MxScript for Fast Models

v1.3

Reference Manual

ARM®

ARM DUI 0371I (ID051811)

MxScript for Fast Models
Reference Manual

Release Information

Change history

Description	Issue	Confidentiality	Change
June 2007	A	Confidential	New document for SoC Designer 7.0.
February 2008	B	Non Confidential	Updated for System Generator 3.2 to include script functions for Model Debugger, and HDL cosimulation. Added new cast operations and constant types.
May 2008	C	Non Confidential	Updated to cover new functions for SoC Designer and System Generator.
June 2008	D	Non Confidential	Updated for MxScript 1.3 as shipped with System Generator 4.0.
December 2008	E	Non Confidential	Updated for MxScript 1.3 as shipped with Fast Models 4.1.
March 2009	F	Non-Confidential	Updated for MxScript 1.3 as shipped with Fast Models 4.2.
April 2009	G	Non-Confidential	Updated for MxScript 1.3 as shipped with Fast Models 5.0.
September 2009	H	Non-Confidential	Updated for MxScript 1.3 as shipped with Fast Models 5.1.
February 2010	I	Non-Confidential	Updated for MxScript 1.3 as shipped with Fast Models 5.2.

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
MxScript for Fast Models Reference Manual

Preface

This preface introduces the *MxScript for Fast Models Reference Manual*. It contains the following sections:

About this book

This book is the reference for the MxScript language.

Intended audience

This book is written for experienced hardware and software developers to enable you to use an MxScript file to control a debug session using Model Debugger.

Organization

This book is organized into the following chapters:

Chapter 1 *Introduction and syntax conventions*

Read this chapter for an introduction to the MxScript language.

Chapter 2 *Common API*

Read this chapter for a description of the common API provided by the MxScript language.

Chapter 3 *Model Debugger Scripting Functions*

Read this chapter for a description of Model Debugger API functions that are available for use in batch-mode scripts.

Typographical conventions

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Additional reading

This section lists related publications by ARM®.

See *ARM Infocenter*, http://infocenter.arm.com/help/index.jsp for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- ARM architecture Reference Manuals, http://infocenter.arm.com/help/index.jsp
- *ARM Cycle Accurate Debug Interface Developer Guide* (ARM DUI 0444).

The following publication provides information about related ARM products:

- *Model Debugger for Fast Models User Guide.* (ARM DUI 0314).

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- the product name
- a concise explanation.

Feedback on this book

If you have any comments on this book, send an e-mail to `errata@arm.com`. Give:

- the title
- the number
- the relevant page number(s) to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction and syntax conventions

This chapter describes the syntax and usage of the MxScript language. It contains the following sections:

1.1 Introduction to MxScript

MxScript is an interpreted language with a syntax that is similar to C. MxScript provides the following benefits:

Easy to learn

Syntax is similar to C.

Integers can contain 64 bit signed values and support all operations that C supports. There are only integer, double, bool, and string types.

Safe Bugs in the script file do not cause a system crash.

Strings in MxScript are safer than in C because features not required for scripting have been removed. There is no use of pointers, structures, user defined functions, or arrays.

Flexible No compilation is required and fast turnarounds are possible. MxScript can be used interactively in a command-line interface.

Fast Unlike many other scripting languages, performance was one of the main goals for MxScript.

The MxScript language can be invoked from the following initial situations:

- a single command can be issued from the Model Debugger Output window
- a script containing multiple commands can be specified on the command line that starts Model Debugger
- a script containing multiple commands can be loaded into Model Debugger after it has started.

1.2 Syntax conventions

This section describes the basic language keywords and structures.

1.2.1 Comments

Two types of comment are supported:

Line comments

These start with "//"and end at the end of the current line.

Block comments

These start with "/*"and end with "*/".

As with C, it is not possible to nest block comments.

In the code "`/* a /* b */ c */ …`", the part after `c */` is not in a comment and probably leads to a syntax error.

Note

Comments cannot occur in string constants.

1.2.2 Identifiers

The following rules apply to identifiers:

- they must consist of letters and digits
- the first character must be a letter
- the underscore '_' counts as a letter
- upper and lower case letters are different
- identifiers are distinguished on their full length.

1.2.3 Keywords

Not all C keywords are supported within MxScript, but they are, however, reserved for compatibility and future extension:

Supported keywords

```
break bool continue do double else false for if int string true while
```

Reserved keywords

```
asm auto case char complex const default enum extern float
goto inline long register return short signed sizeof static struct
switch typedef union unsigned void volatile wchar_t
```

1.2.4 Operators

The supported operators are listed in Table 1-1:

Table 1-1 MxScript operators

Category	Operators	Restrictions
Assignment	=	Works on all types and returns the same type.
Arithmetical	+ - * % ++ -- += -= *= /= %=	Work on all number types (int and double) and the result of same type, except that the increment operators ++ and -- can only be used with int values.
String	+ = += *=	Use to concatenate strings, assign to string, or append to string. The *= form is used to concatenate multiple copies of a string back to the original string as in my_string_var *= 3.
Relationship	== != < > <= >=	Works on all types, including strings. Result is bool. The <, >, <=, and >= cannot be used with bool types.
Logical	&& || !	Works on bool types. Result is bool
Bitwise	& | ^ ~ << >>&= |= ^= <<= >>=	Works on int. Result is int. Shift operators are, unlike in C, well defined for shifts larger than the size of the integer type (64 bits).
Casting	type(exp) (type) exp	Both C and C++ forms of casts are supported in MxScript See *Expressions* on page 1-6.
Pointers	unary * &	Not supported in MxScript.
Structures	. ->	Not supported in MxScript.

The precedence and associativity of operators in MxScript is the same as for C. See Table 1-2:

Table 1-2 Associativity in expressions

Operators	Associativity
( )	left to right
unary operators: !, ~, ++, --, +, -, (type), type( )	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right

Table 1-2 Associativity in expressions (continued)

Operators	Associativity
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
,	left to right

1.2.5 Constants

There are the following types of constant:

Integer constants

Integer constants can be in decimal, hexadecimal, octal and binary format:

- octal constant begin with a leading `0`
- hexadecimal constants begin with the prefix `0x` or `0X`
- binary numbers begin with the prefix `0b` or `0B`.
- all other numbers are treated as decimal constants. Suffixes like `U` or `L` are allowed but are ignored.

String constant

A string constant is surrounded by double quotes. Special escape sequences that begin with a backslash \ can be used to include control characters into a string. See Table 1-3. To put a put a backslash into a string a double backslash \\ must be used.

Characters can also be specified using octal or hexadecimal ASCII code.

Table 1-3 Escape characters for string constants

Name	Escape Sequence
Newline	`\n`
horizontal tab	`\t`
vertical tab	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
alert	`\a`
backslash	`\\`
question mark	`\?`
single quote	`\'`
double quote	`\"`
character by octal ASCII code `ooo`	`\ooo`
character by hexadecimal ASCII code `hh`	`\xhh`

Boolean constants

The Boolean constants are `true` and `false`.

Double constants

A double is a floating-point number represented with 64 bits. For example: 3.14, 5.4E14, or 3E-7.

1.2.6 Types

MxScript supports the following types:

`int` Integers are represented as 64 bit signed values, so numbers between `-9223372036854775808` and `+9223372036854775807` can be represented.

`double` Doubles are represented as 64 bit signed values consisting of a mantissa and exponent. Doubles are represented as floating-point numbers.

`bool` Boolean variables can only have the value `true` or `false`.

`string` Strings are sequences of ASCII characters. String size is only limited by available memory and can contain more characters than any practical application could require.

Variable definitions

A variable definition consists of a type and a list of identifiers that are not already in use for the current scope. The identifiers must not be keywords and must not be the names of functions predefined by the MxScript environment.

The scope for a variable is either:

Local The scope is limited by a surrounding block of curly braces or by being declared inside a `for` loop. A block of code uses the variable definition that is in the innermost definition. This is the same scope as for C.

Global A variable is global if it is on the top level.

1.2.7 Expressions

An expression consists of constants, variables, and function calls that are combined with operators.

Parentheses can be used to group expressions to alter the evaluation sequence from that defined by the precedence:

```
3*(4+7)
```

Unlike in C, there is no automatic type casting in MxScript. The expression `(3.14 *2)` causes an error because double and int types are mixed. Both C and C++ forms of casts are allowed.

A string can be multiplied by an integer to create a concatenated string:

- `"hello" * 2` is equivalent to "`hellohello`"
- `4 * "#"` is equivalent to "`####`".

String/integer casts are permitted:

- `(string)5` is equivalent to `"5"`
- `string(5+77)` is equivalent to `"82"`
- `int("555")` is equivalent to `555`

- `(int)("0b"+ "111")` is equivalent to `7`.
- `int("xfff")` is equivalent to `0` because the string does not start with `0`
- `int("255xfff")` is equivalent to `255` because the non-numbers are ignored.

The results of the different cast combinations are listed in Table 1-4.

Table 1-4 Results of cast operation

Original type	casting to int	casting to string	casting to bool	casting to double
int	Error.	Convert to string containing decimal integer format.	`false` if integer is 0, otherwise `true`.	Convert to double with same value.
string	Interpret string as integer number. Prefixes `0b` (binary), `0x` (hexadecimal) and `0` (octal) are recognized.	Error.	Error.	Interpret string as a decimal floating-point number in C format.
bool	1 if `true`, 0 if `false`.	"`true`" if `true`, "`false`" if `false`.	Error.	Error.
double	Round down to a lower integer value. Same as `floor()` function in C.	Convert to string containing decimal floating-point format.	Error.	Error.

1.2.8 Calling built-in functions

Call built-in functions by using the function name followed by a comma-separated list of parameters in parentheses. A parameter can be a single value or an expression.

For convenience, a function that does not have parameters can be called by its name, if the name does not match the name of any variable in the code. An empty pair of parentheses can be appended but is not mandatory.

1.2.9 Control Statements

This sections describes the supported control statements.

if statement

The `if` statement is used to execute an instruction or a block of instructions depending on a condition.

The condition must be of `bool` type. If it evaluates to `false`, the code is not executed. If it evaluates to `true`, the code is executed.

```
if (condition)
    statement;
```

or

```
if (condition)
    {
        statement 1;
        …
        statement n;
    }
```

If statements can be nested, for example:

```
    if (condition1)
    {
        statement1;
        if (condition2)
        {
            statement2;
        }
    }
```

else statement

The `else` statement is used to append an alternate code block to an `if` statement. The alternate block is executed if the condition of the `if` statement is `false`.

```
if (condition)
    statement;
else
    alternate statement;
```

`if` and `else` statements can be nested. If the relationship is ambiguous, an `else` always belongs to the last `if` statement:

```
if (condition) /* 1 */
    if (condition) /* 2 */
        statement1;
    else /* belongs to if 2 */
        statement2;
```

It is good style, however, to remove ambiguity by using additional blocking:

```
if (condition) /* 1 */
    {
        if (condition) /* 2 */
            statement1;
        else /* belongs to if 2 */
            statement2;
    }
```

To check for multiple conditions of which only one is true, the following construct can be used (no special `elseif` instruction exists):

```
if (condition)
{
}
else if (condition2)
{
}
else if (condition3)
{
}
else
{
}
```

for statement

The `for` keyword is followed by an initial value for an integer variable, an exit condition, a modifier function, and a statement or a block containing statements.

The statements in the `for` loop are executed until the condition is `false`.

```
for (loop_var; condition; modifier)
    statement;
```

or

```
for (loop_var; condition; modifier)
{
    statement1;
    statement2;
}
```

For statements can be nested.

If the loop variable is declared in the `for` statement, its use is local to the `for` block:

```
for (int i; i<3; ++i)
{
    statement1;
    statement2;
}
```

while statement

The `while` keyword is followed by a condition (which must evaluate to an bool) and a statement or a block containing statements. The statements in the `while` loop are executed until the condition is `false`. If the condition is `false` when entering the `while` loop the statements are not executed.

```
while(condition)
    statement;
```

or

```
while(condition)
{
    statement1;
    statement2;
}
```

Loop statements can be nested:

```
while (condition)
{
     …
    while (condition)
    {
        …
    }
    …
}
```

The `do while` form is similar to the `while` form except that the statements are evaluated before the test. If the condition is `false` when entering the `while` loop the statements are executed once.

```
do
    statement
while(condition);
```

or

```
do
{
    statement1;
```

```
    statement2;
}
while(condition);
```

break statement

The `break` statement can be used to prematurely leave `while`, `do while`, or `for` loops. If used in nested loops the innermost loop is exited.

```
while (condition)
{
    if (condition2)
        break;
    …
}
```

continue statement

The keyword `continue` can be used to jump over the remainder of a `while`, `do while`, or `for` loop body and to continue with the evaluation of the condition.

```
while (condition)
{
    if (condition2)
        continue;
    …
}
```

If used in nested loops, the innermost loop is continued.

Chapter 2
Common API

This chapter describes the API functions that are common to batch-mode and GUI scripting environments for Model Debugger. It contains the following sections:

2.1 File input/output

This section describes the functions that perform file input and output.

In MxScript, file I/O is done with functions that are similar to ANSI C file functions.

2.1.1 fopen()

```
int fopen(string filename, string mode)
```

Open a file specified by *filename* (the parameter *filename* can contain a path) with the specified *mode*. Supported modes are listed in Table 2-1:

Table 2-1 Mode options for fopen()

Text mode	Binary mode	Description
r	rb	Open a text/binary file for reading
w	wb	Create a text/binary file for writing. Previous contents, if any, are discarded.
a	ab	Open a text/binary file for update. Data is written at the end of the file.
r+	r+b	Open a text/binary file for reading.
w+	w+b	Create a text/binary file for update. Previous contents, if any, are discarded.
a+	a+b	Open or create text/binary file for update. Data is written at the end of the file.

If successful, a handle to the file opened is returned which can be passed to other file I/O functions. If unsuccessful a error message is displayed and 0 is returned.

2.1.2 fclose()

```
fclose(int filehandle)
```

Executes a standard C++ `fclose()`, closing the file that was opened using `fopen()`. No value is returned.

2.1.3 fprintf()

```
int fprintf(int filehandle, string format, …)
```

This function writes data into a file. Most features of the ANSI C standard are supported.

2.1.4 fputs()

```
fputs(string s, int filehandle)
```

Prints the string *s* into the file associated with *filehandle*.

2.1.5 fgets()

```
int fgets(string s, int n, int filehandle)
```

Reads, at most, the next *n*-1 characters into the string *s* from the file being associated with *filehandle*. If a newline is encountered, the newline is included in the string. The string is terminated by "\0".

Note

In contrast to ANSI C, fgets() returns either:

- the number of characters read
- 0 if the end of file was reached or an error associated with *filehandle* occurred.

2.1.6 fscanf()

```
int fscanf(int filehandle, string format, …)
```

Reads in data. Most format options of the ANSI C standard are supported.

Note

Due to the absence of pointers, variables of type int or string are provided directly rather than pointers as in ANSI C.

2.1.7 ftell()

```
int ftell(int filehandle)
```

Returns the value, in bytes, of the file position pointer for the file associated with *filehandle*.

2.1.8 fflush()

```
void fflush(int filehandle)
```

Commits any pending writes to for the file associated with *filehandle*.

2.1.9 fseek()

```
void fseek(int filehandle, int offset, int whence=SEEK_END)
```

Move the file position pointer by *offset* bytes for the file associated with *filehandle*.

The starting point for the move is determined by the *whence* parameter:

SEEK_SET — The new position is *offset*. The movement was relative to the start of the file.

SEEK_CUR — The new position is the current position plus *offset*.

SEEK_END — The new position is the end of file plus *offset*. The movement is relative to the start of the file. To move backwards from the end of file, a negative value must be supplied for *offset*.

2.2 Handling strings

This section describes functions related to string handling.

2.2.1 sscanf()

```
int sscanf(string str, string format, …)
```

Reads in data from a string. Most format options of the ANSI C standard are supported.

Note

In contrast to ANSI C, `sscanf()` returns either:

- the number of characters read
- `0` if the end of file was reached or an error associated with *`filehandle`* occurred.

2.2.2 sprintf()

```
int sprintf(string buf, string format, …)
```

Formats data (according to *`format`*) and assigns the result to the string *`buf`*. Most format options of the ANSI C standard are supported.

2.2.3 substr()

```
string substr(string s, int pos, int length)
```

Returns a substring of string *`s`* by copying *`length`* number of characters starting at position *`pos`*.

2.2.4 gets()

```
string gets()
```

Reads the next input line from the input console and returns a string. The newline character "`\n`" is replaced with "`\0`".

2.2.5 ascii2int()

```
int ascii2int(string s)
```

Reads the first character of string *`s`*, that is `s[0]`, and interprets it as ASCII character and returns the appropriate integer value.

2.3 Accessing environment variables

Access of environments variable is done with functions that are similar to the standard C versions.

2.3.1 getenv()

```
string getenv(string env_varname)
```

Returns the value of the environment variable with name *varname*. If no such environment variable exists, an empty string is returned.

2.3.2 putenv()

```
int putenv(string putenv_string)
```

Adds a new environment variable or alters the value of an existing one.

The parameter *putenv_string* must have the form "*env_var*name=value". If the setting of the environment variable was successful 0 is returned. If an error occurs, the value –1 is returned.

Note

This function only alters the environment of the current process. It cannot be used to alter the environment of the parent process, therefore it cannot be used to pass back information to a calling process.

2.3.3 system()

```
int system(string cmd_str)
```

`system()` synchronously passes the string *cmd_str* to the environment (host operating system) for execution. Because the call is synchronous, the script does not return from this function until the command in cmd_str has completed.

If *cmd_str* is "" (empty string) and there is a command processor, `system()` returns a non-zero value.

If *cmd_str* is not "" (empty string), the return value is implementation-dependent.

2.4 Preprocessor

The MxScript interpreter contains a preprocessor. Use the `#include` directive to include C header files. This enables sharing `#define` preprocessor statements between MxScript files and C projects.

Note

The preprocessor is currently only available with component scripting. Batch-mode scripting does not support preprocessor commands.

2.4.1 #include

Include C header files containing preprocessor definitions. For example, to include the `header.h` file, use:

```
#include “header.h”
```

2.4.2 #define

Preprocessor define directive. For example, to replace any occurrence of "*base*" with "`0x1234`" in all MxScript source that is parsed after the define, use:

```
#define base 0x1234
```

Chapter 3
Model Debugger Scripting Functions

This chapter describes the MxScript commands available for use with Model Debugger. It contains the following sections:

3.1 Introduction

This section describes how to use MxScript commands with Model Debugger.

MxScript commands can be executed by Model Debugger in the following ways:

Executing a single command from Model Debugger

Some execution and debugging features of Model Debugger can be controlled by entering an MxScript command in the Output window. Enter the command text into the command line, located to the right of the **cmd>** button, and click **cmd>**.

Executing a script from Model Debugger

To run a script file after Model Debugger has started, enter `loadScript("`*`filename`*`")` in the Output window command line.

Specifying a script file at Model Debugger startup

Enter one of the following options on the command line to execute a script file in Model Debugger:

- `modeldebugger --script` *`filename`*
- `modeldebugger -s` *`filename`*.

3.2 Model connection and configuration

This section describes the commands for connecting to a model.

3.2.1 loadModel()

```
void loadModel(string pathAndFileName, bool hostLevelDebugger,
          string targetInstanceName)
```

Load a model library file from the location specified by *pathAndFileName*.

The model shared library file must be supplied. The file extensions for shared libraries can be `.cadi`, `.so` (Unix), `.dll` (Windows), or `.mxdi`.

Note

The option *hostLevelDebugger* is deprecated. Setting this parameter has no effect on the function.

If a model contains multiple targets, *targetInstanceName* is used to select one target. The default is to take the first target. Use `getTargetList()` to identify all available targets.

3.2.2 closeModel()

```
void closeModel()
```

Close the currently loaded model.

3.2.3 connectToModel()

```
void connectToModel(string port:inst)
```

Connect to a model.

3.2.4 debugIsim()

```
void debugIsim(string isimSystem, string parameterFile)
```

Run *isimSystem* and connect automatically. Define parameters for the system in the *parameterFile*. The parameter file is optional.

Note

If the system or parameter file does not exist, then a run-time error occurs.

3.2.5 debugSystemC()

```
void debugSystemC(string systemCSimulation, string application)
```

Run *systemCSimulation* and connect automatically. Defining an *application* is optional. It might be necessary depending on the SystemC simulation you are debugging.

Note

If the simulation or application file does not exist, then a run-time error occurs.

3.2.6 getParameter()

```
string getParameter(string parameterName)
```

Get a model parameter value for *parameterName*. Returns the value as a string.

Note

If a parameter with the specified name does not exist, then a run-time error occurs.

3.2.7 setParameter()

```
void setParameter(string parameterName, string value)
```

Assign the string representation of the value in *value* to the model parameter specified by *parameterName*.

Note

If a parameter with the specified name does not exist, then a run-time error occurs.

3.2.8 getTargetList()

```
void getTargetList(string modelName)
```

Print a list of the available target instances of the specified model.

3.2.9 getTargetName()

```
string getTargetName()
```

Return the qualified name of the currently selected target.

3.2.10 selectTarget()

```
void selectTarget(string targetName)
```

Set the target for all subsequent scripting commands. *targetName* is the qualified target name and must be in the same format as used in the Model Debugger Select Target dialog.

The function can be used in a script or in the command line of the Model Debugger Output window:

- If used in a nested script, the target is set for all subsequent scripts.
- If used on the command line of the Model Debugger Output window, the function only sets the target for the Model Debugger window where it was used.

3.2.11 loadApp()

```
void loadApp(string pathAndFileName, bool debugInfoOnly)
```

Load the application file or`*.elf` file specified by *pathAndFileName*. For ARM cores, the application file is typically a `.axf` file (`axf` is `ELF` compatible).

You can also load `.hex` (Intel), S-record, or COFF files, but there is reduced, or no, debug information.

DebugInfoOnly can be either false or true. The default is false. If true, only the debug information is loaded into the debugger and the code to be executed must have been already loaded.

3.2.12 saveState()

`void saveState(string modelStateFileName)`

Save a state of a model currently being debugged to the `.model_state` file specified by *modelStateFileName*.

3.2.13 restoreState()

`void restoreState(string modelStateFileName)`

Restore a model from the previously saved `.model_state` file specified by *modelStateFileName* and continue debugging.

3.2.14 saveSession()

`void saveSession(string sessionFileName, bool saveModelState)`

Save a Model Debugger session to the file specified by *saveModelState*. All the session data, including, model, application, breakpoints, and model parameters, is saved. If you set *saveModelState* to true, the current model state is also saved.

3.2.15 openSession()

`void openSession(string sessionFileName)`

Restore a Model Debugger session from a previously saved file.

Note

It is not possible to open a session in GUI mode if it was saved in non-GUI mode.

3.2.16 setStateFile()

`void setStateFile(string stateFileName)`

Specify the `.model_state` file that is saved with your Model Debugger session. This state is used if you use the `saveSession()` command with the *saveModelState* parameter equal to true. By default, the session name is used.

3.3 Model execution control

This section describes the script commands related to model execution.

3.3.1 run()

```
void run()
```

Run the simulation until a breakpoint is hit or an exception, such as simulation halt, occurs.

3.3.2 runUntil()

```
void runUntil(int address)
```

Run the simulation until the pc address specified in *address* is reached.

3.3.3 runToLine()

```
void runToLine(string filename, int lineNumber)
```

Run the simulation until the source code line specified in the *lineNumber* of the file specified in *filename* is reached.

3.3.4 stop()

```
void stop()
```

Stop the execution of the model being debugged. This command is not supported in batch mode.

3.3.5 getCurrentSourceFile()

```
string getCurrentSourceFile()
```

Return the name of the source file that matches the current simulation cycle. An empty string is returned if there is no current source file.

3.3.6 getCurrentSourceLine()

```
int getCurrentSourceLine()
```

Return the line number in the source that matches the current simulation cycle. Returns –1 if there is no current source file.

3.3.7 getCurrentSourceColumn()

```
int getCurrentSourceColumn()
```

Return the position in the source line that matches the current simulation cycle. Returns –1 if there is no current source file.

3.3.8 hardReset()

```
void hardReset()
```

Execute a reset of the target model without reloading the application.

3.3.9 reset()

```
void reset()
```

Execute a reset of the target model and reload the application.

3.3.10 pause()

```
void pause()
```

Pause the current high-level simulation step command such as source-level step over.

3.3.11 cont()

```
void cont()
```

Continue the current high-level simulation step command such as, for example source-level step over.

High level simulation-control commands can be interrupted by breakpoints before completion. The control commands can be completed by `cont()`. This is not supported for batch mode.

3.3.12 getStopCond()

```
string getStopCond()
```

Return a message string that describes the reason for the last stop condition if the simulator is currently in the stopped state. The string format depends on the reason for the stop condition:

- For a PC breakpoint, the string describes the stop condition, the source file, and the line number such as, for example:

  ```
  Disassembly breakpoint is hit - address: 0x00008018
  ```

- General stop conditions might return one of `NORMAL USER STOP`, `TERMINATE`, `HALT`, `EXCEPTION`, `ERROR`, or `INVALID OPCODE`.

3.3.13 isSimStopped()

```
int isSimStopped(string stopCondition)
```

Return `True` if the simulator is currently in stopped state or `False` if the simulator is running.

`stopCondition` is an optional parameter to enable more detailed checking:

- To check for a exact stop condition such as a breakpoint at a specific address, the string must be constructed exactly like the string returned by `getStopCondition()`.

- To check for a general stop condition, the string can be one of `TERMINATE`, `HALT`, `BREAKPOINT`, `BP`, `EXCEPTION`, `ERROR`, `INVALID OPCODE` or `NORMAL USER STOP`. (`BP` is a short for `BREAKPOINT` and both strings can be used interchangeably.)

3.3.14 restart()

```
void restart()
```

Execute a restart of the target model. This is a reset plus reload of the application code.

3.3.15 goToMain()

```
void goToMain()
```

Execute a reset of the target model and run until the main function (label) of the application source code is reached.

Note

This command is only available if a `main()` function can be found in the debug information of the application file.

3.3.16 step()

```
void step()
```

Execute the simulation until a different source line is reached. This is a source-level execution control command.

3.3.17 stepOver()

```
void stepOver()
```

Step over function calls. This is a source-level execution control command.

3.3.18 stepOut()

```
void stepOut()
```

Leave the current function. This is a source-level execution control command.

3.3.19 istep()

```
void istep(int numberOfInstructions)
```

Advance the simulation by executing as many instructions as specified in the *`numberOfInstructions`* parameter. One step is assumed if *`numberOfSteps`* is omitted.

3.3.20 getInstCount()

```
int getInstCount()
```

Return the number of totally counted instructions since last reset.

3.3.21 cycleStep()

```
void cycleStep(int numberOfCycles)
```

Advance the simulation by the number of cycles specified in *numberOfCycles*. If *numberOfCycles* is positive, the simulation is stepped forward.

If *numberOfCycles* is negative, the simulation is stepped backward.

Note

A negative parameter value causes a run-time error if stepping back is not enabled.

3.3.22 enableStepBack()

```
void enableStepBack(bool enable)
```

Enable the use of negative values in `cyleStep()` to step back in the simulation cycles.

Note

This command is not supported by all model targets. This command causes a run-time error if the target does not support Step Back.

3.3.23 sleep()

```
void sleep(int numberOfSeconds)
```

Wait for the number of seconds specified in the parameter. One second is assumed if *numberOfSeconds* is omitted.

3.3.24 msleep()

```
void msleep(int numberOfMilliseconds)
```

Wait for the number of milliseconds specified in the parameter. One millisecond is assumed if *numberOfMilliseconds* is omitted.

3.3.25 getCycleCount()

```
int getCycleCount()
```

Return the cycle the simulation is in.

3.4 Breakpoints

This section describes the script commands related to breakpoints.

3.4.1 bpAdd ()

```
int bpAdd (int address, string memspace)
```

Add a breakpoint at the specified program counter address using the specified memory space.

The parameter *memspace* is optional. If omitted the first program memory space is used. Valid values for this parameter are "Normal" and "Secure".

If the specified memory space does not exist a run-time error occurs.

Returns the id number of the new breakpoint.

3.4.2 bpAdd()

```
int bpAdd (string filename,int lineNumber)
```

Add a breakpoint at the source code line specified in *lineNumber* of the file specified in *filename*.

Returns the id number of the new breakpoint.

3.4.3 bpAddReg()

```
int bpAddReg (string regName)
```

Add a breakpoint at the register specified in *regName*. If the register does not exist, a run-time error occurs.

A hierarchical name is required for the parameter if register names are not unique. You must specify the register group. Compound registers must include the name of the parent. The format for hierarchical items uses dots to separate the names. For example:

```
REGGROUP0.reg0.compound0
```

Returns the id number of the new breakpoint.

3.4.4 bpAddReg()

```
int bpAddReg (int id)
```

Add a breakpoint at the register specified in *id*. If the register does not exist, a run-time error occurs.

Returns the id number of the new breakpoint.

3.4.5 bpAddMem()

```
int bpAddMem(int address, string memspace)
```

Add a breakpoint at the address specified in *address* of the memory space specified in *memspace*. If the address is out of range or the memory space does not exist, a run-time error occurs.

Valid values for the memspace parameter are "Normal" and "Secure".

Returns the id number of the new breakpoint.

3.4.6 bpAddMem()

```
int bpAddMem(int address, int id)
```

Add a breakpoint at the address specified in *address* of the memory space specified in *id*. If the address is out of range or the memory space does not exist, a run-time error occurs.

Returns the id number of the new breakpoint.

3.4.7 bpRemove ()

```
void bpRemove (int id)
```

Remove the breakpoint with the specified id.

3.4.8 bpRemoveAll()

```
void bpRemoveAll()
```

Remove all existing breakpoints.

3.4.9 bpEnable ()

```
void bpEnable (int id)
```

Enable the breakpoint specified by *id*.

Note

This command can cause a run-time error.

3.4.10 bpEnableAll()

```
void bpEnableAll()
```

Enable all existing breakpoints.

3.4.11 bpDisable()

```
void bpDisable(int id)
```

Disable the breakpoint specified by *id*.

Note

This command can cause a run-time error.

3.4.12 bpDisableAll()

```
void bpDisableAll()
```

Disable all existing breakpoints.

3.4.13 bpList()

```
void bpList()
```

Print a list of all existing breakpoints with locations, details and conditions.

3.4.14 bpSetTriggerType()

```
void bpSetTriggerType (int breakpoint_id, string triggerType)
```

Trigger the breakpoint specified in *breakpoint_id* only if the breakpoint type specified in *triggerType* occurs. The type can be "READ" , "WRITE", "MODIFY", or combinations of the types separated by '|' .

3.4.15 bpSetIgnoreCount()

```
void bpSetIgnoreCount (int breakpoint id, int numberOfCounts)
```

Stop the simulation run only if the breakpoint specified in *breakpoint_id* has been hit *numberOfCounts* times.

3.4.16 bpSetCond()

```
void bpSetCond (int breakpoint_id, string conditionOperator,
                int comparisonValue)
```

Trigger the breakpoint specified in *breakpoint_id* only if the condition specified by *comparisonValue* and *conditionOperator* is true.

conditionOperator can be one of "ANY", "EQ" , "NE", "GT", "LT", "LE", or "GE" .

3.4.17 bool bpIsHit ()

```
bool bpIsHit (int breakpoint_id)
```

This function returns true if the breakpoint specified by *id* is currently hit.

Note

If the breakpoint specified by *breakpoint_id* does not exist, a run-time error occurs.

3.5 Model resource access

This section describes the script commands related to accessing model memory or register resources. The CADI interface is always used to perform the call.

3.5.1 regWrite()

`void regWrite (string registerName, value)`

Write a value to the specified register.

A hierarchical name is required for the parameter if register names are not unique. You must specify the register group. Compound registers must include the name of the parent. The format for hierarchical items uses periods to separate the names. For example:

`REGGROUP0.reg0.compound0`

Note

If the register does not exist, a run-time error occurs.

3.5.2 regRead()

`int regRead (string registerName)`

Read a value from the specified register.

A hierarchical name is required for the parameter if register names are not unique. You must specify the register group. Compound registers must include the name of the parent. The format for hierarchical items uses periods to separate the names. For example:

`REGGROUP0.reg0.compound0`

Note

If the register does not exist, a run-time error occurs.

3.5.3 memWrite()

`void memWrite(string memspace, int address, int value,int numberOfMAU=1)`

Valid values for the `memspace` parameter are "`Normal`" and "`Secure`".

Write a value in the specified memory space at the address specified in *`address`*. *`Value`* can be of type string or integer. The size of the access depends on the *Minimum Addressable Unit* (MAU) size which is the size of one word defined for that memory space.

Use the optional parameter `numberOfMAU` to specify how many MAUs are written in a single call. The default size for `numberOfMAU` is 1.

Note

This command can cause a run-time error.

The function can only write 64 bits (8 bytes) at a time. To prevent a run-time error, the value of `numberOfMAU * bytePerMAU` must be less than 8.

3.5.4 memRead()

```
int memRead(string memspace, int address,int numberOfMAU=1)
```

Valid values for the `memspace` parameter are "`Normal`" and "`Secure`".

Read a value from the specified memory space at the address specified in *`address`*. Returns the integer value. The size of the access depends on the *Minimum Addressable Unit* (MAU) size which is the size of one word defined for that memory space.

Use the optional parameter `numberOfMAU` to specify how many MAUs are read in a single call. The default size for `numberOfMAU` is 1.

Note

This command can cause a run-time error.

The function can only read 64 bits (8 bytes) at a time. To prevent a run-time error, the value of `numberOfMAU * bytePerMAU` must be less than 8.

3.5.5 disassemble()

```
string disassemble(int address, int memory_space_id, int disassembly_mode)
```

Return the assembler string representation of the code at *`address`* in the memory area specified by *`memory_space_id`*. The *`dissassembly_mode`* parameter selects the architecture used to determine the disassembly.

3.5.6 memStoreToFile()

```
int memStoreToFile(string filename, bool isASCIIMode, string memspace,
                   int startAddress, int endAddress)
```

Read data from the memory space *`memspace`* starting at address *`startAddress`* and stop when address *`endAddress`* is reached. The data that is read is stored in the file *`filename`*. The file format can be either binary or ASCII. The value of *`isASCIIMode`* must be set to true for ASCII file format and false for binary.

If no memory space with the name *`memspace`* exists, a run-time error occurs. The size of the access is determined by the *Minimum Addressable Unit* (MAU) size defined for that memory space. The MAU is the size of one memory word.

Valid values for the `memspace` parameter are "`Normal`" and "`Secure`".

3.5.7 memLoadFromFile()

```
int memLoadFromFile(string filename, bool isASCIIMode, string memspace,
                    int startAddress, int endAddress)
```

Read data from the file *`filename`* and write to memory space *`memspace`* starting at address *`startAddress`* and stop when address *`endAddress`* or the end of the file is reached. The parameter *`endAddress`* is optional, if omitted the memory space max address is used. The file format can be either binary or ASCII. The value of *`isASCIIMode`* must be True for ASCII file format and False for binary.

If no memory space with the name *`memspace`* exists, then a run-time error occurs. The size of the access is determined by the *Minimum Addressable Unit* (MAU) size defined for that memory space. The MAU is the size of one memory word.

Valid values for the `memspace` parameter are "`Normal`" and "`Secure`".

3.6 String and print functions

This section describes the script commands related to string output.

3.6.1 printf()

```
int printf(string format, …)
```

Print a string to the output window. Most format options of the ANSI C standard are supported. The return value is the number of characters printed.

3.6.2 puts()

```
void puts(string s)
```

Write a string to the output window.

3.7 Miscellaneous

This section describes the miscellaneous script commands that do not fit into the other categories.

3.7.1 CADIXfaceBypass()

```
int CADIXfaceBypass(string Command, string result)
```

Call the CADI bypass function for the model with the command passed in *command*. The *result* argument contains the result, if any, as a string.

Return values and their meaning are listed in Table 3-1.

Table 3-1 CADIXfaceBypass return values

Returned value	Status
0	OK. Command was successful.
1	General error
2	Unknown command error
3	Illegal argument error
4	Command not supported error
5	Argument not supported error
6	Insufficient resources error
7	Target not responding error
8	Target busy error

3.7.2 exit()

```
void exit()
```

Exit Model Debugger.

3.7.3 getMxScriptVersion()

```
string getMxScriptVersion()
```

This function returns a string containing the version of MxScript.

3.7.4 help()

```
void help(string command)
```

Show a help list for:

- all commands if the parameter *command* is omitted.
- a detailed description for command specified in *command*.

3.7.5 ld()

```
int ld(int arg)
```

The binary logarithm function returns the bit position of the most significant bit of the *arg* that is set to one.

Note

Values of *arg* smaller than or equal to zero result in a run-time error.

3.7.6 loadScript()

`void loadScript(string scriptFileName)`

Load a Model Debugger script file that contains commands to execute. This can be used instead of using the `-script` switch when starting Model Debugger.

Note

This command can only be nested once in a script file.

If the `loadScript()` command is entered in the command line, the command cannot be nested at all.

3.7.7 printReg()

`void printReg(string regname)`

Print the contents of the register. For example, `printReg("R0")` outputs `R0=0x1234567`.

3.7.8 rand()

`int rand(int min, int max)`

Return a random value from *min* to *max* (inclusive).

3.7.9 string eval()

```
string eval(string expression)
```

Evaluate *expression* and return the value as a string. This has the same functionality as evaluations done in the Watch window.