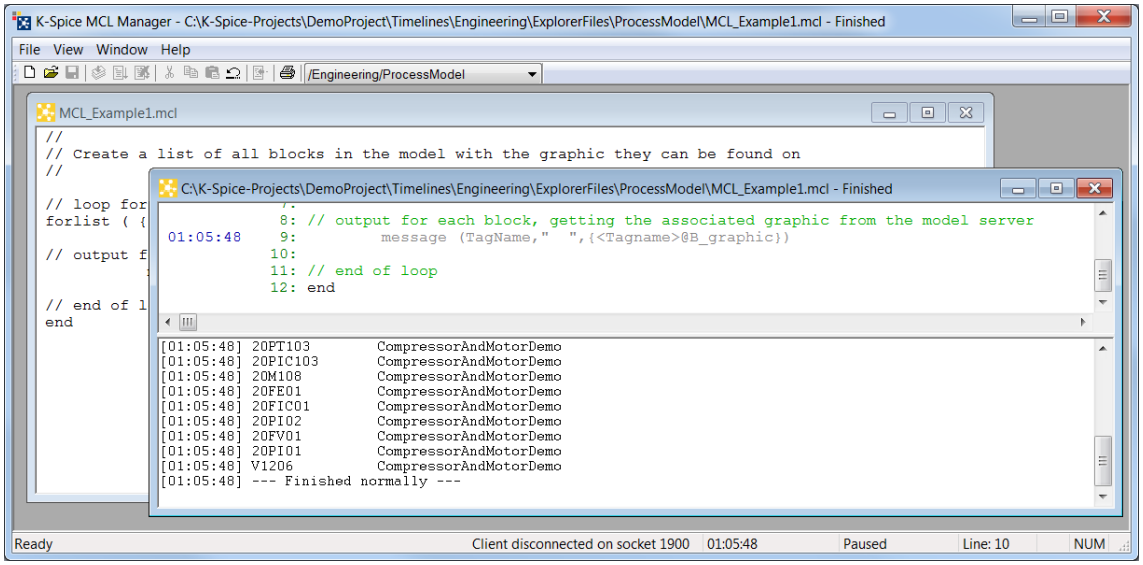
KONGSBERG

KONGSBERG K-Spice® Model Control Language

Reference Guide

Release 3.2



This document describes the K-Spice® language extension used to control and run models non-interactively, and to set up scenarios for configurable model runs.

366790/G

Document history

Document number: 366790		
Rev. A	22 February 2011	Updated earlier document.
Rev. B	1 March 2011	First XML version.
Rev. C	20 April 2011	Added simulator control functions, rearranged appendices
Rev. D	8 December 2011	Reviewed and updated dialogs to Windows 7.
Rev. E	26 April 2012	Clarify command line options. Add remaining reserved keywords. Add Stepped Run commands. Minor edits.
Rev. F	December 2012	Update for release 2.4
Rev. G	September 2014	Minor corrections, version update.
Rev. H	October 2015	Minor update for release 3.2

The reader

This publication is intended as a guide for the K-Spice® user. The guide is based on the assumption that the user is familiar with process modelling, as well as oil and gas production systems.

Note

Comments

To assist us in making improvements to the product and to this guide, we welcome comments and constructive criticism.

e-mail: kogt.documentation@kongsberg.com

For more information on K-Spice®, visit our website: www.kongsberg.com/K-Spice

Kongsberg Oil & Gas Technologies AS
www.kongsberg.com/kogt

Table of contents

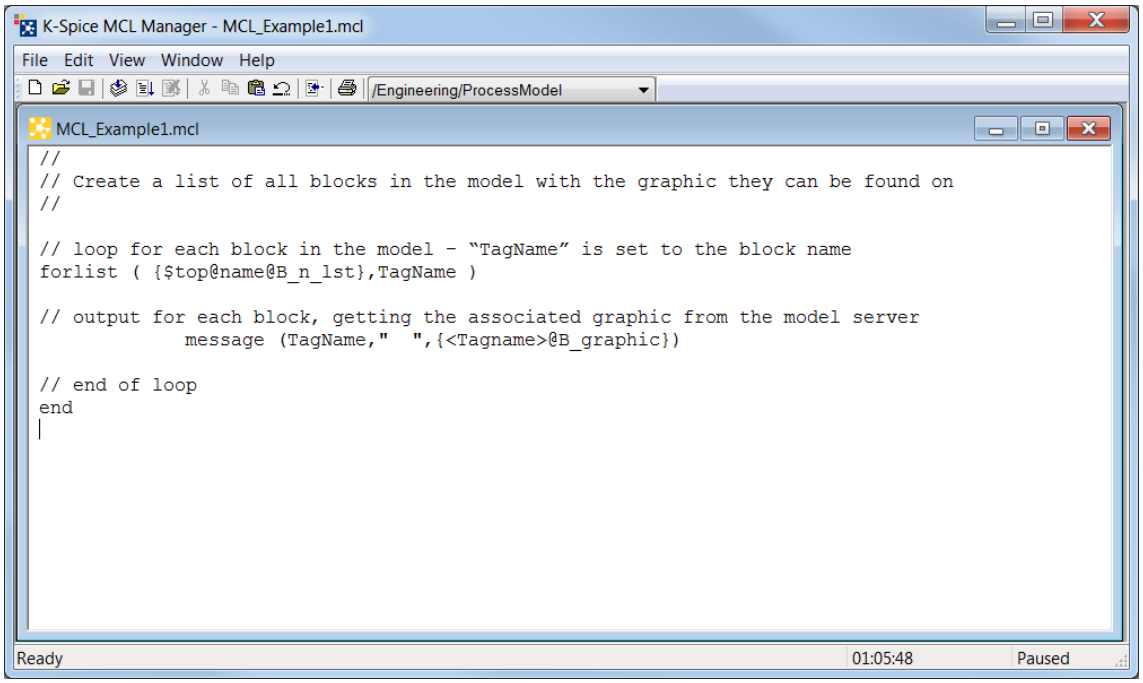
1 MCL Overview

Overview

The K-Spice® Model Control Language (MCL) has been designed as a means of controlling a K-Spice® model using a procedural style control language. An MCL program allows the user to access and modify model data items and to control the loading and execution of a model. Model control tasks can be performed by executing a series of instructions at given times or when a pre-defined model event happens.

An MCL program can also be used as a script to perform common tasks such as loading a specific combination of model, parameter and value files.

Figure 1 MCL Manager – Scenario configuration editor



The screen shown above is the fully-featured *MCL Manager*, an interactive environment for creating and executing MCL scripts.

The *MCL Manager* application has two types of sub-windows:

The editor window – This window contains the code for the Scenario script.

The monitor window – This window shows the execution progress of the Scenario.

1.1 Running an MCL program

There are three ways an MCL program can be run:

- from the command line.
- from K-Spice® *SimExplorer*.
- from K-Spice® *MCL Manager*.

Running MCL from the command line

When run from the command line, the syntax is:

```
mcl <command line options> source_name[.mcl]
```

Where `source_name.mcl` is the full path to the MCL script file, or just the mcl file name if it is in the current directory.

Command line options:

The `-parse` parameter can be used to parse, but not run the *MCL* file. This option can be used to debug programs without the overhead of setting up an actual model run to verify the source file. MCL files can also be parsed inside the *MCL Manager* application.

The `-log` *LogFileName* parameter is used to supply a log file name. All output from the *MCL* will be recorded in the log file.

The `-NetMasterhost` *SimulationManagerMachine* option instructs *MCL* to connect to a *SimulationManager* running on a different machine.

The `-Instance` *kExplorer:InstanceNumber* option instructs *MCL* to connect to the *SimExplorer* with a specific integer instance number.

The `-Mid` *ApplicationName* option instructs *MCL* to connect to a specific application.

The `-nomm` option allows the user to run an *MCL* script without logging on to *MCL Manager* when it is active.

The `-npStatus` option instructs *MCL* to notify *MCL Manager* of a failure when using the `–parse` option.

The `-rem` option removes (deletes) the MCL script file after it has been run.

Running MCL from the SimExplorer

When the K-Spice® *SimExplorer* is started with the **Engineering** application, MCL programs can be run using the **Control →Run MCL** option from the main menus. All messages that are generated by the MCL program using the message() procedure will be sent to the *SimExplorer* console.

Running MCL from the MCL Manager

The final method of running an MCL script is from the K-Spice® *MCL Manager* application. The MCL script will be connected to the application currently selected in the MCL Manager toolbar.

2 Language Reference

Overview

There are six classes of tokens used in the MCL: identifiers, keywords, constants, strings, operators and other separators. Blanks and tab characters (collectively known as "white space") are ignored in most cases. Some white space is required to separate any adjacent identifiers, keywords or constants.

2.1 Lexical conventions

Comments can be included within the body of a program by using: // then (desired characters) in a line. All characters after the // symbol are treated as a comment:

```
// Set up a ramp at 20 seconds

at (20) ramp({E100:Output}, 120, 60, 10)
```

2.1.1 Identifiers

An identifier is split into two classes: local variables and model data items. Local variables consist of a sequence of letters and digits: the first character must be a letter (the underscore character counts as a letter). Upper and lower case characters are treated as equal. Examples are:

```
i
pos4
current_value
```

Model data items are enclosed in curly braces ({}). The characters within the braces constitute any valid model data item name. Examples are:

```
{FIC_100:Output}
{V2045:InletStream[5].p}
```

The maximum length of local variable names is 32 characters (case is not significant). Model data item names have no maximum length and are not case sensitive.

2.1.2 Keywords

There are a number of keywords in MCL. These keywords are reserved and cannot be used as variable names. A full list of keywords is included in Appendix C

2.1.3 Constants

An integer constant is entered as any sequence of digits, optionally preceded with a unary plus or minus sign. All integer constants are treated as 32 bit values.

A real constant is entered as a sequence of digits with an embedded or terminating decimal point and/or an embedded e or E followed by an optionally signed integer exponent. A real constant can be preceded with a unary plus or minus sign. Real constants are treated as 64 bit double-precision values. Examples are:

```
10.0455;   100.;   -1.04023e-3
```

String constants are entered as any sequence of characters (except the return/carriage return character) enclosed with double-quotes. If a double-quote is required within a string constant, it is entered as \" . If a return/carriage return character is required, then it is entered as \n.

Boolean constants can have two values: *true* or *false*.

Time constants can be entered in the same way as an integer or real constant (the value is assumed to be in seconds) or the following format can be used:

```
[dd-mm-yyyy hh:mm:ss.cs c]
[dd/mm/yyyy hh:mm:ss.cs c]
```

d — Day as a decimal value or a day name. Day names are interpreted relative to the current date.

m — Month as a decimal value or month name. 3 letter abbreviations are acceptable.

y — Year as a decimal 2 or 4 digit value.

h — Hour as a decimal value between 0 and 23.

m — Minutes as a decimal value.

s — Seconds as a decimal value.

cs — Centiseconds as a decimal.

c — Single letter units code that is only applicable if the time contains a single decimal component. The values are:

```
H h  Hours [5h]
M m  Minutes [2.5m]
S s  Seconds [23.8s]
```

The date or time string must be enclosed by square brackets if it is anything other than a number of seconds.

Note that for a decimal point to be interpreted as a separator between seconds and centiseconds, a minutes component with a colon separator must be included in the time string. Whichever way a time constant is entered, it will be converted to a real value in seconds. Examples of time constants are:

`2` — 2 seconds

`2.3` — 2.3 seconds

`[2]` — 2 seconds

`[2:30]` — 2 minutes 30 seconds

`[4:30.50]` — 4 minutes 30 seconds and 50 centiseconds

`[01:10:30]` — 1 hour 10 minutes 30 seconds

`[today 16:0:0]` — Today at 4:00 PM

`[thursday 12:00]` — The next time it is Midday on a Thursday

`[02-Mar-12 09:00:00]` — 9 o'clock on the second of March 2012

2.2 Variables

Any identifier used within the program body represents either a local or model data item. Variables can be used on the left of any assignment statement or within expressions and procedure argument lists.

2.2.1 Local variables

Local variables are used within an MCL program to hold a value that can be the result of an expression or the value of a constant. Unlike most programming languages, local variables in the MCL are not explicitly typed, but are declared in context when used on the left side of an expression. For example, see the following statements:

```
tmp1 = 2
tmp2 = 13.56
tmp3 = "test.mdl"
tmp4 = true
```

MCL assigns storage for, and declares local variables with the types integer, real, string and Boolean as appropriate. If the right side of an assignment is a single local or model data item, then the local variable on the left side takes on the type of the referenced variable. If the right side of an assignment is an expression, then the local variable is automatically declared as type *real*.

2.2.2 Model data items

Model data items are differentiated from local variables in that they are enclosed within curly braces. The use of model data item identifiers in any statement enables the current value of a model data item to be fetched or a model data item to be set to a new value. Model data items can be used in the same place as any local variable. For example, the following statement:

```
{FIC_101:Output} = 23.4
```

will set "FIC_101:ot" in the current model to the constant value 23.4. Alternatively,

```
tmp = {E1002:flow} + 1.255e5
```

uses the current value of "E1002:flow" within an expression to set the local variable "tmp".

Unlike local variables, the value of a model data item can be an ever-changing quantity because the model runs in its own process independently of the executing MCL program. Therefore, whenever a model data item references a value in a statement, the current value of the model data item at that point in time is used. To save the current value of a model data item for use in subsequent statements, the value should be assigned to a local temporary variable which is then used in place of the model data item name:

```
if ({TI_100:Temperature} > 35)
 tmpv = {TI_100:Temperature}
 message("TI_100:Temperature has exceeded safe value: ", tmpv)
 if (tmpv > 40) {TI_100:Temperature} = 40
end
```

The model data items used in an MCL program can be any variable known to the model and are mainly variables that are associated with blocks within the model. However, some variables that are used to set model parameters use the *SetModelParameter* instruction. These variables must be set using the send procedure, not an assignment statement. If a fetch instruction such as *GetModelParameter* is required to acquire a value for a model data item, then the *obtain* procedure should be used to get the variable value and assign it to a local variable:

```
obtain (tmp,”GetModelParameter;”,"ModelSpeed",”/Timeline”)
```

Sometimes it is necessary to access or set model data items that use model id (MID) other than default MID or the MID supplied using the -mid option at startup. If this facility is required, then the model data item name can be prefixed with an MID and a "|" separator to indicate which MID the get/set operation should be directed to:

```
{/LAM/LAMApp|PIC101:SetPoint} = 100
x = {/Engineering/ProcessModel|PIC101:SetPoint}
```

The type of any model data item used in the MCL is known at run-time, so any conversion that is required will be carried out.

A local variable can be used to create" model data item names using angle brackets, so the command above for calculating "x" could also be written as follows:

```
MODELID = "/Engineering/processmodel"
BLOCK = "PIC101"
DATAITEM = "SetPoint"
x = {<MODELID>|<BLOCK>:<DATAITEM>}
```

2.2.3 Fixed variables

In addition to local and model data items, there are two fixed system-controlled variables named *time* and *status*. The variable *time* is used obtain the value of the current model time in seconds. The variable *status* will return an integer value indicating the current status of the model. The values are:

0 – Model not loaded

1 – Model loaded and running

2 – Model loaded - execution frozen

The values of *time* and *status* are for the current timeline.

It is also possible to obtain the model time and status for a particular timeline using the *ModelTime* and *ModelStatus* functions as follows:

ModelTime (TimeVariable[,Timeline])

- *ModelTime (MyTime)* would set the variable *MyTime* to the time of the currently active timeline.
- *ModelTime (EngTime,"/Engineering")* would set the variable *EngTime* to the time of the *Engineering* timeline.

ModelStatus (StatusVariable[,Timeline])

2.3 Expressions

Expressions can be used on the right side of an assignment operator, in a conditional statement or in the argument list of a procedure call. The expression can be a single constant or variable or any nested combination of operators that resolves to a single value. The result of an expression is generally a real value, but if the expression contains just a single value or a single value with an unary operator, then the result of the expression is the type of its value.

2.3.1 The assignment operator

The assignment operator (=) is used to assign the result of an expression to a variable:

```
variable = expression
```

The left side of the assignment must be an identifier representing a local or model data item. In the case of a model data item, the execution of an assignment involves sending an instruction to the model to set the specified variable with the expression value. As this is an asynchronous event, referencing the same model data item after the assignment may not always yield the expected value due to delays in event processing by the model. If this type of scenario is required, then the expression should be assigned to a temporary local variable first:

```
tmp = cval + 23.4 / {V1002:Pressure}
{E1002:Input} = tmp
{T1002:x} = tmp
Description = "V1002 = " + {V1002@B_desc}
```

2.3.2 Unary and binary operators

The following operators can be used in any expression:

```
+val   Unary plus
-val   Unary minus
val1 + val2  Addition
val1 - val2  Subtraction
val1 * val2  Multiplication
val1 / val2  Division
val1 ^ val2  Raise to the power
```

The binary operators require that the right side of the operator contains an integer or real constant or variable or an expression. String constants or variables can be used on the left or right of any binary operator, but the string will be converted to a real value before the expression is evaluated.

The result of any of these operators is a real value. The result of a division will be truncated if the right side of the operator is an integer constant or variable.

2.3.3 The operator rules for precedence

The table below summarises the rules for precedence of all operators including the logical operators discussed later on. Operators on the same line have the same precedence. Operators of equal precedence are evaluated from left to right:

Table 1 Operator rules for precedence

Operators
( )
^
* /
+ -
== != > >= < <=
&& ||
=

Where the order of evaluation needs to be forced, parenthesis can be used to isolate sub-expressions that will be evaluated first. For example:

```
(a + b) / (c + d)
```

ensures that the additions are evaluated before the division.

2.3.4 Logical operators

Logical operators compare the left and right side values of an expression to produce a result that is either *true* or *false*. An expression that is *true* results in a real value of 1.0, while an expression that is *false* results in a value of zero. The logical operators are:

```
val1 == val2 True if val1 is equal to val2
val1 != val2 True if val1 is not equal to val2
val1 > val2  True if val1 is greater than val2
val1 >= val2 True if val1 is greater than or equal to val2
val1 < val2  True if val1 is less than val2
val1 <= val2 True if val1 is less than or equal to val2
val1 && val2 True if val1 is true and val2 is true
val1 || val2 True if val1 is true or val2 is true
```

2.3.5 Inline functions

Any expression can use inline functions that return a value used in evaluating the expression. Inline expressions use the format:

```
func(arg1, arg2, ...)
```

where *func* is the name of the function and *arg1* etc. are values (or expressions) used as input to the function.

The following example uses the *int* function which returns the truncated value of its input:

```
ival = int({FIC100:ControllerOutput} / 3.5)
```

MCL supports the following mathematical functions from the standard C/C++ function set:

```
real abs(real x)
real acos(real x)
real asin(real x)
real atan(real x)
real atan2(real x, real y)
real cos(real x)
real cosh(real x)
real exp(real x)
real hypot(real x, real y)
real int(real x)
real log(real x)
real log10(real x)
real mod(real x, real y)
real pow(real x, real y)
real rand(real x)
real sin(real x)
real sinh(real x)
real sqrt(real x)
real tan(real x)
real tanh(real x)
```

Example

```
intTemperature = int(Temperature)
result = sqrt (23·5)
```

Note

All trigonometric functions are in Radians.

and the *getenv* function:

```
char getenv(char environment)
```

Example

```
myPath = getenv("Path")
```

For more information on these functions, refer to any C/C++ documentation.

2.4 Control flow

To aid the control flow of an MCL program, a number of statements can be used to create conditional blocks and loops. Any conditional statements rely on evaluating an expression and executing the conditional block if the expression is ‘true’ (non-zero). To supplement the operators that have already been described, a further set of logical operators are available.

2.4.1 The if statement

The if statement takes the form:

```
if ( expression )
       statement block
elseif ( expression )
       statement block
else
       statement block
end
```

The statement blocks will only be executed if the expression within a parenthesis is true - except for the case of *else*, where the statement block will be executed if all of the preceding *if* and *elseif* expressions are false. Any number of *elseif* statements can be included after the initial *if* and before the terminating end statement. An *else* block, if included, must appear after the last *elseif* and before the *end* statement. At the end of any statement block that gets executed, the program flow continues immediately after the terminating *end* statement.

The statement block can be made up of one or more statements, but an *end* statement must be used to indicate the end of the block. A short version can be used in cases where there is only one statement in the statement block:

```
if ( expression ) statement
elseif ( expression ) statement
else statement
```

When the short version is used, the end statement is not required. For example, the following two program segments are equivalent:

```
if ({TIC_100:Temperature} < 10)
     {E100:mode} = -1
elseif ({TIC_100:Temperature} > 35)
     {E100:mode} = 1
else
     {E100:mode} = 0
end

if ({TIC_100:Temperature} < 10) {E100:mode} = -1
elseif ({TIC_100:Temperature} > 35) {E100:mode} = 1
else {E100:mode} = 0
```

All conditional statements can use the short format for cases where only a single statement is to be executed. This is only valid if the single statement is an assignment, procedure call or an unconditional statement such as *break* or *goto*. Any other type of statement such as *while* or *for* will cause a compile time error.

2.4.2 The while statement

The *while* statement can be used to implement program loops based upon the result of an expression. The format of the statement is:

```
while ( expression )
      statement block
```

```
end
```

or in short form if there is only one statement in the statement block:

```
while ( expression ) statement
```

Upon entry to a *while* loop, the expression is evaluated and the statement block is executed if the result is true. At the end of the block, flow control returns to the *while* statement and the expression is tested once again. In this fashion, the statement block is executed repeatedly until the result of the expression is false. When the expression is false, the program flow is continued at the statement immediately following the *while* loop.

The following example uses a *while* loop to increment a model data item:

```
while ({E100:Output2} < 1.34e-5)
 {E100:Input3} = {E100:Input3} + 0.1
 wait(2)
end
```

2.4.3 The for statement

The *for* statement is used to implement loops with a fixed number of increments. The format of the statement is:

```
for ( expression )
        statement block
end
```

or in short form, if there is only one statement in the statement block:

```
for ( expression ) statement
```

Once entering a *for* loop, the expression is evaluated and the integer result is used to set the number of iterations of the loop. If the expression result is less than or equal to zero, the statement block is not executed. When the statement block has been executed the specified number of times, the program flow is continued at the statement immediately following the *for* loop.

The following example uses a *for* loop to generate a non-linear ramp:

```
eval = 1
for (10)
      {N0231:Input4} = 4.0 + eval^2
      eval = eval + 0.5
      wait(1)
end
```

2.4.4 The forlist statement

The *forlist* statement is used to enumerate a model data item whose type is an array. The format of the statement is:

```
forlist ( array [, element] )
```

```
    statement block
end
```

On entry to the *forlist* loop, the model is queried for the value of the variable array. The loop will then iterate for each element of the array. During each iteration, the value of the array element will be placed in the variable named "list" unless an optional variable name element is used in the *forlist* construction. In this case, the variable element will contain the array element value for each iteration.

2.4.5 The every statement

The *every* statement is used to implement loops that execute at a fixed time increment. The format of the statement is:

```
every ( expression )
      statement block
end
```

or in short form, if there is only one statement in the statement block:

```
every ( expression ) statement
```

Upon entry to an *every* loop, the expression is evaluated and the real result is stored and used as a model time execution increment. The current model time for the first execution of the *every* loop is also stored as a base time for calculating further increments. The statement block associated with the loop is executed immediately on the first iteration. The program then pauses and re-executes the statement block once the execution increment time has passed. The only way to stop the loop incrementing is to add a conditional *break* statement to break out of the loop.

This method of using a time-based loop is preferable to using *wait* statements in a *while* or *for* loop as it will be far more accurate over a long time span and will not accumulate timing errors.

The following example uses an *every* loop to increment and test a variable every minute:

```
every (1:0)
      {E0201:HeatIn} = {E0201:HeatIn} + 0.00142
      if ( {E0201:OutletTemperature} > 70.5 ) break
end
```

2.4.6 The break statement

The *break* statement is used to break out of the current *while* or *for* loop - thus the statement is only valid when used in a statement block within these loops. When the *break* statement is encountered, program flow will continue immediately after the end of the statement block. The following example uses a *break* to terminate a *for* loop prematurely:

```
for( n )
```

```
        if ({V200:Output3} > 1.0) break
        {V200:Input1} = {V200:Output2} + 4
end
```

2.4.7 The stop statement

The *stop* statement is used to halt the execution of the current program thread - any threads that are still stacked will remain on the stack until they are ready for execution or the *abort* statement is used.

2.4.8 The abort statement

The *abort* statement is used to halt the execution of the current program thread and to terminate the MCL process itself. If any stacked threads exists, these will not be executed.

2.4.9 The goto statement

The *goto* statement is used to add an unconditional jump to a program. The keyword is followed by an identifier which must match up to a corresponding label statement placed before or after the *goto* statement. The *label* statement consists of the *label* keyword followed by an identifier. The following example uses *goto* to break out of a nested *while* loop:

```
while({CV_100:Input2} < 0.4352)
      while({CV_100:Setpoint1} > 0.0122)
           if ({CV_100:InletStream.t} >= 1.0) goto done
      end
end
label done
```

Although a *goto*/*label* pair can be placed anywhere in the program body, a *goto* that branches outside of an *at* or *when* thread, or into a different conditional or loop block will result in a compilation error.

2.5 Threads

Threads are blocks of statements that can be ‘stacked’ and executed when the result of an expression is *true* or when a specific model time is reached. The main body of an MCL program is considered to be one thread, while any *at* or *when* statement introduces the start of a new thread. When an executing program encounters an *at* or *when* statement, the entire statement block is stacked and execution continues immediately after the end of that statement block. When the current thread has finished executing, periodic checks are made on all stacked threads to see if any of their starting conditions are

met. If this is the case, the statement block associated with the thread is executed and 'unstacked'. Any remaining threads are checked and executed until there are no more stacked threads, at which point the program is finished.

Because the testing of thread start conditions is only carried out in the gap between executing threads, threads cannot run simultaneously. If simultaneous execution of *MCL* programs is required, then the *execute* procedure should be used to start an additional *MCL* program in its own process.

2.5.1 The at statement

The *at* statement is used to execute a thread at a specified model time. The format of the statement is:

```
at ( expression )
       statement block
end
```

or in short form, if there is only one statement in the statement block:

```
at ( expression ) statement
```

The expression within the parenthesis is evaluated to a real result which is then treated as a model time in seconds. When the actual model time matches the specified time, the statement block is executed. The current program thread will continue immediately after the end of the statement block. The following example uses the *at* statement to set up a thread that will be executed at the current time plus 2 minutes:

```
at (time + [2.0])
      {FIC_1020:mode} = 1
      {V_1020:Input4} = 0.002
end
```

When an MCL program is first started, the initial value of the model time is saved. Any *at* threads that should be executed at a time that is less than this initial value will be discarded. This is done so that any MCL programs that use *at* threads to carry out a sequence of actions can be run at different startup states of the model. i.e., actions that have already been carried out will not be executed twice.

2.5.2 The when statement

The *when* statement is used to execute a thread when the result of an expression is true. The format of the statement is:

```
when ( expression )
      statement block
end
```

or in short form, if there is only one statement in the statement block:

```
when (  expression  )  statement
```

Once the main thread execution is complete, the expression within the parenthesis is evaluated continuously. When the result of the expression is *true* (non-zero), the statement block is executed. The current program thread will continue immediately after the end of the statement block. The following example uses a conditional thread to set up a ramp:

```
when ({RA_FIC:Output} > 0.0425 && {RA_FIC:measure} > 9.5)
      {RA_FIC:mode} = 1
      ramp({RA_FIC:Output}, 0.2, 20, 1)
      wait(60)
      Send ("ModelFreeze;", "/Timeline")

end
```

2.6 Procedures

Procedures are built-in functions that are supported by the MCL. They take the form of a keyword followed by a fixed or variable number of comma-separated arguments within parenthesis. The following are examples of procedure calls:

```
wait(2)
ramp({E100:Input3}, 1.0+{V233:v2}, 100, 2)
message("The value of N001:Stream[2].p is: ", {N001:Stream[2].p})
```

The arguments supplied for a procedure can be any valid constant or expression except for cases where the procedure changes the value of one of its arguments – in this case the argument must be a model or local variable.

2.6.1 History functions

Syntax

```
DisableHistory ("DataItem"[,MID])
EnableHistory ("DataItem"[,MID,Tolerance])
HistorySet (LocalVariable,"DataItem"[,MID])
```

Description

The *history setting* procedures are used to managed historised variables using *MCL* script. History can also be enabled/disabled from the trend label tick boxes in *K-Spice SimExplorer*.

DisableHistory

The *DisableHistory* procedure turns history off for the specified data item. By default the current application will be used. The optional *MID* parameter can be used to disable history for a non-default application. Examples of the *DisableHistory* procedure:

```
DisableHistory ("20PIC103:InternalSetpoint")
DisableHistory ("20PIC103:InternalSetpoint","/Engineering/ProcessModel")
```

EnableHistory

The *EnableHistory* procedure turns history on for the specified data item. By default the current application will be used. The optional *MID* parameter can be used to enable history for a non-default application. The optional *Tolerance* parameter can be used to override the default tolerance for changes in value to be recorded. Examples of the *EnableHistory* procedure:

```
EnableHistory ("20PIC103:Measurement")
EnableHistory ("20PIC103:InternalSetpoint","/Engineering/ProcessModel")
EnableHistory ("20PIC103:Measurement","/Engineering/ProcessModel",0.01)
EnableHistory ("20PIC103:InternalSetpoint",,0.005)
```

HistorySet

The *HistorySet* procedure tests if the specified data item is currently historised. The optional *MID* parameter can be used to test history for a non-default application. The local variable is set to *true* if history is enabled and *false* if not. Examples of the *HistorySet* procedure:

```
HistorySet (IsHistorSet,"20PIC103:InternalSetpoint")
Message ("20PIC103:InternalSetpoint history is",IsHistorSet)

HistorySet (HistoryStatus,"20PIC103:Measurement","/Engineering/ProcessModel")
if (HistoryStatus == false)
 EnableHistory ("20PIC103:Measurement")
end
```

An example to enable history on multiple blocks in a model is included in Appendix A.

2.6.2 The action procedure

Syntax

```
action("action string")
```

Description

The *action* procedure is used to send an action string to a connected *K-Spice SimExplorer*. This procedure can only be used if the MCL was started from *K-Spice SimExplorer*. The action string can be any valid *K-Spice SimExplorer* action. For example, the following code segment will bring up a trend dialog and print it:

```
// Bring up the trend dialog
action("action=set,TAG,CV123;")
action("action=panel,trend_def;")
// Print the dialog
action("action=printdisplay,trend_def:1;")
```

Refer to the *Action Reference* documentation for a full description of all *K-Spice SimExplorer* actions.

2.6.3 The console procedure

Syntax

```
console(rows, columns)
```

Description

The *console* procedure is used to generate a small ‘terminal’ window that interacts with the *get* and *message* procedures. Normally these two procedures use the same terminal window that was used to start the MCL program. The *console* procedure can be used if an independent IO terminal window is required. The console window will disappear when the MCL program is finished or aborted.

The size of the console window is determined by the *rows* and *columns* arguments, which sets the number of rows and columns displayed in the window.

2.6.4 The execute procedure

Syntax

```
execute(mcl_source ,mid)
```

Description

The *execute* procedure is used to start a separate MCL program from the current program. This will start a new MCL process that communicates on the same port and host as the current MCL program. The argument: mcl_source specifies the name of the MCL source file to run. The optional second argument *mid* is the application to connect to, for example, “/Engineering/ProcessModel”.

Because MCL program threads cannot run simultaneously, *execute* can be used in cases where simultaneous control is required because the executed MCL program runs in its own process. It should be noted that there will be a time delay between calling *execute* and the new MCL program running. This is caused by the time taken to hook into the running model process and the time taken to parse and compile the script.

2.6.5 The get procedure

Syntax

```
get(var_name)
```

Description

The *get* procedure is used to obtain a value from the user. When the procedure is called, program execution will be suspended while the user types a value into the console window; the value must be terminated with a return/carriage return character before the program is restarted. The resulting value is placed into the variable *var_name*.

The following program segment uses *get* to obtain a positive value for the variable "iters":

```
while(1)
      message("Please enter the number of iterations: ")
```

```
        get(iters)
        if (iters > 0) break
    end
```

Note

The 'get' command will only work when the "console" is used.

2.6.6 The message procedure

Syntax

```
message(str1, str2, ...)
```

Description

Constants and variables are formatted and printed according to their type, while arguments entered as expressions are formatted and printed as real values. The following example prints the value of a model data item:

```
message("The current value of PI101:Value is: ", {PI101:Value})
```

To use a local MCL variable within a request to the model – use angle brackets around the variable name:

```
VAR = "PI101:Value"
message("The current value of PI101 is: ", {<VAR>})
```

2.6.7 The obtain procedure

Syntax

```
obtain(var, instruction "model_dataitem", [mid])
```

Example

```
obtain (speed, "GetModelParameter", "ModelSpeed", "/Engineering/Process")
```

Description

The *obtain* procedure is used to get the value of a model data item when a fetch instruction other than *GetVariable* is required. The first argument *var* will receive the value of the model data item "model_dataitem"; therefore these arguments must be entered as a variable name and string and not expressions of any sort. The argument *var* will take on the type of the returned value. The second argument is an instruction name such as *GetModelParameter*. This name must exists in the opcode database that is, by default, located here:

```
C:\Program Files\Kongsberg\K-Spice\Explorer\Resources\ExplorerData\opdb
```

The fourth optional argument is the model application to obtain the value from.

A call to *obtain* will block execution of the MCL until a value is received or a timeout is reached. The routine will fail quietly if the named variable does not exist in the model and *var* will be set to zero.

2.6.8 The ramp procedure

Syntax

```
ramp(var, to_value, range, increment)
```

Description

The *ramp* procedure is used to ramp a model data item to a set value over a specified time period. The name of the model data item is "var", while "to_value" specifies the value the procedure will ramp to; additionally, "range" indicates the total ramp time in seconds and "increment" specifies the incremental time in seconds for each step. The procedure returns and the following statements are executed. The ramp will carry on in the background until it is finished.

Note

To suspend execution until the ramp is finished, add a wait procedure call immediately after the ramp call.

A ramp will only be fully synchronised with the model if the RunStep, RunUntilStep or RunForStep commands are used to control the model execution.

The following example sets a variable to zero, then ramps it to 1.0 over 1 minute at 2 second intervals:

```
{FIC_100:Setpoint} = 0
ramp({FIC_100:Setpoint}, 1, 60, 2)
```

2.6.9 The send procedure

Syntax

```
send(opcode, arglist, ...)
```

Description

The *send* procedure is the main communication interface between the MCL and the model process. It is used to send any valid instruction code and associated data from the MCL to the model. This encompasses actions such as loading, starting and stopping the model; setting various model-related parameters; and issuing instructions to save model data and parameters.

The argument list consists of an instruction opcode followed by any data that must be sent with the instruction. The instruction opcode is a single identifier and must be one of the valid opcodes listed in the *opdb* file. The following example uses *send* to load and start a model:

```
Send ("SetModelParameter;","ModelName@F_load","/Timeline","ModelFileName")
Send ("SetModelParameter;","ParaName@F_load","/Timeline","ParameterFileName")
Send ("SetModelParameter;","InitName@F_load","/Timeline","ConditionFileName")
Send ("ModelRun;", "/Timeline")
```

Note

The use of ";" after the SetModelParameter option sends the command to the Simulation Manager, rather than the current application.

Typical instructions are:

Instruction	Format	Usage
SetModelParameter	"ParametersName", value	Set a model parameter to *value*. The type of value depends upon the parameter name.
ModelRun	Timeline	Start the model running.
ModelFreeze	Timeline	Pause the model.
SetVariable	"DataItemName", value	Set the model data item: "DataItemName" to *value*. This is directly equivalent to `{DataItemName} = value` except that the variable is sent as the MCL data type (Boolean, integer etc) of *value*, rather than the data type being requested from K-Spice® first. Most variables associated with blocks in the model have unique names based upon their use in the block, but there is also a standard subset of control variables used by all blocks – see Appendix B for examples. To modify values such as BlockName@B_exec and BlockName@B_dims[x] the *SetVariable* command should be used.

The typing of the arguments supplied to the *send* procedure is determined implicitly by the way the arguments are entered. If an argument is supplied as an integer, real or string constant, then it is added to the message data using the same type. If the argument is a model or system variable, then the known type of the variable will be used. The result of any expression used as an argument will be added to the message as a real value. Thus, the message:

```
send(SetVariable, "Pname", 1, name)
```

will be sent with string, integer and string arguments - assuming the variable *name* is of type string.

There are a number of simplified commands for loading, saving, etc. which are recommended to be used instead of the "send" command. See Simulator controls.

2.6.10 The wait procedure

Syntax

```
wait(seconds[, mode])
```

Description

The *wait* procedure is used to suspend the execution of the MCL for a given number of seconds. The optional *mode* value is an integer to indicate whether a real time or model time value has been supplied; 0 indicates model time (the default); 1 indicates real time. Real time 'waits' may be necessary in situations where a *wait* is required but the model is not running.

2.7 Simulator controls

MCL scripts are able to control the simulator in the same way that it can be controlled from *SimExplorer*. This means loading and saving files, controlling timelines and modifying execution speed etc. These commands can be sent to the Simulation Manager and Model Servers directly using the **Send** command, but the command structure can be complicated. Also, it is necessary to wait for some of these functions to complete before sending the next command. To simplifying things for the users a special set of commands exist to control the simulator execution. For these commands MCL also waits, where appropriate, for the operation to be completed before continuing with the script execution. This section details these commands.

MCL supports two different modes for controlling the execution of the model.

- The standard approach uses the Run, RunUntil and RunFor commands, where the *Simulation Manager* controls the stepping of the model. Using this method the model and the *MCL* script execute asynchronously and results may not be completely reproducible, especially if the model is running very quickly.
- The synchronised approach uses the RunStep, RunUntilStep and RunForStep commands, where *MCL* controls the stepping of the model. Using this method the model and the *MCL* script are completely synchronised and results will be completely reproducible. Because this method bypasses the execution control that normally occurs in the *Simulation Manager*, the model is run at the maximum speed possible and the timeline speed is not calculated.

2.7.1 The SetActiveApp function

The *SetActiveApp* function is used to set the default application for following script commands. The format of the function is:

```
SetActiveApp(text ApplicationName)
```

The following example shows the *SetActiveApp* function being used.

```
// Set the Application
SetActiveApp("/Engineering/ProcessModel")
```

```
// open valves in ProcesModel
{MV101:ControlSignalIn} = 1.0
{MV102:ControlSignalIn} = 1.0
{MV105:ControlSignalIn} = 1.0
// Set the Application
SetActiveApp("/Engineering/ProcessModel2")
// open valves in ProcessModel2
{MV201:ControlSignalIn} = 1.0
{MV202:ControlSignalIn} = 1.0
{MV205:ControlSignalIn} = 1.0
```

2.7.2 The OpenProject function

The *OpenProject* function is used to open an existing K-Spice® project. The format of the function is:

```
OpenProject(text ProjectName)
```

The following example shows the *OpenProject* function being used.

```
// Open the Project
OpenProject("DemoProject")
```

2.7.3 The NewProject function

The *NewProject* function is used to create a new K-Spice® project. The format of the function is:

```
NewProject(text ProjectName)
```

The following example shows the *NewProject* function being used.

```
// Create a new Project
NewProject("MyNewProject")
```

2.7.4 The ActivateTimeline function

The *ActivateTimeline* function is used to activate a timeline in an open K-Spice® project. The format of the function is:

```
ActivateTimeline(text TimelineName)
```

The following example shows the *ActivateTimeline* function being used.

```
// Activate the "Engienering" timeline
ActivateTimeline("/Engineering")
```

2.7.5 The DeactivateTimeline function

The *DeactivateTimeline* function is used to deactivate an active timeline in an open K-Spice® project. The format of the function is:

```
DeactivateTimeline(text TimelineName)
```

The following example shows the *DeactivateTimeline* function being used.

```
// Dectivate the "Tutorial" timeline
DeactivateTimeline("/Tutorial")
```

2.7.6 The NewModel function

The *NewModel* function is used to create a new empty model. The format of the function is:

```
NewModel([text TimelineName])
```

The following examples show the *NewModel* function being used.

```
// New model in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
NewModel(0)

// New model in a specific timeline
NewModel("/Tutorial")
```

2.7.7 The LoadModel function

The *LoadModel* function is used to load an existing model. The format of the function is:

```
LoadModel(text ModelName [,text TimelineName])
```

The following examples show the *LoadModel* function being used.

```
// Load a model in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadModel("CompressorDemo")

// Load a model in a specific timeline
LoadModel("K-SpiceTutorial","/Tutorial")
```

2.7.8 The SaveModel function

The *SaveModel* function is used to save the current model. The format of the function is:

```
SaveModel(text ModelName [,text TimelineName[,text Description[,text Comment]]])
```

The following examples show the *SaveModel* function being used.

```
// Save a model in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveModel("CompressorDemo2")

// Save a model in a specific timeline
SaveModel("MyModel","/MyTimeline")

// Save a model in a specific timeline with description and comment
SaveModel("MyModel","/MyTimeline","MyDescription","MyComment")
```

2.7.9 The LoadParameters function

The *LoadParameters* function is used to load a parameter set into an existing model. The format of the function is:

```
LoadParameters(text ParameterFileName [,text TimelineName])
```

The following examples show the *LoadParameters* function being used.

```
// Load a parameter set into the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadParameters("CompressorDemo")

// Load a parameter set into a specific timeline
LoadParameters("K-SpiceTutorial","/Tutorial")
```

2.7.10 The SaveParameters function

The *SaveParameters* function is used to save the current model parameter set. The format of the function is:

```
SaveParameters(text ParameterFileName [,text TimelineName[,text Description[,text Comm
```

The following examples show the *SaveParameters* function being used.

```
// Save the current model parameter set in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveParameters("CompressorDemo2")

// Save the current model parameter set in a specific timeline
SaveParameters("MyParameters","/MyTimeline")

// Save the current model parameter set in a specific timeline with description and co
SaveParameters("MyParameters","/MyTimeline","MyDescription","MyComment")
```

2.7.11 The LoadConditions function

The *LoadConditions* function is used to load a condition set into an existing model. The format of the function is:

```
LoadConditions(text ConditionFileName [,text TimelineName])
```

The following examples show the *LoadConditions* function being used.

```
// Load a conditions set into the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadConditions("CompressorDemo")

// Load a conditions set into a specific timeline
LoadConditions("K-SpiceTutorial","/Tutorial")
```

2.7.12 The SaveConditions function

The *SaveConditions* function is used to save the current model condition set. The format of the function is:

```
SaveConditions(text ConditionFileName [,text TimelineName[,text Description[,text Comm
```

The following examples show the *SaveConditions* function being used.

```
// Save the current model condition set in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveConditions("CompressorDemo2")

// Save the current model condition set in a specific timeline
SaveConditions("MyConditions","/MyTimeline")

// Save the current model condition set in a specific timeline with description and co
SaveConditions("MyConditions","/MyTimeline","MyDescription","MyComment")
```

2.7.13 The LoadHistory function

The *LoadHistory* function is used to load a history data set into an existing model. The format of the function is:

```
LoadHistory(text HistoryFileName [,text TimelineName])
```

The following examples show the *LoadHistory* function being used.

```
// Load a history data set into the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadHistory("CompressorDemo")

// Load a history data set into a specific timeline
LoadHistory("K-SpiceTutorial","/Tutorial")
```

2.7.14 The SaveHistory function

The *SaveHistory* function is used to save the current model history data set. The format of the function is:

```
SaveHistory(text HistoryFileName [,text TimelineName[,text Description[,text Comment]]
```

The following examples show the *SaveHistory* function being used.

```
// Save the current model history data set in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveHistory("CompressorDemo2")

// Save the current model history data set in a specific timeline
SaveHistory("MyHistory","/MyTimeline")

// Save the current model history data set in a specific timeline with description and
SaveHistory("MyHistory","/MyTimeline","MyDescription","MyComment")
```

2.7.15 The LoadTrends function

The *LoadTrends* function is used to load a trend series setup into an existing model. The format of the function is:

```
LoadTrends(text TrendFileName [,text TimelineName])
```

The following examples show the *LoadTrends* function being used.

```
// Load a trend series setup into the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadTrends("CompressorDemo")

// Load a trend series setup into a specific timeline
LoadTrends("K-SpiceTutorial","/Tutorial")
```

2.7.16 The SaveTrends function

The *SaveTrends* function is used to save the trend series setup. The format of the function is:

```
SaveTrends(text TrendFileName [,text TimelineName])
```

The following examples show the *SaveTrends* function being used.

```
// Save the current trend series setup in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveTrends("CompressorDemo2")

// Save the current trend series setup in a specific timeline
SaveTrends("MyTrends","/MyTimeline")
```

2.7.17 The LoadSnapshot function

The *LoadSnapshot* function is used to load a snapshot of the current model. The format of the function is:

```
LoadSnapshot(text SnapshotFileName [,text TimelineName])
```

The following examples show the *LoadSnapshot* function being used.

```
// Load a snapshot into the current default timeline
SetActiveApp("/Engineering/ProcessModel")
LoadSnapshot("CompressorDemo")

// Load a snapshot into a specific timeline
LoadSnapshot("K-SpiceTutorial1","/Tutorial")
```

2.7.18 The SaveSnapshot function

The *SaveSnapshot* function is used to save a snapshot of the current model. The format of the function is:

```
SaveSnapshot([,text TimelineName[,text Description[,text Comment]]])
```

The following examples show the *SaveSnapshot* function being used.

```
// Save a snapshot of the current model status in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
SaveSnapshot(0)

// Save a snapshot of the current model status in a specific timeline
SaveSnapshot("/MyTimeline")
```

```
// Save a snapshot of the current model status in a specific timeline with descriptio
SaveSnapshot("/MyTimeline","MyDescription","MyComment")
```

2.7.19 The Initialise function

The *Initialise* function is used to initialise a timeline. The format of the function is:

```
Initialise([text TimelineName])
```

The following examples show the *Initialise* function being used.

```
// Initialise the current default timeline
SetActiveApp("/Engineering/ProcessModel")
Initialise(0)

// Initialise a specific timeline
Initialise("/MyTimeline")
```

2.7.20 The Run function

The *Run* function is used to run a timeline. The format of the function is:

```
Run([text TimelineName])
```

The following examples show the *Run* function being used.

```
// Run the current default timeline
SetActiveApp("/Engineering/ProcessModel")
Run(0)

// Run a specific timeline
Run("/MyTimeline")
```

Note

To run the model and the MCL script fully synchronised use the RunStep command.

2.7.21 The Pause function

The *Pause* function is used to pause a running timeline. The format of the function is:

```
Pause([text TimelineName])
```

The following examples show the *Pause* function being used.

```
// Pause the current default timeline
SetActiveApp("/Engineering/ProcessModel")
Pause(0)

// Pause a specific timeline
Pause("/MyTimeline")
```

2.7.22 The RunUntil function

The *RunUntil* function is used to run a timeline to a specified time. The format of the function is:

```
RunUntil(time TargetTime [,text TimelineName])
```

The following examples show the *RunUntil* function being used.

```
// Run to 60 seconds in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
RunUntil(60)

// Run to 1 hour 35 mins and 20 seconds in a specific timeline
RunUntil([1:35:20],"/MyTimeline")
```

Note

To run the model and the MCL script fully synchronised use the RunUntilStep command.

2.7.23 The RunFor function

The *RunFor* function is used to run a timeline for a specified time. The format of the function is:

```
RunFor(time TargetTime [,text TimelineName])
```

The following examples show the *RunFor* function being used.

```
// Run for 60 seconds in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
RunFor(60)

// Run for 1 hour 35 mins and 20 seconds in a specific timeline
RunFor([1:35:20],"/MyTimeline")
```

Note

To run the model and the MCL script fully synchronised use the RunForStep command.

2.7.24 The SetSpeed function

The *SetSpeed* function is used to set the requested execution speed for a timeline. The format of the function is:

```
SetSpeed(speed [,text TimelineName])
```

Where *speed* is a real value greater than or equal to 0.01 (1% of real time). To run in real time *speed* should be set to 1. To run 10 times real time *speed* should be set to 10.

The maximum model speed can be requested by setting the *speed* argument to 0.

Note

The requested speed may be limited by the available processing power.

The maximum timeline speed is limited to the "MaxModelSpeed" parameter defined for each timeline (default 1000).

The following examples show the *SetSpeed* function being used.

```
// Set the active application to '/Engineering/ProcessModel'
// and then run in real time (speed = 1) for 60 seconds
SetActiveApp("/Engineering/ProcessModel")
SetSpeed (1)
RunFor(60)

// Run the 'LookAhead' timeline at maximum speed for 60 seconds
SetSpeed (0 ,"/LookAhead")
RunFor(60 ,"/LookAhead")
```

2.7.25 The RunStep function

The *RunStep* function is used to run a timeline where the timeline steps are managed by *MCL* rather than the *Simulation Manager*.

The *RunStep* command will cause the model to be run with individual step commands so that MCL is able to test any variables and perform any actions at exactly the correct time and in a completely reproducible manner. Using *RunStep* instead of *Run* means the script will be fully synchronised with the model execution.

The format of the function is:

```
RunStep([text TimelineName])
```

The following examples show the *RunStep* function being used.

```
// Run the current default timeline
SetActiveApp("/Engineering/ProcessModel")
RunStep(0)

// Run a specific timeline
RunStep("/MyTimeline")
```

Note

Because this method bypasses the execution control that normally occurs in the Simulation Manager, the model is run at the maximum speed possible and the timeline speed is not calculated.

2.7.26 The RunUntilStep function

The *RunUntilStep* function is used to run a timeline to a specified time where the timeline steps are managed by *MCL* rather than the *Simulation Manager*.

The *RunUntilStep* command will cause the model to be run with individual step commands so that MCL is able to test any variables and perform any actions at exactly the correct time and in a completely reproducible manner. Using *RunUntilStep* instead of *RunUntil* means the script will be fully synchronised with the model execution.

The format of the function is:

```
RunUntilStep([text TimelineName])
```

The following examples show the *RunUntilStep* function being used.

```
// Run to 60s in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
RunUntilStep(60)

// Run to 1 hour 35 mins and 20 seconds in a specific timeline
RunUntilStep([1:35:20],"/MyTimeline")
```

Note

Because this method bypasses the execution control that normally occurs in the Simulation Manager, the model is run at the maximum speed possible and the timeline speed is not calculated.

2.7.27 The RunForStep function

The *RunForStep* function is used to run a timeline for a specified time where the timeline steps are managed by *MCL* rather than the *Simulation Manager*.

The *RunForStep* command will cause the model to be run with individual step commands so that MCL is able to test any variables and perform any actions at exactly the correct time and in a completely reproducible manner. Using *RunForStep* instead of *RunFor* means the script will be fully synchronised with the model execution.

The format of the function is:

```
RunForStep([text TimelineName])
```

The following examples show the *RunForStep* function being used.

```
// Run for 60s in the current default timeline
SetActiveApp("/Engineering/ProcessModel")
RunForStep(60)

// Run for 1 hour 35 mins and 20 seconds in a specific timeline
RunForStep([1:35:20],"/MyTimeline")
```

Note

Because this method bypasses the execution control that normally occurs in the Simulation Manager, the model is run at the maximum speed possible and the timeline speed is not calculated.

2.8 File handling

MCL scripts are able to read and write additional files during the script execution. This can often mean reading or writing lists of blocks or individual data items. It is possible to open file in read or read/write mode. The user can step through open files line by line and use the contents within the script.

2.8.1 The FSOpenRead function

The *FSOpenRead* function is used to open a file for reading. The format of the function is:

```
FSOpenRead(text filename, handle variable, error variable)
```

FSOpenRead opens *filename* file for reading. The variable *handle* is set to a value that must be retained for use in FSReadLine() and FSClose(). If the function succeeds, the variable *error* is set to zero. *error* will be non zero if the function fails.

The following example shows the *FSOpenRead* function being used to open a file which is then processed line by line.

```
// Open a file and echo the contents
FSOpenRead("c:\test\file.txt", handle, error)
if (error != 0)
 message("Failed to open file")
else
 eof = 0
 while(eof == 0)
  FSReadLine(handle, line, eof)
  if (eof == 0) message(line)
 end
 FSClose(handle)
end
```

2.8.2 The FSOpenWrite function

The *FSOpenWrite* function is used to open a file for writing. The format of the function is:

```
FSOpenWrite(text filename, handle variable, error variable)
```

FSOpenWrite opens *filename* file for writing. The variable *handle* is set to a value that must be retained for use in FSReadLine() and FSClose(). If the function succeeds, the variable *error* is set to zero. *error* will be non zero if the function fails.

The following example shows the *FSOpenWrite* function being used to open a file which is then written to.

```
// Write the value of a variable to a file
value = {20V101:input[0].p}
output = "20V101:input[0].p="
FSToString(value, vout)
output = output + vout
FSOpenWrite("c:\test\file.txt", handle, error)
if (error != 0)
 message("Failed to open file")
else
 FSWriteLine(handle, output)
```

```
 FSClose(handle)
end
```

2.8.3 The FSReadLine function

The *FSReadLine* function is used to read the line of a file previously opened using the *FSOpenRead* function. The format of the function is:

```
FSOpenWrite(handle variable, line variable, eof variable)
```

FSReadLine reads characters from the file opened on *handle* until a carriage return or the end of file is found. The characters, excluding carriage return, will be placed in the text variable *line*. If the end of file has been reached, *eof* will be set to 1, otherwise it will be set to 0.

See *FSOpenRead* for an example.

2.8.4 The FSWriteLine function

The *FSWriteLine* function is used to write a line of a file previously opened using the *FSOpenWrite* function. The format of the function is:

```
FSWriteLine(handle variable, line variable )
```

FSWriteLine writes the contents of the *line* variable to the file opened on *handle*. The command automatically adds a carriage return when writing a line.

See *FSOpenWrite* for an example.

2.8.5 The FSWrite function

The *FSWrite* function is used to write text to a file previously opened using the *FSOpenWrite* function. The format of the function is:

```
FSWrite(handle variable, text variable )
```

FSWrite writes the contents of the *text* variable to the file opened on *handle*. The command does not automatically add a carriage return when writing.

The following example writes the contents of an array to a comma separated list.

```
// Write a comma separated set of array values to a file
FSOpenWrite("c:\test\file.txt", handle, error)
if (error != 0)
 message("Failed to open file")
else
 first_line = 1
 forlist({20V101:array}, element)
  if (first_line)
   FSWrite(handle, element)
  else
   FSWrite(handle, ",")
   FSWrite(handle, element)
  end
  first_line = 0
 end
```

```
 FSClose(handle)
end
```

2.8.6 The FSClose function

The *FSClose* function is used to close a file previously opened using the *FSOpenRead* or *FSOpenWrite* functions. The format of the function is:

```
FSClose(handle variable)
```

2.8.7 The FSFileExists function

The *FSFileExists* function is used to test if a file exists. The format of the function is:

```
FSFileExists(filename variable, exists variable)
```

FSFileExists checks for the existence of the file *filename* and sets the *exists* variable to 1 if the file exists and 0 if the file does not exist.

The following script example shows how the *FSFileExists* function can be used:

```
// See if the file exists
FSFileExists("c:\test\file.txt", exists)
if (exists == 0)
 message("The file does not exist")
else
 message("The file exists")
end
```

2.8.8 The FSCountTokens function

The *FSCountTokens* function counts the number of tokens in string. A token is any white-space delimited sequence of characters in the text input. For example, the line "x = 10.0" contains 3 tokens. This function is used with FSExtractToken() to tokenize strings for analysis. The format of the function is:

```
FSCountTokens(line variable, delimiters variable, number variable)
```

where *delimiters* is a set of characters that can act as token delimiters as well as white-space. If delimiters are used, each delimiter character becomes a token as well as the character sequences either side. If no special delimiters are required, then and empty string "" should be used. The variable *number* will contain the number of tokens found.

The following script example shows how the *FSCountTokens* function can be used:

```
// Count the tokens in a string
text = "20V101=100;"
FSCountTokens(text, "", n)
message ("number of tokens = ", n)
index = 0
for(n)
 FSExtractToken(text, "", index, t)
 message(t)
 index = index + 1
end
```

```
FSCountTokens(text, "=;", n)
message("number of tokens", n)
index = 0
for(n)
 FSExtractToken(text, "", index, t)
 message(t)
 index = index + 1
end
```

2.8.9 The FSExtractToken function

The *FSExtractToken* function extracts tokens in from a string. A token is any white-space delimited sequence of characters in the text input. For example, the line "x = 10.0" contains 3 tokens. This function is used with FSCountTokens() to tokenize strings for analysis. The format of the function is:

```
FSExtractToken(line variable, delimiters variable, number variable, token variable)
```

where *delimiters* is a set of characters that can act as token delimiters as well as white-space. If delimiters are used, each delimiter character becomes a token as well as the character sequences either side. If no special delimiters are required, then and empty string "" should be used. The variable *number* is the token number required.

See *FSCountTokens* for an example.

2.8.10 The FSStringLength function

The *FSStringLength* function gets the number of characters in a string. The format of the function is:

```
FSStringLength(line variable, number variable)
```

where *line* is the string variable to be examined and *number* is the number of characters found.

2.8.11 The FSCharAt function

The *FSCharAt* function gets the character from a certain position in a string. The format of the function is:

```
FSCharAt(line variable, position variable, character variable)
```

where *line* is the string variable to be examined, the *position* is the location of the character and *character* is the character found.

2.8.12 The FSToString function

The *FSToString* function converts a number to a string. The format of the function is:

```
FSToString(number variable, text variable)
```

where *number* is an integer or real variable and *text* is the same variable written as a string.

2.8.13 The FSToNumber function

The *FSToNumber* function converts a string to a number. The format of the function is:

```
FSToNumber(text variable, number variable)
```

where *text* is a text string containing a number and *number* is the extracted numeric value.

2.8.14 The TimeToString function

The *TimeToString* function converts a numeric time value into a user readable string. The format of the function is:

TimeToString (number variable, text variable 1, text option[, text variable 2]), where the variables are as follows:

- number variable - this variable has the time in numeric format.
- text variable 1 - this variable will be set to the time in a readable format.
- text option - the format to show the time in:
  - "Time" - set the text variable to show: HH:MM:SS (For RealTime timelines the "date" option will be used).
  - "Date" - set the text variable to show date and time: dd-mm-yy HH:MM:SS.
  - "ModelTime" - set the format as per the model clock in the SimExplorer toolbar.
- text variable 2 - the timeline for which the time format should be taken from when "ModelTime" is chosen.

The following example obtains the model time from the "Engineering" timeline and creates a time string to be shown:

```
ModelTime(time1, "/Engineering")
TimeToString(time1, timeString, "Time")
message("time1", timeString)
```

The output from the above script when run from MCL Manager is as follows:

```
[01:20:00] --- Started execution ---
[01:20:00] time1 01:20:00
[01:20:00] --- Finished normally ---
```

The following example obtains the model time from the "RealTime" timeline and creates a time and date string to be shown:

```
ModelTime(time2, "/RealTime")
TimeToString(time2, dateString, "Date")
message("time2", dateString)
```

The output from the above script when run from MCL Manager is as follows:

```
[08-12-11 10:34:33] --- Started execution ---
[08-12-11 10:34:33] time2 08-12-11 10:34:34
[08-12-11 10:34:33] --- Finished normally ---
```

The following example obtains the model time from the "RealTime" timeline and formats a time string as per the "RealTime" model clock:

```
ModelTime(time3, "/RealTime")
TimeToString(time3, timeString, "ModelTime")
message("time3", timeString)
```

The output from the above script when run from MCL Manager is as follows:

```
[08-12-11 10:34:33] --- Started execution ---
[08-12-11 10:34:33] time3 08-12-11 10:34:34
[08-12-11 10:34:33] --- Finished normally ---
```

Appendix A — MCL Script Examples

Example 1:

```
//
// Create a list of all blocks in the model with the graphic they can be found on
//

// loop for each block in the model – "TagName" is set to the block name
forlist ( {$top@name@B_n_lst},TagName )

// output for each block, getting the associated graphic from the model server
      message (TagName,"  ",{<Tagname>@B_graphic})

// end of loop
end
```

Example 2:

```
//
// Set the History on for all FieldTramsmitter blocks containging "20PI" in their tag.
//
// loop for all FieldTransmitters called 20PT*
forlist ( {$top@name(20PI*)@type(FieldTransmitter)@B_n_lst},TransmitterName )
// set history status for the "Value" data item
 EnableHistory ("<TransmitterName>:Value")
end
```

Appendix B — Examples of Block Variables

Variable	Meaning	Enumeration
@B_alarm	Highest alarm status in the block	
@B_desc	Block description text	
@B_exec	Block execution time	
@B_graphic	Graphic containing the block symbol	
@B_malfunction	Block has active malfunctions	0=no, 1=yes
@B_name	Block Name Returns the block name part of a data item	
@B_no	The execution number of the block	
@B_state	Execution State	0=inactive 1=run dynamic 2=error 3=uninitialised arrays 4=load defaults 5=initialise 6=run steady 7=check
@B_step	Step Block (no arguments)	
@B_thread	Network name for block	
@B_type	Block type (Valve, Transmitter etc)	

Appendix C — Reserved MCL Keywords

Below is a table of the MCL keywords that cannot be used as variable names. Keywords are case insensitive.

Table 2 Reserved MCL Keywords

abort	FSCharAt	LoadHistory	RunUntilStep
abs	FSClose	LoadModel	SaveConditions
acos	FSCountTokens	LoadParameters	SaveHistory
action	FSExtractToken	LoadSnapshot	SaveModel
ActivateTimeline	FSFileExists	LoadTrends	SaveParameters
asin	FSOpenRead	log	SaveSnapshot
at	FSOpenWrite	log10	SaveTrends
atan	FSReadLine	message	send
atan2	FSStringLength	mod	SetActiveApp
break	FSToNumber	ModelStatus	SetSpeed
console	FSToString	ModelTime	sin
cos	FSWrite	NewModel	sinh
cosh	FSWriteLine	NewProject	sqrt
DeactivateTimeline	get	obtain	status
DisableHistory	GetEnv	OpenProject	stop
else	goto	Pause	tan
elseif	HistorySet	pow	tanh
end	hypot	ramp	time
EnableHistory	if	rand	TimeToString
every	Initialise	Run	wait
execute	Initialize	RunFor	when
exp	int	RunForStep	while
for	label	RunStep	
forlist	LoadConditions	RunUntil	

A

L

M

R