# MxScript v3.3 for Cycle Models

**SoC Designer Version 9.4**

## Reference Manual

**Non-Confidential**

ARM®


*ARM 101109_0904_00 (ID080417)*

# MxScript v3.3 for Cycle Models
## Reference Manual



### Release Information

The following changes have been made to this document.

**Change History**

| Date | Issue | Confidentiality | Change |
|---|---|---|---|
| August 2017 | 0904_00 | Non-Confidential | Release with 9.4 |

### Non-Confidential 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

# Preface

This preface introduces the *MxScript v3.3 for Cycle Models Reference Manual*. It contains the following sections:



## About this document

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 with a SoC Designer system.

## 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 *SoC Designer Component-level Scripting***

Read this chapter for a description of functions that control and manage scripting simulations at the component level.

**Chapter 4 *SoC Designer Tool-level Scripting Functions***

Read this chapter for a description of SoC Designer 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. |
| **monospace** | 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:<br>`MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` |

## Terminology

The table below lists the equivalent SoC Designer terms corresponding to the SystemC terms used in the SystemC environment:

**SoC Designer terms**

| SystemC term | SoC Designer term | Description |
|---|---|---|
| module | component | The models for individual devices. For example, CPU core, memory, bus interface, and I/O. |
| port | master port | A port that generates transactions or signals |
| channel | slave port | SystemC channels are also know as sc_export. |

## Further reading

The following publications provide information about SoC Designer:

- *SoC Designer User Guide*
- *SoC Designer Tools API Reference Manual*
- *SoC Designer Installation Guide*
- *ESL API Developer's Guide*
- *SoC Designer SystemC Linking Guide*
- *SoC Designer CDP HDL Cosimulation Guide*
- *SoC Designer Standard Model Library Reference Manual*

The following publications provide reference information about the ARM® or AMBA® architecture:

- *AMBA Specification*
- *AMBA AHB Transaction Level Modeling Specification*
- *AMBA AXI Transaction Level Modeling Specification*
- *ARM Architecture Reference Manual*

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

The following publications provide additional information on simulation:

- *IEEE 1666™ SystemC Language Reference Manual*, (IEEE Standards Association)
- *SPIRIT User Guide*, Revision 1.2, SPIRIT Consortium.

# 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 and this simplifies joint development of scripts and SoC Designer components.

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:

- Simulator batch mode
- command-line on SoC Designer graphic user interface by manually opening script running or stepping through the script
- SoC Designer component (such as MxStub) call

## 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 `b */` 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 | `+ - * %`<br>`++ --`<br>`+= -= *= /= %=` | 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 | `== !=`<br>`< > <= >=` | Works on all types (including strings). Result is `bool`.<br>(The <, >, <=, and >= cannot be used with `bool` types.) |
| Logical | `&& \|\| !` | Works on bool types. Result is `bool` |
| Bitwise | `& \| ^ ~ << >>`<br>`&= \|= ^= <<= >>=` | Works on int. Result is `int`.<br>Shift operators are, unlike in C, well defined for shifts larger that the size of the integer type (64 bits). |
| Casting | `type(exp)`<br>`(type) exp` | Both C and C++ forms of casts are supported in MxScript (see *Expressions* on page 17). |
| Pointers | unitary `* &` | 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` and must only contain the digits `1` and `0`.
- 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 a 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 `elsif` 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 `true`.

```
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 component, batch-mode, and GUI scripting environments for SoC Designer. 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 update (reading and writing). |
| 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, an error message is displayed and -1 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, `fgets()` 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 SoC Designer 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_varname=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
# SoC Designer Component-level Scripting

This chapter describes functions that control and manage scripting simulations at the component level in SoC Designer. It contains the following sections:

## 3.1 Simulation control functions

This section describes functions that control the simulation process.

### 3.1.1 getCycleCount()

```
int getCycleCount()
```

Returns the current cycle that the component is in (that is, the total number of activations since last reset).

### 3.1.2 getTime()

```
int getTime(string timeUnit)
```

Returns the current time that the simulation is in.

The time is returned in the same time unit as specified in the parameter `timeUnit`. *`timeUnit`* must be one of the values listed in Table 3-1.

**Table 3-1 Time unit specifiers**

| Time unit | Description |
|---|---|
| "h" | hours |
| "m" | minutes |
| "s" | seconds |
| "ms" | milliseconds |
| "us" | microseconds |
| "ns" | nanoseconds |
| "ps" | picoseconds |
| "fs" | femtoseconds |

### 3.1.3 getSysClockPeriod()

```
int getSysClockPeriod(string timeUnit)
```

Returns the duration of one simulation cycle of the global system as set in the system properties dialog.

The time is returned in the same time unit as specified in the parameter `timeUnit`. *`timeUnit`* must be one of the values listed in Table 3-1 on page 30.

### 3.1.4 stop()

```
stop()
```

Stops the simulation.

### 3.1.5 wait(cycles)

```
wait(int numCycles)
```

This waits until the simulation has advanced *numCycles* of component cycles (referenced to the clock slave port input). If called in component cycle n, the simulation is in component cycle n+*numCycles* when the call returns (local cycle counter).

If *cycles* is 0, the statement is ignored.

To wait for exactly one cycle, use the short form that has no parameter:

```
wait()
```

### 3.1.6 wait(time)

```
wait(string duration)
```

Wait until the simulation has continued for the simulated time specified by the parameter *duration*.

If *duration* is not a multiple of the clock period the actual activation of the script takes place on the next component's cycle after the duration has expired. This effect must be taken into account especially for components that are not clocked by the SoC Designer system clock.

The format of the *duration* string consists of a decimal integer or decimal floating point number followed by the time unit. See *Time unit specifiers* on page 32.

### 3.1.7 waitUntil(cycle)

```
waitUntil(int targetCycle)
```

Wait until the simulation has reached the absolute component cycle specified in *targetCycle*.

If the current component cycle is already higher than the target component cycle, the function returns immediately.

### 3.1.8 waitUntil(time)

```
waitUntil(string targetTime)
```

Wait until the simulation reaches the time specified in *targetTime*.

If *targetTime* is not a multiple of the clock period, the actual activation of the script takes place on the next component's cycle after the time is reached. This effect must be taken into account especially for components that are not clocked by the SoC Designer system clock.

The format of the time string consists of a decimal integer or decimal floating point number followed by a time unit. See Table 3-2:

**Table 3-2 Time unit specifiers**

| Time unit | Description |
|---|---|
| "h" | hours |
| "m" | minutes |
| "s" | seconds |
| "ms" | milliseconds |
| "us" | microseconds |
| "ns" | nanoseconds |
| "ps" | picoseconds |
| "fs" | femtoseconds |

## 3.2 Accessing SoC Designer components

This section describes how to access SoC Designer components.

References to other components are done using their SoC Designer **instance id**. If a component with that name can not be found, a runtime error occurs.

An **instance id** provides a hierarchical name where the first entry specifies the filename of the loaded SoC Designer system file, followed by subsystem instance names for hierarchical systems, and ends with the instance name of the component to be accessed An example is `design.subsystemA.SRAM`. All items are separated by a dot (.).

**Note**

If the component instance name is unique throughout the system, providing just the component instance name is sufficient.

### 3.2.1 setParameter()

```
setParameter(string instname, string key, string val)
```

This function is used to set component parameters of components included in the current system:

`instname` is the instance name of the component. If a component with name *`instname`* does not exist, a runtime error occurs.

`key` specifies the name of the parameter. A runtime error occurs if the accessed component does not have a parameter of the specified name.

`val` is the new value. If the *`val`* parameter attempts to set an invalid parameter, a runtime error occurs.

### 3.2.2 getParameter()

```
string getParameter(string instname, string key)
```

This function is used to get parameters of other components:

`instname` is the instance name of the component. If a component with name *`instname`* does not exist, a runtime error occurs.

`key` specifies the name of the parameter.

### 3.2.3 CADIRegRead()

```
int CADIRegRead(string instname, string regname)
```

This function is used to read a register. If no SoC Designer component with name *instname* or no register with the name *regname* exists, a runtime error occurs.

The *regname* parameter requires a complete hierarchical name if register names are not unique throughout the component. For example, there could be multiple register groups in which registers with identical names appear. In this case, you must specify the register group as well.

For compound registers, it might be required to include the name of the parent register to specify the register by a unique identifier.

Hierarchical items are separated by dots (.) such as in:

```
SDRAM.ctrl.fid
```

### 3.2.4 CADIRegWrite()

```
CADIRegWrite(string instname, string regname, int value)
```

This function is used to write a value into a register. If no SoC Designer component with name *instname* or no register with name *regname* exists a runtime error occurs.

The *regname* parameter requires a complete hierarchical name if register names are not unique throughout a component. There could, for example, be multiple register groups in which registers with identical names appear. In this case, you must specify the register group.

For compound registers, it might be required to include the name of the parent register to specify the register by a unique identifier.

Hierarchical items are separated by dots “.” such as, for example:

```
SDRAM.ctrl.fid
```

### 3.2.5 CADIMemRead()

```
int CADIMemRead(string instname, string memspace, int address)
```

This function reads from the memory *memspace* at the address specified by the parameter *address*. When CADIMemRead is executed from a component, it reads through all internal memories and sends the request downstream to the next component, through the component master port. It returns when a valid value is found.

The following circumstances result in a runtime error:

- No component with name *instname* exists
- No memory space with name *memspace* exists
- When calling `CADIMemRead()` from a stub for a different component, specifying an *address* that is not in any address range for that component

The size of the access depends on the *Minimum Addressable Unit* (MAU) size. The MAU is the size of one word defined for that memory space

### 3.2.6 CADIMemWrite()

```
CADIMemWrite(string instname, string memspace, int address, int value)
```

This function writes the *value* to the memory space *memspace* at the address specified by parameter *address*. CADIMemWrite is propagated to all components that support full system coherent memory view, and the data value is updated throughout. Refer to the *SoC Designer User Guide* for more about full system coherent memory views.

If no component with name *instname* exists or no memory space with name *memspace* exists, a runtime error occurs.

The size of the access depends on the *Minimum Addressable Unit* (MAU) size. The MAU is the size of one word defined by that memory space.

### 3.2.7 CADIMemLoadFromFile()

```
int CADIMemLoadFromFile(string filename, int isASCIIMode,
          string instname, string memspace, int startAddr, int endAddr)
```

This function reads values from the file *filename* and writes the collected values to the memory space *memspace*, starting at the absolute address *startAddr* and stopping at *endAddr* or the end of file, whichever location occurs first.

The size of the access depends on the *Minimum Addressable Unit* (MAU) size. The MAU is the size of one word defined by memspace, the memory space.

CADIMemLoadFromFile() uses the following arguments in special ways, as follows:

**isASCIIMode**

If not 0, the file is treated as an ASCII file.

**instname**

If no component with name *instname* exists or no memory space with name *memspace* exists, a runtime error occurs.

**endAddr**

Value of the absolute address of the end of the file. Specify a sufficiently large end address. The loading of the file ends when either the `endAddress` is reached or when the end of file is reached, whichever comes first.

### 3.2.8 CADIMemStoreToFile()

```
int CADIMemStoreToFile(string filename, int isASCIIMode, string instname,
          string memspace, int startAddr, int endAddr)
```

This function reads values from the memory space *memspace* starting at address *startAddr* and stopping at *endAddr* (or the end of file if that occurs first).

All collected data is stored to the file named *filename*.

If *isASCIIMode* is 0, the file format is binary. If non-zero, the file format is ASCII.

If no component with name *instname* exists or no memory space with name *memspace* exists, a runtime error occurs. The size of the access depends on the *Minimum Addressable Unit* (MAU) size. The MAU is the size of one word as defined by that memory space.

## 3.3 SoC Designer script component port access

The commands in this section may be used only in a SoC Designer component that:

- implements transaction and signal ports
- is driven by a user-created script

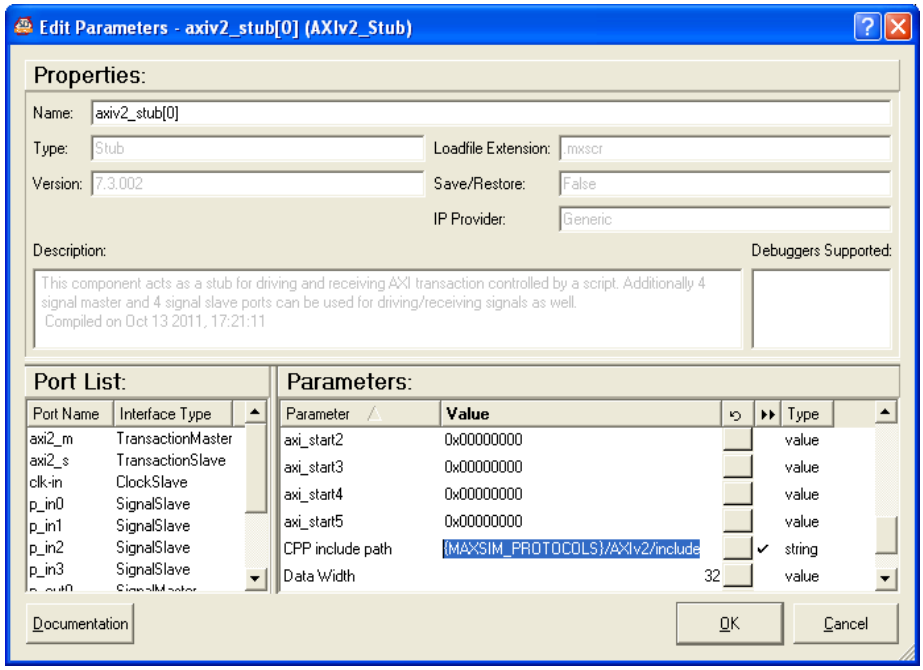
These commands are used by the ARM-provided macros; it is recommended that you use these macros without modification. To create your own macros, you may use the ARM-provided macros as a reference. These can be found in:

- Windows — `%MAXSIM_PROTOCOLS%\<protocol_name>\include\<protocol_name>_Stub_Macros.h`.
- Linux — `$MAXSIM_PROTOCOLS/<protocol_name>/include/<protocol_name>_Stub_Macros.h`.

**Setting the include path**

To use the ARM-provided macros, you must set the include path as the value for the stub component's `CPP include path` parameter. To do so, in SoC Designer Canvas access the Parameters dialog for the stub. The syntax for the include path is: `{ENVIRONMENT_VARIABLE}/<protocol_name>/include/`.

In the following figure, `CPP include path` for the AXIv2 stub component has been set to `{MAXSIM_PROTOCOLS}/AXIv2/include`. This allows you to run the macros provided for AXIv2:




### 3.3.1 getPortID()

```
int getPortID(string portName)
```

This function returns the port identification number of the port named *portName*. The naming of ports depends on the script-driven component implementation.

### 3.3.2 getPortData()

```
int getPortData(int portID, string element)
```

This function provides read access to the *element* of the port's data structure:

`portID` identifies the port. For signal ports, one element is the signal value and can be split into multiple items for data bit widths above 32. For transaction ports, this provides access to all elements of the transaction information and therefore is implementation specific.

`element` identifies the element in the port.

The value of the element is returned by the function. The SoC Designer-specific enumeration types are available for convenience, for example, `eslapi::CASI_ACCESS_READ`.

### 3.3.3 setPortData()

```
setPortData(int portID, string element, int data)
```

This function provides write access to the *element* of the port's data structure. Note that the data is applied after the port is activated via the `drivePort()` function.

`portID` identifies the port. For signal ports, one element is the signal value and can be split into multiple items for data bit widths above 32. For transaction ports, this provides access to all elements of the transaction information and therefore is implementation specific.

`element` identifies the element in the port.

`data` the value to copy into the specified element of the port's data structure. The SoC Designer specific enumeration types are available for convenience, for example, `eslapi::CASI_ACCESS_READ`.

### 3.3.4 drivePort()

```
drivePort(int portID)
drivePort(int portID, int data)
```

This function initiates the activation of a master port. For a signal master port `driveSignal()` is called. For a transaction master port `driveTransaction()`, `a read()` or `write()` function is called, depending on the CASI interface being used.

For multicycle transactions, the function returns after the transaction completes.

If called for a slave port, a runtime error occurs.

`portID` identifies the port. For signal ports, one element is the signal value and can be split into multiple items for data bit widths above 32. For transaction ports, this provides access to all elements of the transaction information and therefore is implementation specific.

`data` the value to write to or read from a transaction port, depending on the CASI interface being used.

### 3.3.5 waitEvent()

```
waitEvent(int portID)
```

This function waits until an event occurs in the signal slave port or transaction slave port with the port identification *portID*.

If *portID* equals -1, any slave port event terminates the wait. The portID that terminated the wait can be interrogated using `getEventPortID()`.

### 3.3.6 getEventPortID()

```
int getEventPortID()
```

This function returns the portID of the event that was triggered during the last cycle. This function can be called multiple times to determine if multiple events might have occurred.

If the event queue is empty a return value of -1 is returned. In the next cycle, the event queue is cleared automatically and it is therefore important to immediately check the portIDs.

## 3.4 Model library preference management

This section describes functions that manage model library preferences.

### 3.4.1 addMaxlibToPreferences()

```
void addMaxlibToPreferences(string filename, int append)
```

This function appends or prepends a specified configuration (`.conf`) file to the user preference list of configuration files in the SoC Designer Canvas and SoC Designer Simulator:

`filename` Full or relative pathname to the `.conf` file, to added to added to model library configuration file list.

`append` Takes values 1 (append) or 0 (prepend).

### 3.4.2 removeMaxlibFromPreferences()

```
void removeAllMaxlibFromPreferences(string filename, int wholeMatch)
```

This function removes a specified configuration (`.conf`) file from the user preference list of configuration files in the SoC Designer Canvas or SoC Designer Simulator:

`filename` Full or relative pathname to the `.conf` file to be removed from model library configuration file list.

`wholeMatch` Set to 1: Match the configuration (`.conf`) file by full pathname.
Set to 0: Match the specified configuration (`.conf`) file by filename, only.

### 3.4.3 removeAllMaxlibFromPreferences()

```
void removeAllMaxlibFromPreferences()
```

This function removes all configuration (`.conf`) files from the user preference list of configuration files in SoC Designer Canvas and SoC Designer Simulator.

## 3.5 Miscellaneous

This section describes functions that do not fit into the previously described categories.

### 3.5.1 getInstanceID()

```
string getInstanceID()
```

This function returns the instance id string of the script component that the script is executing on.

The instance id contains the full hierarchical name of the component instance separated by dots such as:

*system_name.instance_name* : mySystem.MxStub[0]

This function allows a single script to be executed on a set of script components and enables behavior to be differentiated based on the component's instance name.

### 3.5.2 message()

Use the message function display a message in the SoC Designer output window.

There are three forms of the message function:

`message(string msgtxt)`

This form takes a string that is to be output as information to the user.

`message(string msgtext, int type)`

This form takes an additional parameter type that can be used to display other types of messages like error conditions or debug messages. Symbolic constants for the type exist.

`message(int type, string format, …)`

This form has the additional capability to format the output similar to the ANSI C `printf()` function.

Table 3-3 describes the message types that can be specified with the *type* parameter:

**Table 3-3 Type values for message()**

| Message type | Description |
|---|---|
| CASI_MSG_FATAL_ERROR | This message type signals a fatal error. In this case the simulator displays the error message in a separate message box, and is also printed in the output window preceded with the text "FATAL ERROR:" The only way to recover from a fatal error is to load the system again. |
| CASI_MSG_ERROR | This message type indicates an error in the simulation. The simulation can be continued from this point, The message is displayed in a separate message box window, and is also printed in the output window preceded with the text "ERROR:". If the user chooses to stop the simulation on an ERROR then only way to continue is to reset the system. |

**Table 3-3 Type values for message() (continued)**

| Message type | Description |
|---|---|
| CASI_MSG_WARNING | This type classifies the message to be a warning. In this case the message string is printed in the output window preceded by the text "WARNING: ". |
| CASI_MSG_INFO | This is the default message type. The message string is simply printed in the output window. |
| CASI_MSG_DEBUG | If a message is classified as being a debug message then it is only printed if the model is executed in SoC Designer using the debug mode/version. The message is preceded with the text "DEBUG:" |

### 3.5.3 Binary Logarithm Function: ld()

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

This function returns the bit position of the most significant bit of the *arg* that is set to one. Arguments smaller than, or equal to, 0 result in a runtime error.

# Chapter 4
# SoC Designer Tool-level Scripting Functions

This chapter describes the additional functionality present in MxScript if it is invoked from SoC Designer Simulator. It has the following sections:

## 4.1 Basic simulation scripting functions

These functions provide basic simulation control.

### 4.1.1 setAppFile()

```
void setAppFile(string instanceID, string file)
```

This method allows you to specify the application files to be used by the simulation:

`instanceID` is the full instanceID of the component.

`file` specifies the file to be loaded by the component.

These files must be set before loading or resetting the system, so `setAppFile()` must be called before calling `openSystem()` or `resetSystem()`.

**Note**

The only exception is when `openSystem()` is invoked with `doInitialize` equal to 0. In this case, `setAppFile()` must be called before `initSystem()`.

If `doInitialize` is equal to one, the immediate full system initialization, including reset, is performed upon invoking `openSystem()`.

### 4.1.2 openSystem()

```
void openSystem(string filename, int doInitialize = 1)
```

This function loads a system:

`filename` The file to load.

The file name extension can be `.MXP`, `.MXE`, `.MXS`, or `.MXR`.

`doInitialize` specifies whether the system should be initialized for simulation or not. By default the *`doInitialize`* parameter is set to 1 so that the opened system is initialized and is immediately ready for simulation.

However if the *`doInitialize`* parameter is set to 0 (possible for MXP files only), the system is not initialized automatically. This makes it possible to set design-time parameters such as buffer size or memory map from a script.

After setting the parameters, `initSystem()` must be called before the simulation can be started.

### 4.1.3 initSystem()

```
void initSystem()
```

This function initializes a system that was loaded with `openSystem()` but has not yet been initialized. For example:

1. use `openSystem()` with doInitialize equal to 0 to open a system
2. call `setParameter()` to set any required parameters
3. call `initSystem()`

### 4.1.4 closeSystem()

```
void closeSystem()
```

Use this function to close the current system.

### 4.1.5 resetSystem()

```
void resetSystem(string level)
```

This causes the currently loaded simulation to be reset:

`level` is the reset level.

Use "`SOFT`" or "`HARD`" for the *`level`* argument to specify soft or hard resets.

### 4.1.6 debuggerHost()

```
void debuggerHost()
```

This command launches the host level debugger. The debugger is MSVC under Win32 or ddd/gdb under Linux.

### 4.1.7 debuggerComponent()

```
void debuggerComponent(string instanceID)
```

This launches the default debugger for the specified component:

`instanceID` is the full instanceID of the component.

### 4.1.8 systemLoaded()

```
int systemLoaded()
```

If there is a currently loaded system, the function returns `true` (1).

### 4.1.9 openCheckPoint()

```
void openCheckPoint(string filename)
```

Use this method to open a checkpoint in the named file:

`filename` name of the file to hold the checkpoint. It must have the extension .mxc.

### 4.1.10 saveCheckPoint()

```
void saveCheckPoint(string workspace, string filename, string description)
```

Use this method to save a checkpoint in the named file:

`workspace` name of the workspace in which to place the checkpoint file.

`filename` name of the file to hold the checkpoint. It must have the extension .mxc.

`description` description of the checkpoint data.

### 4.1.11 saveSimulation()

```
void saveSimulation(string filename, int state)
```

Use this method to save a simulation, and optionally the simulation state, in the named file:

`filename` name of the file to hold the saved simulation.

`state` integer value controlling whether state information is saved:

- `0` exclude state information and save to an .mxs file
- `1` include state information and save to an .mxr file

### 4.1.12 enableExternalProfiling()

```
int enableExternalProfiling(value)
```

Enables or disables the writing of profiling and transaction data to the external profile database for the current session of SoC Designer Simulator. It overrides the state of the preference "Use External Profile Database" but does not change the preference. This command must be called before the system is opened in order to take effect.

`value` integer value controlling whether data is written to the external profile database:

- `0` disables writing of data to the external profile database
- `1` enables writing of data to the external profile database

For more about the external profiling database, refer to the *SoC Designer System Analyzer User Guide*.

### 4.1.13 getDebugState()

```
bool getDebugState(string instanceID)
```

This returns 0 if the target is not at a debuggable point. It returns a non-zero value otherwise:

`instanceID` is the full instanceID of the component.

### 4.1.14 runToDebuggablePoint()

```
void runToDebuggablePoint(string instanceID)
```

This runs the simulations until the target gets to a debuggable point. Note the simulation may halt due to other reasons (e.g., breakpoints set elsewhere in the system). For this reason, the user should check the debuggable state after issuing this command:

`instanceID` is the full instanceID of the component.

### 4.1.15 addMaxlib()

```
void addMaxlib(string filename)
```

This function enables specification of an additional `maxlib.conf` file.

### 4.1.16 getSysClockUnit()

**Note**

This function is reserved for future use and is not supported in the current release.

```
string getSysClockUnit()
```

This method returns the clock period unit of the current system. The returned value is one of strings listed in Table 4-1.

**Table 4-1 Time unit specifiers**

| Time unit | Description |
|---|---|
| "h" | hours |
| "m" | minutes |
| "s" | seconds |
| "ms" | milliseconds |
| "us" | microseconds |
| "ns" | nanoseconds |
| "ps" | picoseconds |
| "fs" | femtoseconds |

## 4.2 Runtime simulation control functions

These functions control the simulation environment.

### 4.2.1 run()

```
void run()
```

Calling this function has the same effect as clicking the **Run** button in Simulator.

The simulation runs until a breakpoint is hit, the simulation is stopped manually via the Stop button in Simulator, or the simulation is stopped by another means, such as a different script.

### 4.2.2 runUntil()

```
void runUntil(int cycle)
```

This function advances the simulation to the specified cycle. If the specified cycle has already been reached or passed, no action is taken.

### 4.2.3 step()

```
void step(int cycles)
```

Calling this function has the same effect as clicking the **Step** button in Simulator.

This function advances the simulation the number of cycles specified in *`cycles`*.

### 4.2.4 animate()

```
void animate(int cycles, int delay)
```

This function makes available the same functionality found in the Simulator Step N dialog:

`cycles` specifies the number of cycles the simulation should advance at each step.

`delay` specifies the time interval between steps. The parameter value is in ms*10. Valid ranges are from 1 - 99.

### 4.2.5 startCADIServer()

```
void startCADIServer()
```

This function starts the CADI servers and enables external debuggers to connect to the simulation and inspect or control the simulation. The default condition is that the servers are not started.

### 4.2.6 waitForCADIServerExit()

```
void waitForCADIServerExit(int exit On Disconnect)
```

This function pauses the script until an external debugger requests simulation shutdown:

`exitOnDisconnect`

specifies how the external debugger generates the shutdown command:

- If the parameter is 0, execution pauses until the debugger emits an explicit shutdown command.
- If the parameter is 1, execution pauses until the debugger disconnects.

The script resumes execution after the request. After this call returns, the servers are inactive. The script might choose to restart them at a later time by calling `startCADIServer()`. This enables debuggers to gain exclusive control over the simulation without interference from the execution of the script itself.

## 4.3 Connection querying functions

These functions enable obtaining information on a connection.

### 4.3.1 connGetID()

```
int connGetID(string masterComponent, string masterPort, string slaveComponent,
              string slavePort)
```

This function returns the connection id for the connection between the specified components. If the specified connection does not exist in the system, -1 is returned.

### 4.3.2 connIsValidID()

```
int connIsValidID(int id)
```

This function returns 1 or 0 depending upon whether or not the specified *id* is valid for the current simulation.

## 4.4 Monitor functions

These functions enable placing a monitor on a connection.

### 4.4.1 monitorAddConn()

```
int monitorAddConn(int connID, string filename)
```

This function places a monitor on the connection specified by the connection identifier `connID`. `filename` indicates where to write the monitor data.

### 4.4.2 monitorRemConn()

```
void monitorRemConn(int connID
```

This function removes the monitor specified by the connection identifier `connID`.

## 4.5 CASI component-specific functions

These functions relate to SoC Designer components.

### 4.5.1 getProperty()

```
string getProperty(string instaceID, string property)
```

This function enables reading the value of the specified property from the component specified in *`instanceID`*.

### 4.5.2 traceSetFile()

```
void traceSetFile(string filename)
```

This function sets the name of the file to trace to.

### 4.5.3 traceGetFile()

```
string traceGetFile()
```

This function returns the name of the file currently set for tracing.

### 4.5.4 traceSetPeriod()

```
void traceSetPeriod(int time, string unit)
```

Use this function to adjust the time scale used in the VCD file:

`time` is the duration.

`unit` can be in one of the values listed in Table 4-2:

**Table 4-2 Time unit specifiers**

| Time unit | Description |
|---|---|
| "s" | seconds |
| "ms" | milliseconds |
| "us" | microseconds |
| "ns" | nanoseconds |
| "ps" | picoseconds |
| "fs" | femtoseconds |

For a component that takes 5ns to complete a cycle, use:

```
traceSetPeriod(s, "ns");
```

### 4.5.5 traceRestartReset()

```
void traceRestartReset()
```

This function defines the behavior of the trace when the simulation is reset. This causes the trace to restart upon reset.

### 4.5.6 traceContReset()

```
void traceContReset()
```

This function defines the behavior of the trace when the simulation is reset.

It causes the trace to continue upon reset. This is the default behavior.

### 4.5.7 traceAddConn()

```
int traceAddConn(int id)
```

This function places a trace on the connection specified by the connection *id*. The trace format is VCD.

### 4.5.8 traceAddConnFile()

```
int traceAddConn(int id, string vcdFile)
```

This function places a trace on the connection specified by the connection *id*. The trace is dumped to the path *vcdFile*.

### 4.5.9 traceAddReg()

```
int traceAddReg(string instanceID, string reg)
int traceAddReg(string instanceID, int reg)
```

Use these functions set tracing for the register location in the specified component:

`instanceID` is the full instanceID of the component.

`reg` specifies the register by the name (using a string *reg*) or by its id (using an integer *reg*).

### 4.5.10 traceAddRegFile()

```
int traceAddRegFile(string instanceID, string reg, string vcdFile)
```

```
int traceAddRegFile(string instanceID, int reg, string vcdFile)
```

Use these functions to set tracing for the register location in the specified component:

`instanceID` is the full instanceID of the component.

`reg` specifies the register by the name (using a string `reg`) or by its id (using an integer `reg`).

`vcdFile` output vcd file path.

### 4.5.11 traceRemove()

```
void traceRemove(int id)
```

This function removes the trace specified by the trace *id*.

### 4.5.12 traceSetSigBits()

```
void traceSetSigBits(int id, int bits)
```

Use this function to specify how many bits of the signal value are actually traced. The trace is specified by the *id* value returned by the function *traceAddConn()* on page 51.

**Note**

If 0 is passed as the value for `bits`, the signal is not traced.

### 4.5.13 traceSetTransBits()

```
void traceSetTransBits(int id, int address, int value, int control)
```

Use this function to specify how many bits of the control field values are actually traced:

`id` The trace is specified by the *id* value returned by the function *traceAddConn()* on page 51:

`address` specifies the number of bits to trace for the address field of a connection.

`value` specifies the number of bits to trace for the value field of a connection.

`control` specifies the number of bits to trace for the control field of a connection.

**Note**

If 0 is passed as the value for any field, that field is not traced.

## 4.6 Profiling functions

These functions are used to profile information from a simulated run of a component.

### 4.6.1 enableProfiling()

**Note**

This function is deprecated and is to be removed in a future release.

```
void enableProfiling(string component_name, string stream_name,
                     string file_name, int compression_factor)
```

This function writes the summarized hardware profiling data that has been collected by the CAPI interfaces into a *Comma Separated Values* (CSV) file.

`component_name`
: name of the component to profile.

`stream_name` is the id for the profiling stream. (See *profilingStreamGetID()*.)

`file_name` name of the CSV file to hold the summarized profile data.

`compression_factor`
: specifies the number of cycles for which each new entry is generated.

**Note**

Symbolic events are summarized over several cycles to reduce data size.

Non-symbolic channels that cannot be compressed, address channels for example, are not collected.

### 4.6.2 profilingStreamGetID()

```
int profilingStreamGetID(string objectName, string streamName)
```

This function returns a unique integer that represents the steam id. The return value uniquely identifies the I/O stream and profiled object and is used by profiling functions to identify a stream.

`objectName` object name of the component to be profiled.

`streamName` specifies the I/O stream that contains profiling data.
If `streamName` is `"Software"`, the id of the software profiling stream is returned.

### 4.6.3 profilingStreamEnable()

```
void profilingStreamEnable(int streamID)
```

This function enables a profiling stream.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

This function is equivalent to checking the corresponding **Enable** check box in the Profiling Manager window.

For software profiling, ARM recommends enabling the stream at cycle 0.

### 4.6.4 profilingStreamEnableCSV()

```
void profilingStreamEnableCSV(int streamID)
```

This function enables a profiling stream that outputs profiling values into a *Comma Separated Values* (CSV) file.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

This function is equivalent to checking the corresponding **Dump to CSV** check box in the Profiling Manager window.

For software profiling streams, the CSV file contains a list of all function activation and return points.

### 4.6.5 profilingStreamDisable()

```
void profilingStreamDisable(int streamID)
```

This function disables a profiling stream.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

This function is equivalent to unchecking the corresponding **Enable** check box in the Profiling Manager window.

### 4.6.6 profilingStreamDisableCSV()

```
void profilingStreamDisableCSV(int streamID)
```

This function disables a profiling stream that outputs profiling values into a *Comma Separated Values* (CSV) file.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

This function is equivalent to unchecking the corresponding **Dump to CSV** check box in the Profiling Manager window.

### 4.6.7 profilingStreamSetAttribute()

```
void profilingStreamSetAttribute(int streamID, string attributeName,
                                 string attributeValue)
```

This function sets the value of an attribute of a profiling stream that writes to a CSV file.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

`attributeName`

is the name of the attribute to set:

`"csv_filename"`

`csv_filename` is the name of the CSV file to which the profiling stream is written.

`"csv_compression_factor"`

`csv_compression_factor` specifies the number of cycles for which each added new entry in the CSV file is generated. The default value for `csv_compression_factor` is 1000.

**Note**

Symbolic events are summarized over several cycles to reduce data size. Non-symbolic channels that cannot be compressed, address channels for example, are not collected, summarized, or aggregated.

`attributeValue`

new value for the attribute.

### 4.6.8 profilingStreamGetAttribute()

```
string profilingStreamGetAttribute(int streamID, string attributeName)
```

This function returns the value of an attribute of a profiling stream that writes to a CSV file.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

`attributeName`

is the name of the attribute to return. This can be one of:

- `"csv_filename"`
- `"csv_compression_factor"`.

See also *profilingStreamSetAttribute()* on page 54.

### 4.6.9 profilingSoftwareSnapshotSummaries()

```
void profilingSoftwareSnapshotSummaries(int streamID, string fileName)
```

This function writes a summary of all function calls to the specified file.

`streamID` is the id for the profiling stream. See *profilingStreamGetID()* on page 53.

`fileName` is the name of the output file.

The functionality is equivalent to selecting the **Save Summary to File** context menu option from the Summary pane of the Software Profiling View.

### 4.6.10 profilerAddConn()

```
int profilerAddConn(int id)
```

This function places a profiler on the connection specified by the connection *id* and profiling is performed on all streams. The profile file format is CSV.

## 4.7 Breakpoint functions

These functions relate to setting breakpoints.

### 4.7.1 bpAddConn()

```
int bpAddConn(int id)
```

This function places a breakpoint on the connection specified in *id*.

It returns the breakpoint id for the newly created breakpoint.

### 4.7.2 bpAddReg()

```
int bpAddReg(string instanceID, string reg)
int bpAddReg(string instanceID, int reg)
```

This function places a breakpoint on the register in the component:

`instanceID` is the full instanceID of the component.

`reg` specifies the register name as a string or its id as an integer.

### 4.7.3 bpAddMem()

```
int bpAddMem(string instanceID, int address, string space)
int bpAddMem(string instanceID, int address, int space)
```

This function places a breakpoint on the specified address and space:

`address` is the specified address in the component

`instanceID` is the component instance identifier.

`space` can either be the space name or space id.

### 4.7.4 bpAdd()

```
int bpAdd(string instanceID, int pc)

int bpAdd(string instanceID, int address, string space)

int bpAdd(string instanceID, int address, int space)
```

This function adds a disassembly breakpoint at the specified address:

`pc` is the specified *pc* address in the component

`instanceID` is the component instance identifier.

`space` is the the memory space name or memory space id.

**Note**

This function can only be used for components that support loading and running a target application, such as processor cores.

### 4.7.5 bpRemove()

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

This function removes the breakpoint with the specified *id*.

### 4.7.6 bpRemoveAll()

```
void bpRemoveAll()
```

This function removes all breakpoints in the current system.

### 4.7.7 bpEnable()

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

This function enables the breakpoint specified with *id*.

### 4.7.8 bpEnableAll()

```
void bpEnableAll()
```

This function enables all breakpoints in the current system.

### 4.7.9 bpDisable()

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

This function disables the breakpoint specified with *id*.

### 4.7.10 bpDisableAll()

```
void bpDisableAll()
```

This function disables all breakpoints in the current system.

### 4.7.11 bpInfoEnabled()

```
int bpInfoEnabled(int id)
```

This function returns 1 or 0 depending on if the specified breakpoint is enabled or disabled.

### 4.7.12 bpInfoExists()

```
int bpInfoExists(int id)
```

This function returns 1 if the specified breakpoint *id* actually exists, otherwise 0 is returned.

### 4.7.13 bpInfoLastHit()

```
int bpInfoLastHit()
```

This function returns the breakpoint id of the last-hit breakpoint. If this function returns a positive value, it must be a breakpoint *id*. A return value of "-1" signifies that there are no more breakpoints.

Note that multiple calls to this function at a point in simulation may return multiple breakpoint IDs. This is possible in the case of multi-core systems, where multiple CPUs may have hit breakpoints within the last run.

### 4.7.14 bpSetCond()

```
void bpSetCond(int id, string condition, int value)
```

This function enables setting a breakpoint condition:

`id` specifies the breakpoint to modify

`condition` is one of: "`ANY`", "`EQ`", "`NE`", "`GT`", "`LT`", "`LE`", "`GE`".

**Note**

It is also possible to use the C-operator, in quotes, instead.

`value` is the comparison value for the specified condition.

### 4.7.15 bpConnSetAct()

```
void bpConnSetAct(int id, string action)
```

Use this function set the trigger action for connection breakpoints:

`id` specifies the breakpoint.

`action` can be one of: "`ANY`", "`READSIG`", "`DRIVESIG`", "`READ`", "`WRITE`", "`READREQ`", "`WRITEREQ`".

### 4.7.16 bpConnSetCond()

```
void bpConnSetCond(int id, string field, string condition, int value)
void bpConnSetCond(int id, string condition_expression)
```

**Caution**

The variant void `bpConnSetCond(int id, string field, string condition, int value)` is deprecated and is to be made obsolete in a future version.

`void bpConnSetCond(int id, string condition_expression)` is used instead.

Use this function with connection breakpoints. A breakpoint can be configured to only break if the condition specified above is true, where:

`id` specifies the breakpoint.

`field` can be one of: "`RETURN`", "`ADDRESS`", "`VALUE`", "`EXTVALUE`", "`CONTROL`"

`condition` can be one of: "`ANY`", "`EQ`", "`NE`", "`GT`", "`LT`", "`LE`", "`GE`"

`condition_expression`

is an expression that combines fields, logical operators, and values into a simple or compound logical expression.

These expressions allow the following logical operators:

`&&`, `||`, `==`, `<`, `>`, `<=`, and `>=`.

The mask operator `&` can be used to select specific bits in a field.

The following is an example of a compound `condition_expression`:

```
"( ADDRESS == 0x100 ) || ( VALUE & 0xff ==  0 ) && ( RETURN ==  0 )"
```

`value` is the comparison value for the specified condition.

`example`

```
// test breakpoint
int bpt_id = bpAddConn(conn_id);
message("INFO", "breakpoint id: %d", bpt_id);
bpConnSetCond(bpt_id, "(((AWVALID == 0x1) && (AWADDR==
0xfffffffffffff0cc))||((ARVALID == 0x1) && (ARADDR== 0xfffffffffffff0cc))||((WVALID
== 0x1) && (WADDR == 0xfffffffffffff0cc))||((RVALID  == 0x1) && (RADDR ==
0xfffffffffffff0cc))||((BVALID  == 0x1) && (BADDR == 0xfffffffffffff0cc)))");
```

## 4.8 Logging control functions

These functions relate to writing messages to log files.

### 4.8.1 setLogFile()

```
void setLogFile(string filename)
```

This function sets the name and location of the log file.

### 4.8.2 enableLogging()

```
void enableLogging()
```

This function enables logging.

### 4.8.3 disableLogging()

```
void disableLogging()
```

This function disables logging.

### 4.8.4 loggingEnabled()

```
int loggingEnabled()
```

This function returns 1 if logging is enabled. If not, 0 is returned.

### 4.8.5 getLogFile()

```
string getLogFile()
```

This function returns the file name currently set as the log file.

## 4.9 Output message functions

These functions display messages or control how messages are displayed.

### 4.9.1 message(...)

```
void message(string msg)
void message(string msg, string type)
void message(string type, string msg)
void message(string type, string format, arglist)
```

These methods provide a way for a script to access the message functionality in SoC Designer:

`msg` specifies the text sent to the Simulator output handler enabling it to also be logged.

`type` can be one of the following values:

- `"FATAL_ERROR"`
- `"ERROR"`
- `"WARNING"`
- `"INFO"`
- `"DEBUG"`

The variant with the *`arglist`* parameter can be used just like the `printf()` function to provide formatted message output.

### 4.9.2 setVerbosity()

```
void setVerbosity(string type)
```

This method provides a way to disable the internal messages printed by SoC Designer and the messages that are printed by `message()` calls from components. The output of `message()` function calls from the script itself is not disabled.

`type` can be one of the following values:

- `"FATAL_ERROR"`
- `"ERROR"`
- `"WARNING"`
- `"INFO"`
- `"DEBUG"`

This function disables all messages with a lower priority than the one specified in the `setVerbosity()` argument. The order of priorities is as in the above list, with `FATAL_ERROR` the highest priority and `DEBUG` the lowest one.

For example, `setVerbosity("WARNING")` disables messages with the following types:

- `INFO` (`MX_MSG_INFO`)
- `DEBUG` (`MX_MSG_DEBUG`)

## 4.10 CoDesign Package control functions

The SoC Designer CoDesign Package HDL Cosimulation supports different cosimulation modes. Use the functions in this section to check or configure the HDL cosimulation mode.

### 4.10.1 getCurrentHDLCosimMode()

```
int getCurrenHDLCosimMode()
```

This method returns the current mode that HDL simulation is running in:

`0` Cosimulation is running in *Normal* mode

`1` Cosimulation is running in *Turbo Clocked* mode

`2` Cosimulation is running in *Turbo Unclocked* mode

`-1` Error, SoC Designer Simulator is not running cosimulation

### 4.10.2 configureHDLCosimMode(...)

```
int configureHDLCosimMode(int mode)
```

This method sets the HDL cosimulation mode to the mode specified in the argument:

`mode` specifies the new mode to switch to. The mode options are described in *getCurrentHDLCosimMode()*.

`return value` can be one of the following values:

- Non-zero if the mode change was successful
- Zero if the mode change was not successful or if sdsim is not currently running HDL cosimulation.