
Tellsequence <id> <action>
```

The execution state is an asynchronous(detached) child sequence can be changed using the tellsequence command.

This statement will be implemented as a MOIS directive statement with two single mandatory arguments:

`<id>` identifier of the target sequence as returned by the callasync command. It will be implemented as the name of the sequence in MOIS procedure TBD.

`<action>` must be one of the following keywords “SUSPEND”, “RESUME” and “TERMINATE”.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`Tellsequence <id> <action>`’ and convert them to a MOIS directive statement.

b. The converter shall create a MOIS global variable with the name `<name>` of the type String, assuming a variable of this name does not already exist.

c. The converter shall raise an error and fail if the two mandatory parameters are not defined.

d. The converter shall raise an error and fail if the two mandatory parameters are not defined.

e. The converter shall raise an error and fail if the `<id>` is not defined child sequence.

2.4.42 Suspend

```
Suspend
```

It interrupts the execution of the current sequence. If it receives a resume notification, it will continue. While a sequence is suspended, it will ignore incoming telemetry data.

This statement will be implemented as a MOIS directive statement.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`Suspen`’ and convert it to a MOIS directive statement.

2.4.43 putlog

```
putlog ?-error? ?-warn? ?--? <expr>
```

This statement writes the log message `<expr>` to the logbook. The log entry is tagged as an error or warning if the appropriate switch is set.

The statement will be implemented as a MOIS directive with two arguments :-

- Switch – statement switch taking values ‘-error’, ‘-warn’ or ‘—‘. Optional.
- `<expr>` - quoted string containing the logging message text.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`putlog ?(-error | -warn | --)? <expr>`’ and convert these to a MOIS directive statement.

2.4.44 syslog

```
syslog ?-error? ?-warn? ?--? <expr>
```

This statement writes the log message `<expr>` to the system event log. The log entry is tagged as an error or warning if the appropriate switch is set.

The statement should be implemented as a MOIS directive with two arguments :-

- Switch – statement switch taking values ‘-error’, ‘-warn’ or ‘—‘. Optional.
- `<expr>` - quoted string containing the logging message text.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`syslog ?(-error | -warn | --)? <expr>`’ and convert these to a MOIS directive statement.

2.4.45 setrevision

```
setrevision <expr>
```

This statement stores the string `<expr>` as the revision information for the sequence.

The statement should be implemented as a MOIS directive with a single mandatory argument:-

- `<expr>` - quoted string containing the revision information

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form ‘`setrevision <expr>`’ and convert these to a MOIS directive statement.

NB – in MOIS Writer, use of this directive should ideally provide a mechanism for setting the current version of the MOIS procedure as the value of `<expr>`.

2.4.46 verified

```
verified ?-timestamp <time>? -tc <cmd> ?<param>...?
```

```
verified ?-timestamp <time>? -tm <parameter>
```

The `verified` command is a convenience function for adding records to the report file. The revision information stored by the `setrevision` command is included in the report file. If the `setrevision` command has been omitted, the revision of the current test sequence will be "unspecified".

Each command formulation will be implemented as a directive.

- `-timestamp` – Optional switch defining a time, which will be written in the report fil.
- `<time>` – The absolute time value to write in the report file.
- `-tc` – Switch defining a telecommand verified.
- `<cmd>` – Name of the command incase of telecommand verified.
- `<param>` – Optional argument listing the name of telecommand parameters.
- `-tm` – Switch defining a telemetry verified.
- `<parameter>` – Name of the telemetry to verified.

The requirements on the MOIS converter are as follows:-

a. MOIS converter will recognise the two following formulations Tcl/TOPE command:-

   - `verified (?-timestamp <time>?|--) -tc <cmd> ?<param>...?`
   - `verified ?-timestamp <time>? -tm <parameter>`

   Then it will convert it to a MOIS directive statement.

b. If <time> is defined and its format is not correct, the converter will raise an error and the Tcl/TOPE command conversion will fail.

c. The converter will raise an error if mandatory parameters are not defined and the Tcl/TOPE command conversion will fail.

2.4.47 prompt

```
prompt ?<type>? <message>
```

This statement displays a dialogue to the user of the control system. The dialogue displays the text in `<message>`, and requests the user to provide an input, which is then returned. The dialogue buttons and the type of the user input / return value depend on `<type>`.

```
<type> = signed | unsigned | float | bool | abstime |
reltime
```

Some of this functionality is represented in MOIS by the SET statement with the user input option set. In cases where the `<type>` maps directly to a MOIS variable type, the statement will be implemented as a SET statement. Cases where there is no corresponding MOIS type anyway can't be implemented. The title of the SET statement will be set to the text in the `<message>` argument.

The case in which no `<type>` is defined will be implemented as a MOIS directive. The return value is irrelevant in this case, the dialogue simply delays procedure execution until the dialogue 'Ok' button is hit. The directive shall have a single mandatory argument :-

- `<message>` - String representing the message to display.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <return> [prompt <type> <message>]`' where the <type> corresponds to a MOIS variable type. Such statements shall be translated into MOIS SET statements with the user input option set.

b. The converter shall create a MOIS local variable of a type corresponding to `<type>` and with name corresponding to the `<return>` argument (interpreted as a literal) if a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return>` and this is not of the appropriate type.

d. The converter shall set the SET statement title to the text contained in the `<message>` argument.

e. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`prompt <message>`'. Such statements shall be translated into MOIS directive statements.

2.4.48 displaystatus

```
displaystatus <message>
```

This statement displays a status message on the control system.

This statement will be implemented as a MOIS directive with a single mandatory argument :-

- `<message>` - String containing the status message.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl/TOPE statements of the form '`displaystatus <message>`' and convert these to the corresponding MOIS directive.

2.4.49 bintohex

`bintohex <string>`

Convert `<string>` to a readable (hexadecimal) dump of the binary data. `<string>` is an arbitrary binary Tcl string. It may contain null and all kinds of special characters.

This statement will be implemented as a MOIS function with a single mandatory argument and returning a string value :-

- `<string>` - String containing the binary data.

In general the handling of Tcl/TOPE statement arguments has been restricted to literals. However, in this case the statement is only likely to be applied to values returned from other statements (no one is going to create a literal binary string). In this case the argument is assumed to be a variable of type string.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <return> [bintohex $<var>]`' and convert these to the corresponding MOIS function.

b. The MOIS function shall be created such that `<string>` argument is assigned the value of the variable `<var>`.

c. The converter shall raise an error and fail the statement conversion if `<var>` is not a MOIS local variable of type String.

d. The converter shall create a MOIS local variable of string type and with name corresponding to the `<return>` argument (interpreted as a literal) if a variable of this name does not already exist.

e. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return>` and this is not of the string type.

2.4.50 hextobin

`hextobin <hex-string>`

Convert `<hex-string>` to a binary Tcl string.

This statement will be implemented as a MOIS function with a single mandatory argument and returning a string value :-

- `<string>` - String containing the hex data.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <return> [hextobin <hex-string>]`' and convert these to the corresponding MOIS function.

b. The converter shall create a MOIS local variable of string type and with name corresponding to the `<return>` argument (interpreted as a literal) if a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return>` and this is not of the string type.

2.4.51 asdtomsec

`asdtomsec <delta-time-string>`

Convert `<delta-time-string>` to an integer number of milliseconds.

This statement will be implemented as a MOIS function with a single mandatory argument and returning an integer value :-

- `<delta-time-string>` - String containing the delta time.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <return> [asdtomsec <delta-time-string>]`' and convert these to the corresponding MOIS function.

b. The converter shall create a MOIS local variable of integer type and with name corresponding to the `<return>` argument (interpreted as a literal) if a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return>` and this is not of the integer type.

2.4.52 Asdtosec

`Asdtosec <time-string> ?<secs-var> <usecs_var>?`

Convert `<delta-time-string>` to integer number of seconds and microseconds from EPOCH.

This statement will be implemented as a MOIS function with a single mandatory argument and returning an integer value :-

- `< time-string >` - String containing the delta time.

The requirements on the MOIS converter are as follows :-

d. The MOIS converter shall recognise compound Tcl/TOPE statements of the form '`set <return> [asdtomsec <time-string>]`' and convert these to the corresponding MOIS function.

e. The converter shall create a MOIS local variable of integer type and with name corresponding to the `<return>` argument (interpreted as a literal) if a variable of this name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return>` and this is not of the integer type.

2.5 Tcl Statements

This section details the converter requirements relating to basic Tcl statements not covered in section 2.1, above. It is assumed that only the Tcl statements in section 8.2 of AD1 need to be addressed here. Other Tcl statements are not implemented by the MOIS converter.

2.5.1 append - Append to variable

append `varName ?value value value ...?`

Append all of the *value* arguments to the current value of variable *varName*. If *varName* doesn't exist, it is given a value equal to the concatenation of all the *value* arguments.

This command provides an efficient way to build up long variables incrementally. For example, ``**append a $**b'' is much more efficient than ``**set a a**b'' if **$a** is long.

This statement will be implemented as a MOIS function, returning a string value. The function has a single repeated argument :-

- `<value>` - String containing the data to be appended to the `<varName>` variable.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`append <varName> <value>+`' and convert these to the MOIS append function.

b. The converter shall create a MOIS local variable of string type and with name corresponding to the `<varName>` argument (interpreted as a literal) if a variable of this name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<varName>` and this is not of the string type.

2.5.2 array - Manipulate array variables

`array` `option arrayName ?arg arg ...?`

This command consists of a range of alternate structures to support array handling in Tcl. MOIS does not support array types or array handling, so none of the array variants are supported for the conversion.

2.5.3 bgerror - Command invoked to process background errors

`bgerror` `message`

The **bgerror** command doesn't exist as built-in part of Tcl. Instead, individual applications or users can define a **bgerror** command (e.g. as a Tcl procedure) if they wish to handle background errors. A background error is one that occurs in an event handler or some other command that didn't originate with the application. For example, if an error occurs while executing a command specified with the **after** command, then it is a background error. For a non-background error, the error can simply be returned up through nested Tcl command evaluations until it reaches the top-level code in the application; then the application can report the error in whatever way it wishes. When a background error occurs, the unwinding ends in the Tcl library and there is no obvious way for Tcl to report the error.

When Tcl detects a background error, it saves information about the error and invokes the **bgerror** command later as an idle event handler. Before invoking **bgerror**, Tcl restores the **errorInfo** and **errorCode** variables to their values at the time the error occurred, then it invokes **bgerror** with the error message as its only argument. Tcl assumes that the application has implemented the **bgerror** command, and that the command will report the error in a way that makes sense for the application.

The bgerror command may or may not be implemented in Tcl/TOPE, but certainly this command is not one for use in operational procedures. It will be not implemented in MOIS or the MOIS converter.

2.5.4 binary - Insert and extract fields from binary strings

`binary format` `formatString ?arg arg ...?`

binary scan `string formatString ?varName varName ...?`

This command provides facilities for manipulating binary data. The first form, **binary format**, creates a binary string from normal Tcl values. For example, given the values 16 and 22, on a 32 bit architecture, it might produce an 8-byte binary string consisting of two 4-byte integers, one for each of the numbers. The second form of the command, **binary scan**, does the opposite: it extracts data from a binary string and returns it as ordinary Tcl string values.

The binary format command will be implemented as a MOIS function, returning a string representing the binary number as formatted. The command takes the following arguments :-

- `formatString` – String representing the formatting to be applied to the other arguments. Mandatory.
- `arg` – Repeated string argument defining the values to be formatted.

The binary scan command will be implemented as a MOIS directive, with arguments as follows :-

- `string` – String containing the binary representation to be converted. Mandatory.
- `formatString` – String containing the formatting instructions for the binary string. Mandatory.
- `varName` – Repeated argument representing variable names into which the converted binary values are placed according to the formatting instructions. Note that these variables could be numeric or string types, but there is no way to know without parsing the format string. MOIS must treat them all as strings.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <binary-string> [binary format <format-string> <arg>+]`’ and convert these to the ‘binary format’ MOIS function.

b. For the `<arg>` arguments, the converter shall recognise either single words (that is undelimited strings with no spaces) or lists (strings containing spaces delimited by braces {}).

c. The converter shall create a MOIS local variable of String type with the name corresponding to `<binary-string>`, provided a variable of that name does not already exist.

d. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<binary-string>` and this is not of the string type.

e. The MOIS converter shall recognise simple Tcl statements of the form ‘`binary scan <binary> <format-string> <var-name>+`’ and convert these to the ‘binary scan’ MOIS directive.

f. The converter shall create a MOIS local variable of String type with the name corresponding to `<var-name>`, provided a variable of that name does not already exist for each instance of the `<var-name>` argument.

g. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the string type for any instance of the `<var-name>` argument.

2.5.5 cd - Change working directory

`cd` `?dirName?`

Change the current working directory to *dirNam*e, or to the home directory (as specified in the HOME environment variable) if *dirName* is not given. Returns an empty string.The cd command will be implemented as a MOIS directive with the following argument :-

- `dirName` – Optional string argument representing the new directory path.

The requirerments on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`cd <dir-name>?`' and convert these to the 'cd' MOIS directive.

2.5.6 clock - Obtain and manipulate time

`clock clicks` `?-milliseconds?`

`clock format` `*clockValue* ?-format *string*? ?-gmt *boolean*?`

`clock scan` `*dateString* ?-base *clockVal*? ?-gmt *boolean*?`

`clock seconds`

This is a set of commands for manipulating and obtaining the system time. The different forms are outlined in the bullets below :-

- Clicks – returns an integer representing a high resolution time corresponding to the number of clock ticks since some arbitrary time. The returned value is only suitable for measuring elapsed time.
- Format – converts an integer time to a formatted date and / or time string.
- Scan – converts a date / time string into a an integer time.
- Seconds – returns an integer time, representing the number of seconds elapsed since a baseline epoch.

The `clock clicks` form will be implemented as a MOIS function with the following arguments and returning an integer value :-

- `-milliseconds` – Optional switch argument with a single allowed value "-milliseconds".

The `clock format` form will be implemented as a MOIS function with the following arguments and returning a string containing the formatted time.

- `clockValue` – Integer value representing a clock time (as returned from `clock seconds`). Mandatory.
- `-format` – Optional switch argument with a single allowed value of "-format".
- `<format-string>` - String containing formatting instructions for the time value. It is associated with the `-format` switch and is present if and only if the `-format` switch is present.

- `-gmt` – Optional switch argument with a single allowed value of "-gmt".
- `<gmt-switch>` - Boolean value identifying whether the time value should be defined as a GMT or local time. It is associated with the `-gmt` switch and is present if and only if the `-gmt` switch is present.

The `clock scan` form will be implemented as a MOIS function with the following arguments and returning an integer value representing the system time.

- `<date-string>` - String containing the date and/or time formatted for display. Mandatory.
- `-base` – Optional switch argument with a single allowed value of "-base".
- `<clock-value>` - Integer representing a clock time to specify the date relating to this command. It is associated with the `-base` switch and is present if and only if the `-base` switch is present.
- `-gmt` – Optional switch argument with a single allowed value of "-gmt".
- `<gmt-switch>` - Boolean value identifying whether the time value should be defined as a GMT or local time. It is associated with the `-gmt` switch and is present if and only if the `-gmt` switch is present.

The `clock seconds` form will be implemented as a MOIS function with no arguments and returning an Integer value representing the current time.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <time-millisecs> [clock clicks (-millseconds)?]`' and convert these to the 'clock clicks' MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<time-millisecs>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<time-millisecs>` and this is not of the Integer type.

d. The MOIS converter shall recognise compound Tcl statements of the form '`set <time-string> [clock format <clock-value> (-format <format-string>)? (-gmt <gmt-switch>)?]`' and convert these to the 'clock format' MOIS function.

e. The converter shall create a MOIS local variable of String type with the name corresponding to `<time-string>`, provided a variable of that name does not already exist.

f. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<time-string>` and this is not of the String type.

g. The MOIS converter shall recognise compound Tcl statements of the form '`set <time-value> [clock scan <time-string> (-base`

<base-value>)? (-gmt <gmt-switch>)?]' and convert these to the 'clock scan' MOIS function.

h. The converter shall create a MOIS local variable of Integer type with the name corresponding to <time-value>, provided a variable of that name does not already exist.

i. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name <time-value> and this is not of the Integer type.

j. The MOIS converter shall recognise compound Tcl statements of the form 'set <time-value> [clock seconds]' and convert these to the 'clock seconds' MOIS function.

k. The converter shall create a MOIS local variable of Integer type with the name corresponding to <time-value>, provided a variable of that name does not already exist.

l. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name <time-value> and this is not of the Integer type.

2.5.7 close - Close an open channel.

close channelId

Closes the channel given by *channelI*d. *ChannelId* must be a channel identifier such as the return value from a previous **open** or **socket** command. All buffered output is flushed to the channel's output device, any buffered input is discarded, the underlying file or device is closed, and *channelId* becomes unavailable for use.

This command will be implemented as a MOIS directive, with a single mandatory argument :-

- channelId – String identifying the channel to be closed.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form 'close <channel-id>' and convert these to the 'close' MOIS directive.

2.5.8 concat - Join lists together

concat ?arg arg ...?

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in the *ar*g's and adds a single separator space between *ar*g's. It permits any number of arguments. For example, the command 'concat a b {c d e} {f {g h}}' will return 'a b c d e f {g h}' as its result. If no *ar*gs are supplied, the result is an empty string.

This command will be implemented as a MOIS function with a single repeated argument and returns a string value containing the concatenated list.

- <arg> - Repeated string argument containing a list item

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <list-string> [concat <list-item>+]`’ and convert these to the ‘concat’ MOIS function.

b. For the `<list-item>` arguments, the converter shall recognise either single words (that is undelimited strings with no spaces) or lists (strings containing spaces delimited by braces { }).

c. The converter shall create a MOIS local variable of String type with the name corresponding to `<list-string>`, provided a variable of that name does not already exist.

d. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<list-string>` and this is not of the string type.

2.5.9 eof - Check for end of file condition on channel

`eof` `channelId`

Returns 1 if an end of file condition occurred during the most recent input operation on *channelId* (such as **get**s), 0 otherwise.This command will be implemented as a MOIS function, with a single mandatory argument and returning a boolean value :-

- `channelId` – String identifying the channel to be checked.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <eof-result> [eof <channel-id>]`’ and convert these to the ‘eof’ MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<eof-result>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<eof-result>` and this is not of the Boolean type.

2.5.10 exec - Invoke subprocess(es)

This command is used to trigger execution of subprocesses.This Tcl command will not be implemented in MOIS procedures.

2.5.11 exit - End the application

`exit` `?returnCode?`

Terminate the process, returning *returnCode* to the system as the exit status. If *returnCode* isn't specified then it defaults to 0.

This command will be implemented as a MOIS CTL/XIT statement, but with no support for the return code. The MOIS converter requirements are as follows :-

The MOIS converter shall recognise simple Tcl statements of the form ‘`exit`’ and convert these to the MOIS CTL/XIT statement.

2.5.12 expr - Evaluate an expression

expr arg ?arg arg ...?

Concatenates *arg*'s (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression ‘`expr 8.2 + 6`’ evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

For the MOIS conversion it would not seem to be practical to implement this command with the repreated arguments, as each argument can be a string, number operator, bracket, variable reference, etc. A simpler solution would be to treat this as a MOIS function with a single string argument (so that everything after the `expr` keyword to the statement terminator is assumed to be part of the expression). This approach means that virtually no validation can be performed on the expression elements, but clearly this command must be implemented somehow. The biggest hole here will be that variable references will be able to be made without any type or existance validation

Another issue is the type of the return value. We can’t identify the return type from the expression, so we have no choice but to use a string type for the result. This is likely to cause problems when using the return value as arguments to other functions / directives and in conditions. Probably the best solution is to define several MOIS functions, all mapping to the `expr` command, but returning different MOIS types. Procedures created using MOIS Writer can then select the variant appropriate to the return type expected from the expression used. Converted procedures would have to use the string variant and then manually edit to the appropriate variant as required.

This command will be implemented as a set of MOIS functions, each with a single mandatory string argument. The MOIS functions and their return value types are given below :-

- `expr` – returns string
- `expr_boolean` – returns Boolean
- `expr_integer` – returns Integer
- `expr_real` – returns Real

The requirements for the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <expr-result> [expr <expression>]`’ and convert these to the ‘expr’ MOIS function (i.e. the version returning a string value).

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<expr-result>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<expr-result>` and this is not of the String type.

d. The converter shall treat everything following the expr keyword up to the '`]`' bracket as the `<expression>` argument. It is not expected that the argument is a delimited string (although it may be). This contradicts the normal handling of string arguments which are single words or delimited strings.

e. The converter shall insert a MOIS comment statement prior to the function call to indicate that the expression has been converted by default to the string return variant, and may require manual edit to a correctly typed return variant.

2.5.13 fblocked - Test whether last input operation exhausted all input

`fblocked` *`channelId`*

The **fblocked** command returns 1 if the most recent input operation on *channelId* returned less information than requested because all available input was exhausted. For example, if **gets** is invoked when there are only three characters available for input and no end-of-line sequence, **gets** returns an empty string and a subsequent call to **fblocked** will return 1.

This command will be implemented as a MOIS function, with a single mandatory argument and returning a boolean value :-

- `channelId` – String identifying the channel to be checked.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <result> [fblocked <channel-id>]`' and convert these to the 'fblocked' MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<result>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<result>` and this is not of the Boolean type.

2.5.14 fconfigure - Set and get options on a channel

`fconfigure` *`channelId`*

`fconfigure` *`channelId name`*

`fconfigure` *`channelId name value ?name value ...?`*

The **fconfigure** command sets and retrieves options for channels. *ChannelId* identifies the channel for which to set or query an option. If no *name* or *value* arguments are supplied, the command returns a list containing alternating option names and values for the channel. If *name* is supplied but no *value* then the command returns the current value of the given option. If one or more pairs of *name* and *value* are supplied, the command sets each of the named options to the corresponding *value*; in this case the return value is an empty string.

This command is implemented as a function to cover the first two forms and a directive to handle the third form. The directive will not, however, support multiple option settings, as a repeated pair or arguments is not supported in the envisaged extensions to the function / directive mechanism. Further, only a single return type (string) will be supported for the functions, despite the different types returned depending on the specified options.

The first two forms of the command will be implemented by a MOIS function, returning a string value and taking the following arguments :-

- `channelId` – Mandatory string containing the channel identifier.
- `Name` – Optional string identifying the required channel configuration parameter

The last form of the copmmand is implemented as a MOIS directive taking the following arguments :-

- `channelId` – Mandatory string containing the channel identifier.
- `Name` – Mandatory string identifying the required channel configuration parameter
- `Value` – Mandatory string specifying the new configuration parameter setting

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <config-param> [fconfigure <channel-id> <name>?]`' and convert these to the 'fconfigure' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<config-param>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<config-param>` and this is not of the String type.

d. The MOIS converter shall recognise simple Tcl statements of the form '`fconfigure <channel-id> <name> <value>]`' and convert these to the 'fconfigure' MOIS directive.

2.5.15 fcopy - Copy data from one channel to another.

`fcopy` *`inchan outchan`* **`?-size`** *`size`*`?` **`?-command`** *`callback`* `?`

The **fcopy** command copies data from one I/O channel, *inchan* to another I/O channel, *outcha*n. The **fcopy** command leverages the buffering in the Tcl I/O system to avoid extra copies and to avoid buffering too much data in main memory when copying large files to slow destinations like network sockets.

This command will be implemented using a MOIS directive taking the following arguments :-

- `inchan` – Mandatory string identifying the channel id of the input channel
- `outchan` – Mandatory string identifying the channel id of the output channel
- `-size` – Switch argument with allowed value "-size"

- `size` – Integer argument identifying the number of bytes to copy. This argument is associated with the `-size` switch argument and is present if and only if the `-size` argument is present.
- `-command` – Switch argument with allowed value "-command"
- `callback` – String argument representing the Tcl commands to be executed when the copy is complete. This argument is associated with the `-command` switch argument and is present if and only if the `-command` argument is present. Note that the Tcl defined here will not be validated by MOIS in any way.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`fcopy <in-chan-id> <out-chan-id> (-size <size>)? (-command <callback-tcl>)?`' and convert these to the 'fcopy' MOIS directive.

b. The converter shall recognise <callback-tcl> either as a single word, or as an arbitrary string delimited by square brackets.

2.5.16 fileevent - Execute a script when a channel becomes readable or writable

`fileevent` *`channelId`* `readable ?`*`script`*`?`

`fileevent` *`channelId`* `writable ?`*`script`*`?`

This command is used to create *file event handlers*. A file event handler is a binding between a channel and a script, such that the script is evaluated whenever the channel becomes readable or writable. File event handlers are most commonly used to allow data to be received from another process on an event-driven basis, so that the receiver can continue to interact with the user while waiting for the data to arrive. If an application invokes **gets** or **read** on a blocking channel when there is no input data available, the process will block; until the input data arrives, it will not be able to service other events, so it will appear to the user to ``freeze up''. With **fileeven**t, the process can tell when data is present and only invoke **gets** or **read** when they won't block.

This command will not be implemented by MOIS procedures.

2.5.17 file - Manipulate file names and attributes

`file` `option name ?arg arg ...?`

This is a set of commands for operating on a file's name or attributes. *Name* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command. *Option* indicates what to do with the file name. Any unique abbreviation for *option* is acceptable (in Tcl, but not in any MOIS conversion).

The valid options are given below with their argument patterns and a brief description :-

`file atime` *`name`*

Returns an integer representing the time at which file *`name`* was last accessed, as a number of seconds from an epoch (c.f. `clock seconds`).

`file atime` *`name`* `time`

Sets the access time for the file *`name`*.

file attributes *name*

Returns the complete list of file attribute names and their values for the file *name*.

file attributes *name* `option`

Returns the value of the attribute specified by `option` for the file *name*.

file attributes *name* `option value ?option value...?`

Sets the value of the one or more attributes (specified by `option`) to the `value` for the file *name*.

file channels *?pattern?*

If *pattern* isn't specified, the command returns a list of names of all registered open channels in this interpreter. If *pattern* is specified, only those names matching the pattern are returned.

file copy ?-force? `?- -?` *source target*

Makes a copy of the file or directory *source* at the pathname defined in *target*. The `–force` switch forces overwrite of existing files if present. The – – switch terminates the switch list (allowing source to be defined with a leading '-' character).

file copy ?-force? `?- -?` *source ?source ...? targetDir*

Makes a copy of the files or directories defined in each instance of *source* in the existing directory defined by *targetDir*. The `–force` switch forces overwrite of existing files if present. The – – switch terminates the switch list (allowing source to be defined with a leading '-' character).

file delete ?-force? `?- -?` *pathname ?pathname ... ?*

Removes the file or directory specified by each *pathname* instance. Non-empty directories will be removed only if the `–force` option is specified. The `–force` switch forces overwrite of existing files if present. The – – switch terminates the switch list (allowing pathname to be defined with a leading '-' character).

file dirname *name*

Returns the directory path extracted from the file path defined by *name*.

file executable *name*

Returns a Boolean identifying whether the file defined by *name* is executable by the current user.

file exists *name*

Returns a Boolean identifying whether the file defined by *name* exists.

File extension *name*

Returns a string identifying the file extension extracted from the file path defined by *name*.

file isfile *name*

Returns a Boolean identifying whether the item defined by *name* is a file.

`file isdirectory` *`name`*

Returns a Boolean identifying whether the item defined by *`name`* is a directory.

`file join` `name ?name ...?`

Returns a string representing the file path constructed from the path elements defined by the instances of the `name` argument.

`file lstat` `name varName`

Invokes the lstat kernel call on `name`, and uses the variable given by `varName` to hold information returned from the kernel call. `VarName` is treated as an array variable.

`file mkdir` *`dir ?dir ...?`*

Creates the directories defined in each instance of *`dir`*.

`file mtime` *`name ?time`*`?`

Returns an integer representing the time at which file *`name`* was last modified, as a number of seconds from an epoch (c.f. `clock seconds`).

`file nativename` *`name`*

Returns a string representing the file path in *`name`* formatted according to the native operating system.

`file owned` *`name`*

Returns a Boolean identifying whether the file defined by *`name`* is owned by the current user.

`file pathtype` *`name`*

Returns the string "absolute", "relative" or "volumerelative", identifying the type of path defined by *`name`*.

`file readable` *`name`*

Returns a Boolean identifying whether the file defined by *`name`* is readable by the current user.

`file readlink` *`name`*

Returns a string representing the value of the symbolic link given by *`name`* (i.e. the name of the file it points to).

`file rename` `?-force? ?- -?` *`source target`*

Moves the file or directory *`source`* to the pathname defined in *`target`*. The `–force` switch forces overwrite of existing files if present. The `– –` switch terminates the switch list (allowing source to be defined with a leading '-' character).

`file rename` `?-force? ?- -?` *`source ?source ...? targetDir`*

Moves the files or directories defined in each instance of *`source`* to the existing directory defined by *`targetDir`*. The `–force` switch forces overwrite of existing files if present. The `– –` switch terminates the switch list (allowing source to be defined with a leading '-' character).

`file rootname` *`name`*

Returns a string representing the path defined in *`name`* excluding the file extension and associated period.

`file size` *`name`*

Returns an integer representing the size of the file defined in *`name`*.

`file split` *`name`*

Returns a string containing a Tcl list of the path elements defined in the path *`name`*.

`file stat` `name varName`

Invokes the stat kernel call on `name`, and uses the variable given by `varName` to hold information returned from the kernel call. `VarName` is treated as an array variable.

`file tail` *`name`*

Returns a string consisting of the file name, without directory information, extracted from the path defined in *`name`*.

`file type` *`name`*

Returns the string “file”, “directory”, “characterSpecial”, “blockSpecial”, “fifo”, “link”, or “socket” identifying the type of file *`name`*.

`file volume`

Returns the absolute paths to the volumes mounted on the system as a Tcl list string.

`file writable` *`name`*

Returns a Boolean identifying whether the file defined by *`name`* is writable by the current user.

It statement will not be implemented in MOIS procedures.

2.5.18 format - Format a string in the style of sprintf

`format` `formatString ?arg arg ...?`

This command generates a formatted string in the same way as the ANSI C **sprintf** procedure (it uses **sprintf** in its implementation). *FormatString* indicates how to format the result, using **%** conversion specifiers as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. The return value from **format** is the formatted string.

The repeated `arg` arguments are a problem here as they could be of any MOIS types. Probably the easiest way to deal with this is to take the same approach as for the expr command, assuming that everything following the `formatString` is part of a single string argument. As in the case of expr, this will mean that variable references can be made that are not verified by MOIS.

Therefore, it will be implemented the command as a MOIS function, returning a string value and taking the following arguments

- `formatString` – Mandatory string containing the formatting instructions
- `formatArgs` – Optional string containing all of the values to be formatted

The requirements for the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <format-result> [format <format-string> <format-args>?]`' and convert these to the 'format' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<format-result>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<format-result>` and this is not of the String type.

d. The converter shall treat everything following the `<format-string>` up to the '`]`' bracket as the `<format-args>` argument. It is not expected that the argument is a delimited string (although it may be). This contradicts the normal handling of string arguments which are single words or delimited strings.

2.5.19 gets - Read a line from a channel

`gets` `channelId ?varName?`

This command reads the next line from *channelId*, returns everything in the line up to (but not including) the end-of-line character(s), and discards the end-of-line character(s). If *varName* is omitted the line is returned as the result of the command. If *varName* is specified then the line is placed in the variable by that name and the return value is a count of the number of characters returned.

It will be implemented only as a MOIS directive with two mandatory arguments :-

- `channelId` – String argument representing the channel id of the channel being read.
- `varName` – String representing the name of the variable into which the data read from the channel is placed.

The requirements of the MOIS converter is as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`gets <channel-id> <var-name>`' and convert these to the 'gets' MOIS directive.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<var-name>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the String type.

2.5.20 global - Access global variables

`global` `varname ?varname ...?`

This command is ignored unless a Tcl procedure is being interpreted. If so then it declares the given *varname*'s to be global variables rather than local ones. Global variables are variables in the global namespace. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varnam*es will refer to the global variable by the same name.

MOIS is able to handle local, global and system variables. Then it will be possible to handle global variables.

2.5.21 glob - Return names of files that match patterns

`glob` `?switches? pattern ?pattern ...?`

This command performs file name "globbing" in a fashion similar to the csh shell. It returns a list of the files whose names match any of the *pattern* arguments.

If the initial arguments to **glob** start with - then they are treated as switches. The following switches are currently supported:

`-directory` *`directory`*

Search for files which match the given patterns starting in the given *director*y. This allows searching of directories whose name contains glob-sensitive characters without the need to quote such characters explicitly. This option may not be used in conjunction with **-path**.

`-join`

The remaining pattern arguments are treated as a single pattern obtained by joining the arguments with directory separators.

`-nocomplain`

Allows an empty list to be returned without error; without this switch an error is returned if the result list would be empty.

`-path` `pathPrefix`

Search for files with the given *pathPrefix* where the rest of the name matches the given patterns. This allows searching for files with names similar to a given file even when the names contain glob-sensitive characters. This option may not be used in conjunction with **-directory**.

`-types` `typeList`

Only list files or directories which match *typeLis*t, where the items in the list have two forms. The first form is like the -type option of the Unix find command: *b* (block special file), *c* (character special file), *d* (directory), *f* (plain file), *l* (symbolic link), *p* (named pipe), or *s* (socket), where multiple types may be specified in the list. **Glob** will return all files which match at least one of the types given.

The second form specifies types where all the types given must match. These are r, **w**, ***x*** as file permissions, and *readonl*y, *hidden* as special permission cases. On the Macintosh, MacOS types and creators are also supported, where any item which is four characters long is assumed to be a MacOS type (e.g. TEXT). Items which are of the form *{macintosh type XXXX}* or *{macintosh creator XXXX}* will match types or creators respectively. Unrecognised types, or specifications of multiple MacOS types/creators will signal an error. The two forms may be mixed, so **-types {d f r w}** will find all regular files OR directories that have both read AND write permissions.

`- -`

Marks the end of switches. The argument following this one will be treated as a pattern even if it starts with a -.

The *pattern* arguments may contain any of the following special characters:

- **`?`** Matches any single character.
- **`*`** Matches any sequence of zero or more characters.
- `[chars]` Matches any single character in *chars*. If *chars* contains a sequence of the form **a-***b* then any character between *a* and *b* (inclusive) will match.
- `\x` Matches the character x.
- `{a,b,...}` Matches any of the strings a, b, etc.

The command will be implemented as a MOIS function, returning a string value and taking the following arguments.

- `-directory` – This is a switch argument that takes the value '-directory'.
- `<dir-name>` - This is string argument associated with the `-directory` switch, it is present if and only if the `-directory` switch is present. It defines the directory in which the file search is to be performed.
- `-join` – This is a switch argument that takes the value '-join'.
- `-nocomplain` – This is a switch argument that takes the value '-nocomplain'.
- `-path` – This is a switch argument that takes the value '-path'.
- `<path-prefix>` - This is string argument associated with the `-path` switch, it is present if and only if the `-path` switch is present. It defines a path prefix within which the file search is to be made.
- `-types` - This is a switch argument that takes the value '-types'.
- `<type-list>` - This is string argument associated with the `-types` switch, it is present if and only if the `-types` switch is present. It defines a list of file types for which the file search is to be made.
- `- -` - This is a switch argument that takes the value '- -'. It is used to indicate the end of the switches in cases where the pattern begins with a '-' character.
- `pattern` – Repeated string argument defining the file name matching pattern(s) to be used.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <file-list> [glob <switches>* (- -)? <pattern>+ ]`' where '`<switches> = -directory <dir-name> | -join | -nocomplain | -path <path-prefix> | -types <type-list>`' and convert these to the 'glob' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<file-list>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<file-list>` and this is not of the String type.

2.5.22 incr - Increment the value of a variable

incr varName ?increment?

Increments the value stored in the variable whose name is *varName*. The value of the variable must be an integer. If *increment* is supplied then its value (which must be an integer) is added to the value of variable *varName*; otherwise 1 is added to *varName*. The new value is stored as a decimal string in variable *varName* and also returned as result.

This command will be implemented as a MOIS set statement (as modified to allow definitions of increments and offsets). The converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`incr <var-name> <inc-value>?`' and convert these to the MOIS SET statements.

b. The converter shall raise an error and fail the statement conversion if a MOIS variable corresponding to `<var-name>` does not exist or corresponds to a non-numeric type.

c. The converter shall raise an error and fail the statement conversion if the `<inc-value>` is not an integer.

d. The converter shall set the increment value to 1 if no `<inc-value>` is specified.

2.5.23 info - Return information about the state of the Tcl interpreter

info option ?arg arg ...?

This command provides information about various internals of the Tcl interpreter.

This command will not be implemented in MOIS procedures.

2.5.24 join - Create a string by joining together list elements

join list ?joinString?

The *list* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *list* together with *joinString* separating each adjacent pair of elements. The *joinString* argument defaults to a space character.

This command will be implemented as a MOIS function, returning a string value and taking the following arguments :-

- `list` – Mandatory string containing the list of items to be joined.
- `JoinString` – Optional string identifying the separator character(s).

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <join-string> [join <string-list> <separator>?]`’ and convert these to the ‘join’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<join-string>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<join-string>` and this is not of the String type.

2.5.25 lappend - Append list elements onto a variable

`lappend` `varName ?value value value ...?`

This command treats the variable given by *varName* as a list and appends each of the *value* arguments to that list as a separate element, with spaces between elements. If *varName* doesn't exist, it is created as a list with elements given by the *value* arguments.

The command will be implemented as a MOIS directive with the following arguments :-

- `varName` – Mandatory string argument identifying the name of the variable to which the strings defined in `value` are to be appended.
- `value` – Repeated string argument identifying the values to be appended to the list.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form ‘`lappend <var-name> <value>*`’ and convert these to the ‘lappend’ MOIS directive.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<var-name>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` and this is not of the String type.

2.5.26 lindex - Retrieve an element from a list

`lindex` `list index`

This command treats *list* as a Tcl list and returns the *inde*x'th element from it (0 refers to the first element of the list). In extracting the element, *lindex* observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *index* is negative or greater than or equal to the number of elements in *valu*e, then an empty string is returned. If *index* has the value **end**, it refers to the last element in the list, and **end-***integer* refers to the last element in the list minus the specified integer offset.

The command will be implemented as a MOIS function returning a string value and taking the following arguments :-

- `list` – Mandatory string representing a Tcl list.

- `index` – Mandatory string representing the index position (needs to be a string to handle the ‘end’ and ‘end-*x*’ formats).

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [lindex <list-string> <index>]`’ and convert these to the ‘lindex’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.27 linsert - Insert elements into a list

`linsert` `list index element ?element element ...?`

This command produces a new list from *list* by inserting all of the *element* arguments just before the *index*th element of *list*. Each *element* argument will become a separate element of the new list. If *index* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *index* has the value **end**, or if it is greater than or equal to the number of elements in the list, then the new elements are appended to the list. **end-***integer* refers to the last element in the list minus the specified integer offset.

The command will be implemented as a MOIS function, returning a string value and taking the following arguments :-

- `list` – Mandatory string representing a Tcl list.
- `index` – Mandatory string identifying the index position within the list at which the new elements are to be inserted.
- `element` – Mandatory, repeated string argument defining the new elements to be inserted.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [linsert <list-string> <index> <new-element>*]`’ and convert these to the ‘linsert’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.28 list - Create a list

`list` `?arg arg ...?`

This command returns a list comprised of all the *arg*s, or an empty string if no *arg*s are specified. Braces and backslashes get added as necessary, so that the **index** command may be used on the result to re-extract the original arguments, and also so that **eval** may be used to execute the resulting list, with *arg1* comprising the command's name and the other *ar*gs comprising its arguments. **List** produces slightly different results than **concat**: **concat** removes one level of grouping before forming the list, while **list** works directly from the original arguments. For example, the command '`list a b {c d e} {f {g h}}`' will return '`a b {c d e} {f {g h}}`' while **concat** with the same arguments will return '`a b c d e f {g h}`'.

The command will be implemented as a MOIS function, returning a string value and taking the following arguments :-

- `arg` – Optional, repeated string argument defining the new elements to be inserted.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [list <new-element>*]`' and convert these to the 'list' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.29 llength - Count the number of elements in a list

`llength` *`list`*

Treats *list* as a list and returns a decimal string giving the number of elements in it.

This command will be implemented as a MOIS function returning an integer value and taking the following arguments :-

- `list` – Mandatory string value representing a Tcl list.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [llength <list>]`' and convert these to the 'llength' MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.30 lrange - Return one or more adjacent elements from a list

`lrange` `list first last`

List must be a valid Tcl list. This command will return a new list consisting of elements *first* through *las*t, inclusive. *First* or *last* may be **end** (or any abbreviation of it) to refer to the last element of the list. If *first* is less than zero, it is treated as if it were zero. If *last* is greater than or equal to the number of elements in the list, then it is treated as if it were **end**. If *first* is greater than *last* then an empty string is returned. The command will be implemented as a MOIS function returning a string value and taking the following arguments :-

- `list` – Mandatory string representing a Tcl list.
- `first` – Mandatory string representing the index position (needs to be a string to handle the 'end' and 'end-*x*' formats).
- `last` – Mandatory string representing the index position (needs to be a string to handle the 'end' and 'end-*x*' formats).

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [lrange <list-string> <first> <last>]`' and convert these to the 'lrange' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.31 lreplace - Replace elements in a list with new elements

`lreplace` `list first last ?element element ...?`

lreplace returns a new list formed by replacing one or more elements of *list* with the *element* arguments. *first* and *last* specify the first and last index of the range of elements to replace. 0 refers to the first element of the list, and **end** (or any abbreviation of it) may be used to refer to the last element of the list. If *list* is empty, then *first* and *last* are ignored. If *first* is less than zero, it is considered to refer to the first element of the list. For non-empty lists, the element indicated by *first* must exist. If *last* is less than zero but greater than *firs*t, then any specified elements will be prepended to the list. If *last* is less than *first* then no elements are deleted; the new elements are simply inserted before *firs*t. The *element* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *element* argument will become a separate element of the list. If no *element* arguments are specified, then the elements between *first* and *last* are simply deleted. If *list* is empty, any *element* arguments are added to the end of the list.

The command will be implemented as a MOIS function returning a string value and taking the following arguments :-

- `list` – Mandatory string representing a Tcl list.
- `first` – Mandatory string representing the index position (needs to be a string to handle the 'end' and 'end-*x*' formats).
- `last` – Mandatory string representing the index position (needs to be a string to handle the 'end' and 'end-*x*' formats).
- Element – Optional repeated string argument defining the new list elements to be inserted.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [lreplace <list-string> <first> <last> <list-element>*]`’ and convert these to the ‘lreplace’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.32 lsearch - See if a list contains a particular element

`lsearch` `?mode? list pattern`

This command searches the elements of *list* to see if one of them matches *pattern*. If so, the command returns the index of the first matching element. If not, the command returns **-**1. The *mode* argument indicates how the elements of the list are to be matched against *pattern.*

The command will be implemented as a MOIS function returning an Integer value, and taking the following arguments :-

- `mode` – Switch argument which can take the value ‘-exact’, ‘-glob’ or ‘-regexp’ (or may not be specified).
- `list` – Mandatory string argument representing the Tcl list to be searched.
- `pattern` – Mandatory string argument specifying the pattern to be matched

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [lsearch <mode-switch>? <list-string> <pattern>]`’ and convert these to the ‘lsearch’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.33 lsort - Sort the elements of a list

`lsort` `?options? list`

This command sorts the elements of *list*, returning a new list in sorted order. The implementation of the **lsort** command uses the merge-sort algorithm which is a stable sort that has O(n log n) performance characteristics.

The command will be implemented as a MOIS function, returning a string value and taking the arguments defined below. Note that for simplicity, the `-command` option is not supported by MOIS.

- `sort-type` – Optional switch argument taking the values '-ascii' (this is the default), '-dictionary', '-integer' or '-real'. Defines the type of sort to be performed.
- `sort-direction` – Optional switch argument taking the values '-increasing' or '-decreasing'
- `sort-index` – Optional switch argument taking the value '-index'
- `index` – String argument associated with the `sort-index` switch above. It is present if and only if `sort-index` is set to '-index'.
- `unique` - Optional switch argument taking the value '-unique'
- `list` – Mandatory string argument representing the list to be sorted.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [lsort <sort-type>? <sort-direction>? (<sort-index> <index>)? <unique>? <list-string>]`' and convert these to the 'lsort' MOIS function.
b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.
c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.34 memory - Control Tcl memory debugging capabilities.

`memory` `option ?arg arg ...?`

The **memory** command gives the Tcl developer control of Tcl's memory debugging capabilities. The memory command has several suboptions, which are described below. It is only available when Tcl has been compiled with memory debugging enabled (when **TCL_MEM_DEBUG** is defined at compile time).

This command will not be implemented in MOIS procedures.

2.5.35 namespace - create and manipulate contexts for commands and variables

`namespace` *`?option? ?arg ...?`*

The **namespace** command lets you create, access, and destroy separate contexts for commands and variables.

This command will not be implemented in MOIS procedures.

2.5.36 open - Open a file-based or command pipeline channel

`open` `fileName`

`open` `fileName access`

`open` `fileName access permissions`

This command opens a file, serial port, or command pipeline and returns a channel identifier that may be used in future invocations of commands like **read**, **puts**, and **close**. If the first character of *filename* is not | then the command opens a file: *fileName* gives the name of the file to open, and it must conform to the conventions described in the **filename** manual entry. The *access* argument, if present, indicates the way in which the file (or command pipeline) is to be accessed.

The third form is not supported by MOIS, this is because the arguments are somewhat ill-conditioned (they can only be identified by their value rather than by type or position).

The first two forms will be implemented by a MOIS directive with the following arguments :-

- `fileName` – Mandatory string identifying the file or channel to open.
- `access` – Optional string identifying the file access required.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form ‘`open <file-name> <access>`’ and convert these to the ‘open’ MOIS directive.

2.5.37 pid - Retrieve process id(s)

`pid` *`?fileId`*`?`

If the *fileId* argument is given then it should normally refer to a process pipeline created with the **open** command. In this case the **pid** command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if *fileId* refers to an open file that isn't a process pipeline. If no *fileId* argument is given then **pid** returns the process identifier of the current process. All process identifiers are returned as decimal strings.

The command will be implemented as a MOIS function returning an Integer value and taking the following arguments :-

- `fileId` – Optional string identifying the file

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [pid <file-id>?]`’ and convert these to the ‘pid’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.38 puts - Write to a channel

`puts ?-nonewlin``e?` *`?channelId`*`?` *`string`*

Writes the characters given by *string* to the channel given by *channelId*. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or

socket. It must have been opened for output. If no *channelId* is specified then it defaults to **stdout**. **Puts** normally outputs a newline character after *strin*g, but this feature may be suppressed by specifying the **-nonewline** switch.

This command is rather poorly specified, as it includes an optional argument which is not a switch and also not at the end of the argument list. The channelId will be defined as a mandatory argument, but the MOIS converter will set the value to '' (empty string) if the argument is not included.

The command will be implemented as a MOIS directive with the following arguments :-

- `nonewline` – Optional switch argument taking value '-nonewline'.
- `channelId` – Mandatory string argument identifying the channel to be written to
- `string` – Mandatory string argument defining the text to be written.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`puts <nonewline>? <channel-id> <out-string>`' and convert these to the 'puts' MOIS function.

b. The MOIS converter shall recognise simple Tcl statements of the form '`puts <nonewline>? <out-string>`' and convert these to the 'puts' MOIS function. In this case the `<channel-id>` argument shall be set to ''.

2.5.39 pwd - Return the current working directory

```
pwd
```

Returns the path name of the current working directory.

The command will be implemented as a MOIS function with no arguments and returning a string value.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> pwd`' and convert these to the 'pwd' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.40 read - Read from a channel

```
read ?-nonewline? channelId
```

```
read channelId numChars
```

In the first form, the **read** command reads all of the data from *channelId* up to the end of the file. If the **-nonewline** switch is specified then the last character of the file is discarded if it is a newline. In the second form, the extra argument specifies how many characters to read. Exactly that many characters will be read and returned, unless there are fewer than *numChars* left in the file; in this case all the remaining characters are returned. If the channel is configured to use a multi-byte encoding, then the number of characters read may not be the same as the number of bytes read.

The command will be implemented as a MOIS directive with the following arguments :-

- `nonnewline` – Optional switch argument, taking the value '-nonnewline'.
- `channelId` – Mandatory string argument identifying the channel to be read from.
- `numChars` – Optional integer argument identifying the number of characters to be read

The MOIS converter requirements are as follwos :-

a. The MOIS converter shall recognise simple Tcl statements of the form 7'`read <nonewline>? <channel-id> <num-chars>?`' and convert these to the 'read' MOIS directive.

2.5.41 regexp - Match a regular expression against a string

```
regexp ?switches? exp string ?matchVar? ?subMatchVar
subMatchVar ...?
```

Determines whether the regular expression *exp* matches part or all of *string* and returns 1 if it does, 0 if it doesn't, unless **-inline** is specified (see below).

If additional arguments are specified after *string* then they are treated as the names of variables in which to return information about which part(s) of *string* matched *ex*p. *MatchVar* will be set to the range of *string* that matched all of *ex*p. The first *subMatchVar* will contain the characters in *string* that matched the leftmost parenthesized subexpression within *ex*p, the next *subMatchVar* will contain the characters that matched the next parenthesized subexpression to the right in *ex*p, and so on. If the initial arguments to **regexp** start with **-** then they are treated as switches.

The command will be implemented as a MOIS function, returning a boolean value and taking the following arguments :-

- `about` – Optional switch argument taking the value '-about'
- `expanded` – Optional switch argument taking the value '-expanded'
- `indices` – Optional switch argument taking the value '-indices'
- `line` – Optional switch argument taking the value '-line'
- `linestop` – Optional switch argument taking the value '-linestop'
- `lineanchor` – Optional switch argument taking the value '-lineanchor'
- `nocase` – Optional switch argument taking the value '-nocase'
- `all` – Optional switch argument taking the value '-all'
- `inline` – Optional switch argument taking the value '-inline'
- `start` – Optional switch argument taking the value '-start'
- `startIndex` – Integer argument defining the search start position, associated with the `start` switch such that it is present if and only if the `start` switch is present.
- `--` Optional switch argument taking the value '--'

- `regularExp` – Mandatory string argument defining the regular expression to be used in the match.
- `matchStr` – Mandatory string argument defining the string to be matched against the regular expression
- `matchVar` – Optional, repeated string argument defining variable names in which the matched string segments are to be placed.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [regexp <about>? <expanded>? <indices>? <line>? <linestop>? <lineanchor>? <nocase>? <all>? <inline>? (<start> <start-index>)? <->? <regular-exp> <match-str> <match-var>*]`' and convert these to the 'regexp' MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Boolean type.

d. The converter shall create a MOIS local variable of String type with the name corresponding to `<match-var>` for each instance of `<match-var>`, provided a variable of that name does not already exist.

e. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<match-var>` (for each instance) and this is not of the String type.

2.5.42 regsub - Perform substitutions based on regular expression pattern matching

`regsub` `?switches? exp string subSpec varName`

This command matches the regular expression *exp* against *string*, and it copies *string* to the variable whose name is given by *varName*. If there is a match, then while copying *string* to *varName* the portion of *string* that matched *exp* is replaced with *subSpec*. If *subSpec* contains a ``&'' or ``\0'', then it is replaced in the substitution with the portion of *string* that matched *exp*. If *subSpec* contains a ``\ n'', where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of *string* that matched the n-th parenthesized subexpression of *exp*. Additional backslashes may be used in *subSpec* to prevent special interpretation of ``&'' or ``\0'' or ``\n'' or backslash. The use of backslashes in *subSpec* tends to interact badly with the Tcl parser's use of backslashes, so it's generally safest to enclose *subSpec* in braces if it includes backslashes. If the initial arguments to **regexp** start with **-** then they are treated as switches.

The command will be implemented as a MOIS function, returning a boolean value and taking the following arguments :-

- `all` – Optional switch argument taking the value '-about'
- `expanded` – Optional switch argument taking the value '-expanded'

- `line` – Optional switch argument taking the value '-line'
- `linestop` – Optional switch argument taking the value '-linestop'
- `lineanchor` – Optional switch argument taking the value '-lineanchor'
- `nocase` – Optional switch argument taking the value '-nocase'
- `inline` – Optional switch argument taking the value '-inline'
- `start` – Optional switch argument taking the value '-start'
- `startIndex` – Integer argument defining the search start position, associated with the `start` switch such that it is present if and only if the `start` switch is present.
- `--` Optional switch argument taking the value '--'
- `regularExp` – Mandatory string argument defining the regular expression to be used in the match.
- `matchStr` – Mandatory string argument defining the string to be matched against the regular expression
- `replaceStr` – Mandatory string argument defining the new string to be used to replace the matched string.
- `matchVar` – Mandatory string argument defining the variable name in which the replaced string is placed.

The requirements on the MOIS converter are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`regsub <all>? <expanded>? <line>? <linestop>? <lineanchor>? <nocase>? <inline>? (<start> <start-index>)? <->? <regular-exp> <match-str> <replace-str> <match-var>]`' and convert these to the 'regsub' MOIS directive.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<match-var>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<match-var>` and this is not of the String type.

2.5.43 rename - Rename or delete a command

`rename` `oldName newName`

Rename the command that used to be called *oldName* so that it is now called *newName*. If *newName* is an empty string then *oldName* is deleted. *oldName* and *newName* may include namespace qualifiers (names of containing namespaces). If a command is renamed into a different namespace, future invocations of it will execute in the new namespace. The **rename** command returns an empty string as result.

This command will not be implemented in MOIS procedures.

2.5.44 scan - Parse string using conversion specifiers in the style of sscanf

`scan` `string format ?varName varName ...?`

This command parses fields from an input string in the same fashion as the ANSI C **sscanf** procedure and returns a count of the number of conversions performed, or -1 if the end of the input string is reached before any conversions have been performed. *String* gives the input to be parsed and *format* indicates how to parse it, using **%** conversion specifiers as in **sscan**f. Each *varName* gives the name of a variable; when a field is scanned from *string* the result is converted back into a string and assigned to the corresponding variable. If no *varName* variables are specified, then **scan** works in an inline manner, returning the data that would otherwise be stored in the variables as a list. In the inline case, an empty string is returned when the end of the input string is reached before any conversions have been performed.

This command will be implemented as both a directive and a function to deal with the normal and inline forms. The directive form takes the following arguments (note that the integer return value is not modelled by MOIS) :-

- `string` – Mandatory string argument, identifying the string to be parsed
- `format` – Mandatory string argument, identifying the parse instructions
- `varName` – Mandatory, repeated string argument representing variable names in which the parsed elements are placed.

The function form returns a string value containing a Tcl list of the parsed elements and takes the following arguments :-

- `string` – Mandatory string argument, identifying the string to be parsed
- `format` – Mandatory string argument, identifying the parse instructions

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form ‘`scan <parse-string> <format> <var-name>+`’ and convert these to the ‘scan’ MOIS directive.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<var-name>` for each instance of `<var-name>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<var-name>` (for each instance) and this is not of the String type.

d. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <element-list> [scan <parse-string> <format>]`’ and convert these to the ‘scan’ MOIS function.

e. The converter shall create a MOIS local variable of String type with the name corresponding to `<element-list>`, provided a variable of that name does not already exist.

f. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<element-list>` and this is not of the String type.

2.5.45 seek - Change the access position for an open channel

`seek` `channelId offset ?origin?`

Changes the current access position for *channelI*d. *ChannelId* must be a channel identifier such as returned from a previous invocation of **open** or **socket**. The *offset* and *origin* arguments specify the position at which the next read or write will occur for *channelI*d. *Offset* must be an integer (which may be negative) and *origin* must be one of the following:

The command will be implemented as a MOIS directive taking the following arguments :-

- `channelId` – Mandatory string argument representing the identifier for the channel
- `offset` – Mandatory integer argument identifying the byte offset to be applied to the channel.
- `origin` – Optional switch argument, taking the values 'start', 'end' or 'current'.

The MOIS converter requirements are :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`seek <channel-id> <offset> <origin>?`' and convert these to the 'seek' MOIS directive

2.5.46 set - Read and write variables

`set` `varName ?value?`

Returns the value of variable *varName*. If *value* is specified, then set the value of *varName* to *valu*e, creating a new variable if one doesn't already exist, and return its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

This command will be implemented as a MOIS SET var statement.

- `varName` – Mandatory string argument identifying the name of the variable to be set.
- `value` – Mandatory string argument identifying the value to be set.

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise simple Tcl statements of the form '`set <var-name> <value>`' and convert these to the 'set' MOIS directive.

2.5.47 socket - Open a TCP network connection

`socket` `?options? host port`

`socket -server` `command ?options? port`

This command opens a network socket and returns a channel identifier that may be used in future invocations of commands like **read**, **puts** and **flush**. At present only the TCP network protocol is supported; future releases may include support for additional protocols. The **socket** command may be used to open either the client or server side of a connection, depending on whether the **-server** switch is specified.

This command will not be implemented in MOIS procedures.

2.5.48 source - Evaluate a file or resource as a Tcl script

`source` `fileName`

`source -rsrc` `resourceName ?fileName?`

`source -rsrcid` `resourceId ?fileName?`

This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from **source** is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script then the **source** command will return that error. If a **return** command is invoked from within the script then the remainder of the file will be skipped and the **source** command will return normally with the result from the **return** command. The *–rsrc* and *-rsrcid* forms of this command are only available on Macintosh computers. These versions of the command allow you to source a script from a **TEXT** resource. You may specify what **TEXT** resource to source by either name or id. By default Tcl searches all open resource files, which include the current application and any loaded C extensions. Alternatively, you may specify the *filename* where the **TEXT** resource can be found.

This command will not be implemented in MOIS procedures.

2.5.49 split - Split a string into a proper Tcl list

`split` `string ?splitChars?`

Returns a list created by splitting *string* at each character that is in the *splitChars* argument. Each element of the result list will consist of the characters from *string* that lie between instances of the characters in *splitChar*s. Empty list elements will be generated if *string* contains adjacent characters in *splitChar*s, or if the first or last character of *string* is in *splitChar*s. If *splitChars* is an empty string then each character of *string* becomes a separate element of the result list. *SplitChars* defaults to the standard white-space characters.

The command will be implemented as a MOIS function, returning a string value (representing a Tcl list) and taking the following arguments :-

- `string` – Mandatory string argument identifying the string to split.
- `splitChar` – Optional string argument identifying the separator character(s).

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [split <split-string> <separator>?]`' and convert these to the 'split' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50 string - Manipulate strings

`string` `option arg ?arg ...?`

Performs one of several string operations, depending on *optio*n. The legal *optio*ns (which may be abbreviated) are:

This is a set of commands for operating on a strings. The valid options are given below with their argument patterns and a brief description :-

`string bytelength` *`string`*

Returns a decimal string giving the number of bytes used to represent *string* in memory. Because UTF-8 uses one to three bytes to represent Unicode characters, the byte length will not be the same as the character length in general. The cases where a script cares about the byte length are rare. In almost all cases, you should use the **string length** operation.

`string compare` `?-nocase? ?-length int?` *`string1 string2`*

Perform a character-by-character comparison of strings *string1* and *string*2. Returns -1, 0, or 1, depending on whether *string1* is lexicographically less than, equal to, or greater than *string*2. If **–length** is specified, then only the first *length* characters are used in the comparison. If **-length** is negative, it is ignored. If **-nocase** is specified, then the strings are compared in a case-insensitive manner.

`string equal` `?-nocase? ?-length int?` *`string1 string2`*

Perform a character-by-character comparison of strings *string1* and *string*2. Returns 1 if *string1* and *string2* are identical, or 0 when not. If **-length** is specified, then only the first *length* characters are used in the comparison. If **-length** is negative, it is ignored. If **-nocase** is specified, then the strings are compared in a case-insensitive manner.

`string first` `string1 string2 ?startIndex?`

Search *string2* for a sequence of characters that exactly match the characters in *string*1. If found, return the index of the first character in the first such match within *string*2. If not found, return -1. If *startIndex* is specified (in any of the forms accepted by the **index** method), then the search is constrained to start with the character in *string2* specified by the index.

`string index` `string charIndex`

Returns the *charIndex*'th character of the *string* argument. A *charIndex* of 0 corresponds to the first character of the string. *charIndex* may be specified as follows:

- *`integer`* `-` The char specified at this integral index
- **`end -`** The last char of the string.
- **`end-`***`integer`* `-` The last char of the string minus the specified integer offset (e.g. **end-1** would refer to the "c" in "abcd"). If *charIndex* is less than 0 or greater than or equal to the length of the string then an empty string is returned.

`string is` *`class`* `?-strict? ?-failindex` *`varname`*`?` *`string`*

Returns 1 if *string* is a valid member of the specified character class, otherwise returns 0. If **–strict** is specified, then an empty string returns 0, otherwise and empty string will return 1 on any class. If **-failindex** is specified, then if the function returns 0, the index in the string where the class was no longer valid will be stored in the variable named *varname*. The *varname* will not be set if the function returns 1. The following character classes are recognized (the class name can be abbreviated):

- **`alnum -`** Any Unicode alphabet or digit character.
- **`alpha -`** Any Unicode alphabet character.
- **`ascii -`** Any character with a value less than \u0080 (those that are in the 7-bit ascii range).
- **`boolean -`** Any of the forms allowed to **Tcl_GetBoolean**.
- **`control -`** Any Unicode control character.
- **`digit -`** Any Unicode digit character. Note that this includes characters outside of the [0-9] range.
- **`double -`** Any of the valid forms for a double in Tcl, with optional surrounding whitespace. In case of under/overflow in the value, 0 is returned and the *varname* will contain -1.
- **`false -`** Any of the forms allowed to **Tcl_GetBoolean** where the value is false.
- **`graph -`** Any Unicode printing character, except space.
- **`integer -`** Any of the valid forms for an integer in Tcl, with optional surrounding whitespace. In case of under/overflow in the value, 0 is returned and the *varname* will contain -1.
- **`lower -`** Any Unicode lower case alphabet character.
- **`print -`** Any Unicode printing character, including space.
- **`punct -`** Any Unicode punctuation character.
- **`space -`** Any Unicode space character.
- **`true -`** Any of the forms allowed to **Tcl_GetBoolean** where the value is true.
- **`upper -`** Any upper case alphabet character in the Unicode character set.
- **`wordchar -`** Any Unicode word character. That is any alphanumeric character, and any Unicode connector punctuation characters (e.g. underscore).
- **`xdigit -`** Any hexadecimal digit character ([0-9A-Fa-f]). In the case of **boolea**n, **true** and **false** , if the function will return 0, then the *varname* will always be set to 0, due to the varied nature of a valid boolean value.

`string last` `string1 string2 ?startIndex?`

Search *string2* for a sequence of characters that exactly match the characters in *string*1. If found, return the index of the first character in the last such match within *string*2. If there is no match, then return -1. If *startIndex* is specified (in any of the forms accepted by the **index** method), then only the characters in *string2* at or before the specified *startIndex* will be considered by the search.

`string length` *`string`*

Returns a decimal string giving the number of characters in *strin*g. Note that this is not necessarily the same as the number of bytes used to store the string.

`string map` `?-nocase?` *`charMap string`*

Replaces characters in *string* based on the key-value pairs in *charMa*p. *charMap* is a list of *key value key value* ... as in the form returned by **array ge**t. Each instance of a key in the string will be replaced with its corresponding value. If **-nocase** is specified, then matching is done without regard to case differences. Both *key* and *value* may be multiple characters. Replacement is done in an ordered manner, so the key appearing first in the list will be checked first, and so on. *string* is only iterated over once, so earlier key replacements will have no affect for later key matches.

`string match` `?-nocase?` *`pattern string`*

See if *pattern* matches *strin*g; return 1 if it does, 0 if it doesn't. If **-nocase** is specified, then the pattern attempts to match against the string in a case insensitive manner. For the two strings to match, their contents must be identical except that the following special sequences may appear in *patter*n:

- **`* -`** Matches any sequence of characters in *strin*g, including a null string.
- **`? -`** Matches any single character in *strin*g.
- *`[chars`***`] -`** Matches any character in the set given by *char*s. If a sequence of the form *x*-*y* appears in *char*s, then any character between *x* and y, inclusive, will match. When used with **-nocase** , the end points of the range are converted to lower case first. Whereas {[A-z]} matches '_' when matching case-sensitively ('_' falls between the 'Z' and 'a'), with **-nocase** this is considered like {[A-Za-z]} (and probably what was meant in the first place).
- `\`*`x`* - Matches the single character x. This provides a way of avoiding the special interpretation of the characters ***?[]\** in *patter*n.

`string range` `string first last`

Returns a range of consecutive characters from *strin*g, starting with the character whose index is *first* and ending with the character whose index is *las*t. An index of 0 refers to the first character of the string. *first* and *last* may be specified as for the **index** method. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were **en**d. If *first* is greater than *last* then an empty string is returned.

`string repeat` *`string count`*

Returns *string* repeated *count* number of times.

`string replace` `string first last ?newstring?`

Removes a range of consecutive characters from *strin*g, starting with the character whose index is *first* and ending with the character whose index is *las*t. An index of 0 refers to the first character of the string. *First* and *last* may be specified as for the **index** method. If *newstring* is specified, then it is placed in the removed character range. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were **end**. If *first* is greater than *last* or the length of the initial string, or *last* is less than 0, then the initial string is returned untouched.

`string tolower` *`string ?first? ?last?`*

Returns a value equal to *string* except that all upper (or title) case letters have been converted to lower case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

`string totitle` *`string ?first? ?last?`*

Returns a value equal to *string* except that the first character in *string* is converted to its Unicode title case variant (or upper case if there is no title case variant) and the rest of the string is converted to lower case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

`string toupper` *`string ?first? ?last?`*

Returns a value equal to *string* except that all lower (or title) case letters have been converted to upper case. If *first* is specified, it refers to the first char index in the string to start modifying. If *last* is specified, it refers to the char index in the string to stop at (inclusive). *first* and *last* may be specified as for the **index** method.

`string trim` *`string ?chars?`*

Returns a value equal to *string* except that any leading or trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

`string trimleft` *`string ?chars?`*

Returns a value equal to *string* except that any leading characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

`string trimright` *`string ?chars?`*

Returns a value equal to *string* except that any trailing characters from the set given by *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

`string wordend` `string charIndex`

Returns the index of the character just after the last one in the word containing character *charIndex* of *string*. *charIndex* may be specified as for the **index** method. A word is considered to be any contiguous range of alphanumeric (Unicode letters or decimal digits) or underscore (Unicode connector punctuation) characters, or any single character other than these.

`string wordstart` `string charIndex`

Returns the index of the first character in the word containing character *charIndex* of *string*. *charIndex* may be specified as for the **index** method. A word is considered to be any contiguous range of alphanumeric (Unicode letters or decimal digits) or underscore (Unicode connector punctuation) characters, or any single character other than these.

The implementation and MOIS converter requirements for each variant of string are detailed in the subsections below.

2.5.50.1 string bytelength

Command will be implemented as a MOIS function, returning an Integer value and taking a single mandatory argument :-

- `string` – String for which the length is required

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string bytelength <string>]`’ and convert these to the ‘string bytelength’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.2 string compare

Command will be implemented as a MOIS function, returning an Integer value and taking arguments as follows :-

- `nocase` – Optional switch argument with allowed value ‘-nocase’.
- `length` – Optional switch argument with allowed value ‘-length’.
- `lengthVal` – Integer argument representing the comparison length. Argument is associated with the `length` switch and is present if and only if the `length` switch is present.
- `string1` – First comparison string
- `string2` – Second comparison string

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string compare <nocase>? (<length> <length-val>)? <string1> <string2>]`’ and convert these to the ‘string compare’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.3 string equal

Command will be implemented as a MOIS function, returning an Boolean value and taking arguments as follows :-

- `nocase` – Optional switch argument with allowed value '-nocase'.
- `length` – Optional switch argument with allowed value '-length'.
- `lengthVal` – Integer argument representing the comparison length. Argument is associated with the `length` switch and is present if and only if the `length` switch is present.
- `string1` – First comparison string. Mandatory.
- `string2` – Second comparison string. Mandatory.

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string equal <nocase>? (<length> <length-val>)? <string1> <string2>]`' and convert these to the 'string equal' MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Boolean type.

2.5.50.4 string first

Command will be implemented as a MOIS function, returning an Integer value and taking arguments as follows :-

- `string1` – Mandatory string argument representing the text being searched for
- `string2` – Mandatory string argument representing the string being searched
- `startIndex` – Optional integer argument identifying the start point for the search

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string first <string1> <string2> <start-index>?]`' and convert these to the 'string first' MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.5 string index

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument representing the text being searched
- `charIndex` – Mandatory string argument identifying the position of the character to be returned

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string index <string> <char-index>]`' and convert these to the 'string index' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.6 string is

Command will be implemented as a MOIS function, returning a Boolean value and taking arguments as follows :-

- `class` – Mandatory switch argument identifying the character class being checked for. The valid values are :- 'alnum', 'alpha', 'ascii', 'boolean', 'control', 'digit', 'double', 'false', 'graph', 'integer', 'lower', 'print', 'punct', 'space', 'true', 'upper', 'wordchar' or 'xdigit'.
- `strict` – Optional switch argument with valid value '-strict'.
- `failIndex` – Optional switch argument with valid value '-failindex'.
- `failVar` – String argument representing a variable name. This is associated with the `failIndex` switch, and is present if and only if the switch is present.
- `String` – Mandatory string argument identifying the string to be checked

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string is <class> <strict>? (<fail-index> <fail-var>)? <string>`' and convert these to the 'string is' MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Boolean type.

d. If the `<fail-var>` argument is specified, the converter shall create a MOIS local variable of String type with the name corresponding to `<fail-var>`, provided a variable of that name does not already exist.

e. If the `<fail-var>` argument is specified, the converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<fail-var>` and this is not of the String type.

2.5.50.7 string last

Command will be implemented as a MOIS function, returning an Integer value and taking arguments as follows :-

- `string1` – Mandatory string argument representing the text being searched for
- `string2` – Mandatory string argument representing the string being searched
- `startIndex` – Optional integer argument identifying the start point for the search

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string first <string1> <string2> <start-index>?]`’ and convert these to the ‘string last’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.8 string length

Command will be implemented as a MOIS function, returning an Integer value and taking arguments as follows :-

- `string` – Mandatory string argument for which the length is required

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string length <string>]`’ and convert these to the ‘string length’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.9 string map

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `nocase` – Optional switch argument with valid value ‘-nocase’
- `charMap` – Mandatory string argument identifying the key value pairs for the replacement
- `string` – Mandatory string argument identifying string to be remapped

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string map <nocase>? <char-map> <string>]`' and convert these to the 'string map' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.10 string match

Command will be implemented as a MOIS function, returning a Boolean value and taking arguments as follows :-

- `nocase` – Optional switch argument with valid value '-nocase'
- `pattern` – Mandatory string argument representing the match pattern
- `string` – Mandatory string argument identifying string to be matched

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string match <nocase>? <pattern> <string>]`' and convert these to the 'string match' MOIS function.

b. The converter shall create a MOIS local variable of Boolean type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Boolean type.

2.5.50.11 string range

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `first` – Mandatory integer argument identifying the first character of the target range
- `last` – Mandatory integer argument identifying the last character of the target range

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string range <string> <first> <last>]`' and convert these to the 'string range' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.12 string repeat

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `count` – Mandatory integer argument identifying the number of iterations

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string repeat <string> <count>]`' and convert these to the 'string repeat' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.13 string replace

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `first` – Mandatory integer argument identifying position of the first character to be replaced
- `last` – Mandatory integer argument identifying the position of the last character to be replaced
- `newString` – Optional string argument identifying the replacement string

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string replace <string> <first> <last> <new-string>?]`' and convert these to the 'string replace' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.14 string tolower

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `first` – Optional integer argument identifying position of the first character to be case lowered
- `last` – Optional integer argument identifying the position of the last character to be case lowered

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string tolower <string> <first>? <last>?]`’ and convert these to the ‘string tolower’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.15 string totitle

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `first` – Optional integer argument identifying position of the first character to be case changed
- `last` – Optional integer argument identifying the position of the last character to be case changed

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [string totitle <string> <first>? <last>?]`’ and convert these to the ‘string totitle’ MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.16 string toupper

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string

- `first` – Optional integer argument identifying position of the first character to be case changed
- `last` – Optional integer argument identifying the position of the last character to be case changed

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string toupper <string> <first>? <last>?]`' and convert these to the 'string toupper' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.17 string trim

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `chars` – Optional string argument identifying the characters to be trimmed (defaults to whitespce characters).

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string trim <string> <chars>?]`' and convert these to the 'string trim' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.18 string trimleft

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `chars` – Optional string argument identifying the characters to be trimmed (defaults to whitespce characters).

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string trimleft <string> <chars>?]`' and convert these to the 'string trimleft' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.19 string trimright

Command will be implemented as a MOIS function, returning a String value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `chars` – Optional string argument identifying the characters to be trimmed (defaults to whitespce characters).

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string trimright <string> <chars>?]`' and convert these to the 'string trimright' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.50.20 string wordend

Command will be implemented as a MOIS function, returning a Integer value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `charIndex` – Mandatory string argument identifying word for which the end position is required

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string wordend <string> <char-index>]`' and convert these to the 'string wordend' MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.50.21 string wordstart

Command will be implemented as a MOIS function, returning a Integer value and taking arguments as follows :-

- `string` – Mandatory string argument identifying the source string
- `charIndex` – Mandatory string argument identifying word for which the start position is required

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [string wordstart <string> <char-index>]`' and convert these to the 'string wordstart' MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name <return-val> and this is not of the Integer type.

2.5.51 subst

`subst` `?-nobackslashes? ?-nocommands? ?-novariables?` *`string`*

This command performs variable substitutions, command substitutions, and backslash substitutions on its *string* argument and returns the fully-substituted result. The substitutions are performed in exactly the same way as for Tcl commands. As a result, the *string* argument is actually substituted twice, once by the Tcl parser in the usual fashion for Tcl commands, and again by the *subst* command. If any of the **-nobackslashe**s, **-nocommands**, or **-novariables** are specified, then the corresponding substitutions are not performed. For example, if **-nocommands** is specified, no command substitution is performed: open and close brackets are treated as ordinary characters with no special interpretation.

The command will be implemented as a MOIS function , returning a string value and taking the following arguments :-

- `nobackslashes` – Optional switch argument taking the value '-nobackslashes'
- `nocommands` – Optional switch argument taking the value '-nocommands'
- `novariables` – Optional switch argument taking the value '-novariables'
- `string` – Mandatory string argument containing the text to be substituted

The MOIS converter requirements are :-

a. The MOIS converter shall recognise compound Tcl statements of the form '`set <return-val> [subst <nobackslashes>? <nocommands>? <novariables>? <string>]`' and convert these to the 'subst' MOIS function.

b. The converter shall create a MOIS local variable of String type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the String type.

2.5.52 tell - Return current access position for an open channel

`tell` `channelId`

Returns an integer string giving the current access position in *channelI*d. This value returned is a byte offset that can be passed to **seek** in order to set the channel to a particular position. Note that this value is in terms of bytes, not characters like **read**. The value returned is -1 for channels that do not support seeking.

Command will be implemented as a MOIS function returning an integer value and taking the following arguments :-

- channelId – Mandatory string argument identifying the channel

The MOIS converter requirements are as follows :-

a. The MOIS converter shall recognise compound Tcl statements of the form ‘`set <return-val> [tell <channel-id>]`’ and convert these to the ‘tell’ MOIS function.

b. The converter shall create a MOIS local variable of Integer type with the name corresponding to `<return-val>`, provided a variable of that name does not already exist.

c. The converter shall raise an error and fail the statement conversion if a MOIS variable already exists with name `<return-val>` and this is not of the Integer type.

2.5.53 time - Time the execution of a script

`time` `script ?count?`

This command will call the Tcl interpreter *count* times to evaluate *script* (or once if *count* isn't specified). It will then return a string of the form ‘`503 microseconds per iteration`’ which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

This command will not be implemented in MOIS procedures.

2.5.54 trace - Monitor variable accesses

`trace` `option ?arg arg ...?`

This command causes Tcl commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented.

This command will not be implemented in MOIS procedures.

2.5.55 unknown - Handle attempts to use non-existent commands

`unknown` `cmdName ?arg arg ...?`

This command is invoked by the Tcl interpreter whenever a script tries to invoke a command that doesn't exist. The implementation of **unknown** isn't part of the Tcl core; instead, it is a library procedure defined by default when Tcl starts up. You can override the default **unknown** to change its functionality.

This command ewill not be implemented in MOIS procedures.

2.5.56 unset - Delete variables

```
unset name ?name name ...?
```

This command removes one or more variables. Each *name* is a variable name, specified in any of the ways acceptable to the **set** command. If a *name* refers to an element of an array then that element is removed without affecting the rest of the array. If a *name* consists of an array name with no parenthesized index, then the entire array is deleted. The **unset** command returns an empty string as result. An error occurs if any of the variables doesn't exist, and any variables after the non-existent one are not deleted.

This command will not be implemented in MOIS procedures.

2.5.57 upvar - Create link to variable in a different stack frame

```
upvar ?level? otherVar myVar ?otherVar myVar ...?
```

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *Level* may have any of the forms permitted for the **uplevel** command, and may be omitted if the first letter of the first *otherVar* isn't # or a digit (it defaults to 1). For each *otherVar* argument, **upvar** makes the variable by that name in the procedure frame given by *level* (or at global level, if *level* is #0) accessible in the current procedure by the name given in the corresponding *myVar* argument. The variable named by *otherVar* need not exist at the time of the call; it will be created the first time *myVar* is referenced, just like an ordinary variable. There must not exist a variable by the name *myVar* at the time **upvar** is invoked. *MyVar* is always treated as the name of a variable, not an array element. Even if the name looks like an array element, such as **a(b)**, a regular variable is created. *OtherVar* may refer to a scalar variable, an array, or an array element.

This command will not be implemented in MOIS procedures.

2.5.58 variable - create and initialise a namespace variable

```
variable ?name value...? name ?value?
```

This command is normally used within a **namespace eval** command to create one or more variables within a namespace. Each variable *name* is initialised with *valu*e. The *value* for the last variable is optional.

The structure of this command does not easily fit into the generic function / directive definition, as it uses multiple repeated arguments which are dependent on each other. This command will not be implemented in MOIS procedures.

2.6 Generic Requirements

This section identifies any generic requirements

a. Tcl switch values may be abbreviated, provided the value is unique within the value list for the switch argument. The MOIS converter shall not support such abbreviations, if such are found they shall be treated as an unmatched argument value and the statement conversion failed with an error.

b. In all cases a failure to match a Tcl statement shall result in the statement being translated as a MOIS CMT (comment) statement. Failure to match may either be as a result of a specific requirement or as a result of no requirements existing for a particular structure (or structure variant).

c. In all cases where requirements call for failure of statement conversion and error notification, this shall apply only to the current statement. Conversion for subsequent statements shall be performed normally.

d. Conversion errors shall be reported in the converter interface and also in the translated MOIS procedure as a CMT (comment) statement immediately following the failed statement (which will also have been translated as a CMT).

2.6.1 Functions & Directives

2.6.1.1 Arguments

a. Unless otherwise stated in the requirements for a specific item, the MOIS converter shall identify function / directive arguments in the original Tcl as either literal values or as simple variable references (i.e. of the form '$<var-name>').

b. For variable references, the converter shall check that a MOIS variable with name <var-name> exists and is of the appropriate type for the argument. The statement conversion shall fail and an error be raised if a variable of the correct name and type does not exist.

In all cases in which an argument represents a variable name, array references are not allowed and shall cause the converter to fail the statement conversion and raise an error.

3 ASSUMPTIONS

3.1 Untranslated Statements

In the requirements, it is assumed that any statements that are not translated are inserted into the MOIS Free-text directive. This is the case for all statements which will not be implemented in MOIS procedures.

This Free-text directive will take as argument a string, which will be a part of script couldn't be implemented in MOIS procedure. Such statements will be treated just as a directive statement by MOIS, but could be exported as Tcl/TOPE code.

However, if an error occurred and a statement couldn't be translated, a MOIS comment statement is added instead with the error report.

3.2 Conditions

In the condition section the requirements are based on the existing MOIS condition definitions.

3.3 Function / Directive Definitions

Many of the Tcl/TOPE statements have argument definitions that conflict with the current MOIS function / directive definitions.

Problem areas, with examples are :-

- Repeated command argument

  In this case the final command argument can be repeated an arbitrary number of times. This can be modelled in the directive / function mechanism to some extend by defining a fixed number of optional arguments and then applying a coding rule to limit the number of allowed repeats. Examples are :-

  ```
  verified ?-timestamp <time>? -tc <cmd> ?<param>...?
  ```

- Multiple command forms

  In this case the command has several forms (i.e. argument patterns) depending on a command qualifier.

  ```
  verified ?-timestamp <time>? -tc <cmd> ?<param>...?
  ```

  ```
  verified ?-timestamp <time>? -tm <parameter>
  ```

  ```
  binary format formatString ?arg arg ...?
  binary scan string formatString ?varName varName ...?
  ```

  Each command formulation will be defined as a directive. The reverse tool will recognise it and reverve it to the correct directive.

- TM name as argument

  In this case, a command uses a TM name as an argument. In fact this may be already implemented as there is a TELEMETRY value type in the DIR_ARGUMENTS table of the common directives DB. Some of the TOPE commands have arguments which are TM names - clearly it makes sense for MOIS to be able to validate such arguments to ensure the TM is defined in the s/c DB.

  ```
  patchlocation <param-name> <spid> <byteoffset>
  <bitoffset>
  ```

  ```
  setparameter ?-raw? <param-name> <value>
  ```

- List arguments

  In this case a command argument can be a list of items. In Tcl these would be written as a space separated list inside braces. If only one item is in the list, the item could be written inside braces or without the braces. For example, a list of TM parameters could be written `'{W123 X456 Y987 Z765}'`. These are not necessarily a problem - the converter can just assign the whole argument string.excluded braces to the argument on conversion. Then the directive/function will define delimiter for this function (braces), which will be added during the export to Tcl/TOPE.

  ```
  enableparam <param-list>
  ```

  ```
  inhibitparam <param-list>
  ```

- Functions/Directives with several signatures

  Most of Tcl/TOPE commands will be implemented as functions or directives and these commands could have several signatures. Then the idea to support that would be to define one function/directive per signature. However found the correct associated function in CommonDirectiveDb database will raise problem if no naming convention is defined. The example of verified Tcl/TOPE commands is relevant:-

  Two signatures:

  - `verified ?-timestamp <time>? -tc` *`<cmd> ?<param>...?`*
  - `verified ?-timestamp <time>? -tm` *`<parameter>`*

  So two directives will be defined with the following relevant name verifiedtc and verifiedtm. How to know that the first signature correspond to the verifiedtc directives and the second one to the verifiedtm directives. Indeed, to avoid any hardcoding of function ID in reverse tool, a naming convention needs to be defined.

4 CODING STANDARDS

It is intended to convert Tcl/TOPE procedures into MOIS procedures for use in flight. MOIS provides extensive configuration control, validation & formatting features for managing the procedure set. The procedures will be managed within MOIS, but for execution will be exported into Tcl/TOPE procedures to be executed by the control system.

MOIS will provide a converter program to convert legacy Tcl/TOPE procedures (e.g. from the AIT program) into MOIS procedures to act as a basis for the MOIS procedure set. However, MOIS procedures do not have the same richness of expression and flexibility as the Tcl/TOPE . In addition there is a cost/function trade off to be made in the converter which also limits the translation.

These coding standards are intended to describe the restrictions to be placed on Tcl/TOPE procedures in order that they may be translated to MOIS procedures. In order to understand the coding restrictions, it is necessary to understand (in outline) how the translation is performed.

- Firstly the program structure constructs are analysed (conditions, loops, etc) and mapped to equivalent MOIS structures.
- Remaining code blocks are then split into individual Tcl/TOPE statements.
- Group of statement is checked to see if it can be converted into a specific group of statements or any of the specific types (e.g.telemetry, command, pause, control,etc.).
- Each statement is then checked to see if it can be converted into any of the specific MOIS statements types (e.g. telemetry, command, pause, control, etc). Statements that can be translated are added into the MOIS procedure at the appropriate position.
- Untranslated statements are then checked to see if they can be converted into MOIS functions or directives (functions return a value, directives are similar but do not return a value). This is a generic method of converting Tcl/TOPE statements into MOIS in a structured way. A DB contains a list of Tcl/TOPE functions and defines the required arguments. A conversion is effected if the statement matches any of the function / directive names and argument set.
- Any statements still untranslated at this stage represent statements not defined in the directives DB or statements which could not be translated for other reasons (e.g. non-conformance to coding standards). The translator should convert these as a free instruction directive taking a string as parameter.
- If an error occurred during the reverse process for any of statements, then the statement will be implemented as a MOIS comment statement.

As a consequence of the above, the coding standards described here have several levels of applicability as follows :-

1. Rule is mandatory, failure to apply it may cause complete or partial failure to translate.
2. Rule is recommended, failure to apply will cause MOIS verifiable statements (e.g. TM or TC references) to be degraded into non-verifiable elements or less verifiable elements (functions or directives).
3. Rule is suggested, failure to apply will degrade functions and directives into non-verifiable elements statements.

4.1 Tcl Language Constraints

- Braces ({}) have not to be used to define quoted strings, they must be used only to designate code structure elements (such as the conditionally executed block in an IF clause). The MOIS converter needs to identify quoted strings and ignore the contents, obviously code blocks can't be ignored and therefore the converter assumes that {} do not represent literal string delimiters. Level 1.
- In program structure constructs (conditions, loops, etc), braces ({}) have always to be used to delimit elements, even where this is not strictly required by Tcl. E.g. use 'if {$cnd} {….}', but not 'if $cnd { … }'.This anyway makes the code easier to read and better allows the converter to identify the elements of the structure. Level 1.
- Command substitutions (expressions contained in [ ]) will be set in a MOIS local variable. Thus all expressions into square barckets will be set in a MOIS local variable and that will be applied recursively, this is described in the relevant example:-:

  `'addtime [getrawvalue $obt] [lindex [split $dif .] 0] [string trimleft [lindex [split $dif .] 1] 0]'.`

  `[getrawvalue $obt]` a variable will be set for the value.

  `[lindex [split $dif] 0]` a variable will be set for the value.

  `[split $dif]` a variable will be set for the value.

  `[string trimleft [lindex [split $dif ] 1] 0]` a variable will be set for the value.

  `[split $dif ]` a variable will be set.

  `[lindex [split $dif ] 1]` a variable will be set.

  Note the more the substitution will be done directly in the Tcl/TOPE code, easier will be the reverse process for the substitution. Then expression as described above should be avoided as often as possible

- Tests will be coded in modular fashion to allow the conversion of MOIS-Tcl and reverse to be achieved much quicker. An example is described in the relevant example:-

```
if { [file exists $env(HPCCSTESTENV)/TSEQ/Tools.tcl ] } {
     source $env(HPCCSTESTENV)/TSEQ/Tools.tcl
} else {
putlog "Can't source Tools.tcl"
exit
}
```

This is an example of a modular fashion to make the reverse process easiest.

4.2 Other Supported Tcl Statements

This section details the other Tcl statements that are specifically supported by MOIS. Any other Tcl statements used will be converted only as a free directive statement.

Most of Tcl statements will be implemented as a MOIS directives/functions in MOIS procedure. Then the way of directives/functions are managed in MOIS and their limits of implementation is explained in section 3.3.

The way that `foreach` control structure will be implemented in MOIS procedure will not be represented properly in the MOIS flowcharter since it will be implemented as a directive. Then Tcl/TOPE scripts should avoid the use of this control structure as possible. Then the MOIS flowcharter will keep a relevant representation of the Tcl/TOPE script. The same behaviour should be apply for `continue` or `break` Tcl statements since these two Tcl statements will be implemented as directives.

Note that several Tcl statements won't be implemented in MOIS procedure, it doesn't mean that it will not be taken into account. In fact these statement will be reversed as a MOIS Free text directives. To be reversed as a MOIS Free-Text directives, Tcl statements have to be consistent with their formulation. Any erro on the formalution will raise an error and the statement will be reversed to a MOIS Commen statement.

4.3 General TOPE Constraints

TOPE reverse implementation in MOIS procedure could be divided into two parts:

- Statements can be directly implemented as a MOIS statement (`waittime`).
- Statements sequence can be directly implemented as a MOIS statement or a sequence of MOIS statements.

4.3.1 TOPE statements directly reverse

This part deals with TOPE statements, which can be implemented as a MOIS statement.

Most of TOPE statements will be implemented as MOIS directives or functions statements. The TOPE statement must match the pattern defined in TOPE statements part to make sure it will be implemented properly.

As Tcl statements, if an error occurs, the TOPE statement will be implemented as a MOIS Comment statement.

4.3.2 TOPE statements structure reverse

Some directives should be implemented as directives or functions. However if they are included in a defined TOPE structure, they can be implemented in a specific MOIS statement.These TOPE structures are explaind in the relevant examples:-

- `‘tcsend T1 referby rT1`

```
waitfor sT1 -until
getcompleted $rT1’
```

This command sequence will be implemented to a MOIS TC statement followed by a CTL MOIS statement SEV waiting the end of execution.

- `‘subscribepacket <pktid> referby <varname>`

```
waitfor -timeout <time> <varname>
unsubscribepacket <pktid>’.
```

This will be equivalent to a Wait for Packet statement with a timeout. Then only this structure could be equivalent to a wait for packet in MOIS procedure.

- `‘if {getrawvalue[fetch <param>]} {… exit}’`

```
‘if {getengvalue[fetch <param>]} {… exit}’
```

These two pattern will be implemented in MOIS procedure as a PERFORM step including a MOIS verify TLM statement.

All patterns described above must have the defined formulation to make sure that during the reverse process , the MOIS statements will appear.

4.4 Individual restrictions

- Lists and arrays handling shall be avoided in Tcl/TOPE script. Indeed MOIS can't manage a such data.
- File directories and channel handling: Set up of the specific files and channels should be in a configuration file, as this may not necessary to be replicated in MOIS. Then these instructions should not be used in normal procedures.
- System instructions handling should not be coded in normal procedures such as CD command. This also applies to some of the instructions included for specifically debugging the TCL itself (MEMORY, GLOB etc…)