IFS User's Guide

March 2010

Contents

1 Getting Started: Simulation with IFS

1.1 What Is IFS?

As shown in Figure 1.1 an IFS-based simulation consists of the following parts:

- The design under test (DUT) (in either SystemC or VHDL)
- IFS modules (in either SystemC or VHDL)
- IFS Controller
- A testbench in SystemC that instantiates and connects the DUT, the IFS modules, and the IFS controller
- An IFS script containing commands for the IFS modules

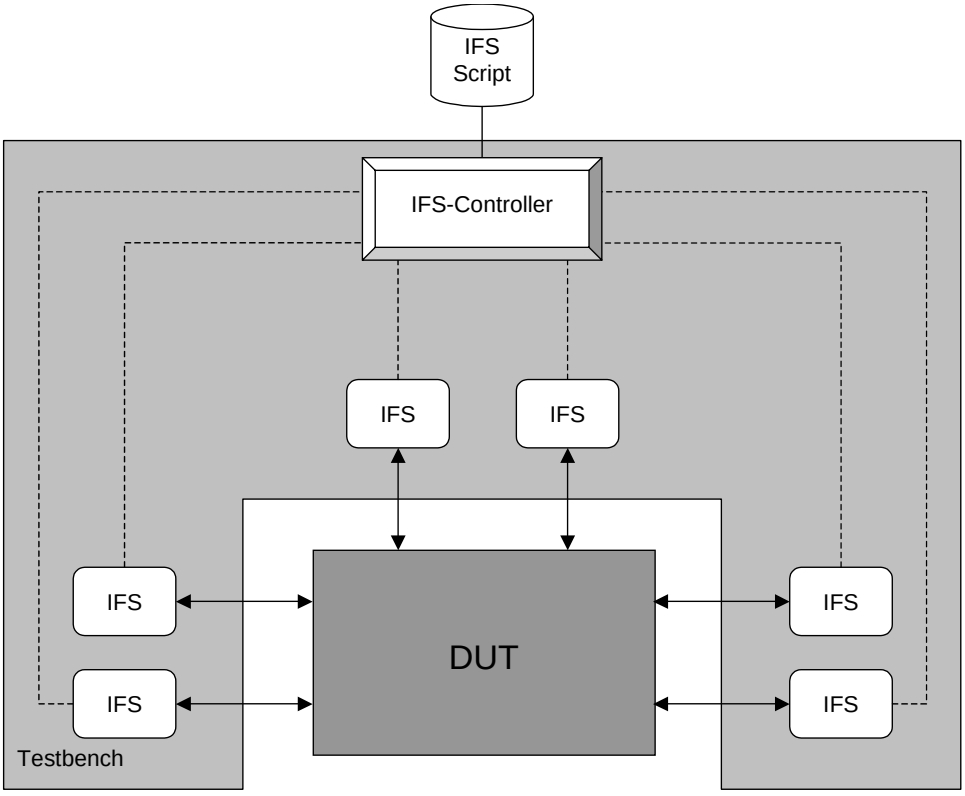

Figure 1.1: Structure of an IFS-based testbench

IFS modules are the building blocks of a testbench. An IFS module typically provides some kind of standard interface. For example, such an IFS module could be a simple clock generator or an I²S interface. Each IFS module provides user-defined commands which control the IFS module's behaviour. For example, a clock module may provide a command which sets the clock frequency and the I²S interface may provide a write command which transmits a value given as parameter to the command.

These commands are invoked from a so-called IFS script. The IFS controller reads these commands, does some pre-processing, and dispatches the commands to the IFS modules. In other words, it is the IFS script which defines a simulation – the testbench is just composed of programmable modules.

1.2 IFS script

Figure 1.2 shows a simple IFS script. An IFS script is processed line-by-line, i.e. each line can contain at most one command. However, a command may be stretched across multiple lines by ending each intermediate line by a backslash '\'. A normal command consists of three parts: the name of the receiving module (e.g. `module2`), the name of the command (e.g. `Write`), and a list of space-separated parameters for the command (e.g. `2 true`). A module's name is the string given as the constructor argument during the instantiation of that module. The command name is the string specified in the `IFS_REGISTER_COMMAND(…)` macro. For a complete documentation of a script's syntax and its semantics refer to Chapter 3.

```
-- Interrupt handler for interrupt 1
#isr 1
  module1 print "Handler 1: simtime should be 100 ns"
  module2 print "Handler 1: simtime should be 100 ns"

  module1 WAIT deltime 400 ns
  module2 WAIT simtime 500 ns
#end

-- Begin of the script body
module1 print "Simtime should be 0"
module2 print "Simtime should be 0"

-- Enable interrupt 1
IFS IRQ MASK #1 1

module1 WAIT simtime 1000 ns
module2 WAIT deltime 100 ns

module2 print "Triggering interrupt 1"
module2 Write 2 true
module2 WAIT simtime 800 ns

module1 print "Simtime should be 1000 ns"
module2 print "Simtime should be 800 ns"
```

Figure 1.2: An exemplary IFS script.

The script in Figure 1.2 starts with the definition of the interrupt handler for interrupt number 1. The body of the script starts with two print commands which cause module1 and module2 to print the given string. Note that print, as well as for example WAIT, is a predefined command which does not have to be implemented by the user, but which is handled by the

controller. The only user-defined module command in this example is the line "module2 Write 2 true". This command causes the method void SimpleIO::Write(list<string> parameters) to be executed with parameters being a list containing the two strings "2" and "true" .

Note that although the order of the commands in the IFS script suggests that this order is equal to the execution order of the commands it is not necessarily the case. The reason is the underlying execution mechanism. All the IFS modules independently query for new commands from the script and execute them until they are finished. Therefore each IFS module can be thought as having its own "instruction pointer". Even if one IFS module executes a command which takes a certain amount of time, other IFS modules can execute their commands which may be located below that command and effectively "overtake" each other. However, there is a possibility to synchronise the execution of commands. For detailed information on this topic refer to Chapter 0.

2 Simulation Semantics

This chapter gives an overview of the basic execution mechanism and provides references to more detailed descriptions.

2.1 Execution of Module Commands

Each IFS module contains its own process with a command loop. When a module command from a script is executed and the corresponding method from the IFS module is called, it is done from the module's command loop via the controller. Each command loop permanently queries the controller for new commands. The controller steps through the script and looks for the next command of the requesting module and invokes the corresponding method. As a consequence of this execution process, a module can only process one command at a time and a `wait` statement within a command method will suspend the command loop, thereby preventing the module from fetching new commands. However, it is possible for a command to activate other internal processes within the module that work in parallel to the execution of successive commands.

Another consequence of the underlying execution mechanism is that each IFS module has its own 'instruction pointer', i.e. its own location in the script from where the next command is fetched. For example in the script shown Figure 2.2 the order of the `print` statements depends on the (simulated) execution time of `Cmd1`, `Cmd3` and `Cmd4`: since `moduleA` will not execute its `print` statement until `Cmd1` is finished, `moduleB` may execute its `print` statement before `moduleA` – namely when `Cmd3` and `Cmd4` take less time than `Cmd1`. Note that the only way for a command to consume time is to call a `wait(...)` function of the SystemC kernel. Also note that the order of execution of `moduleA`'s and `moduleB`'s first command is not determined, but depends on the SystemC scheduler. For the example shown in Figure 2.2 this means that it is not determined whether `Cmd1` or `Cmd3` is executed first. The same is true for commands following `SYNC` or `WAIT` commands.

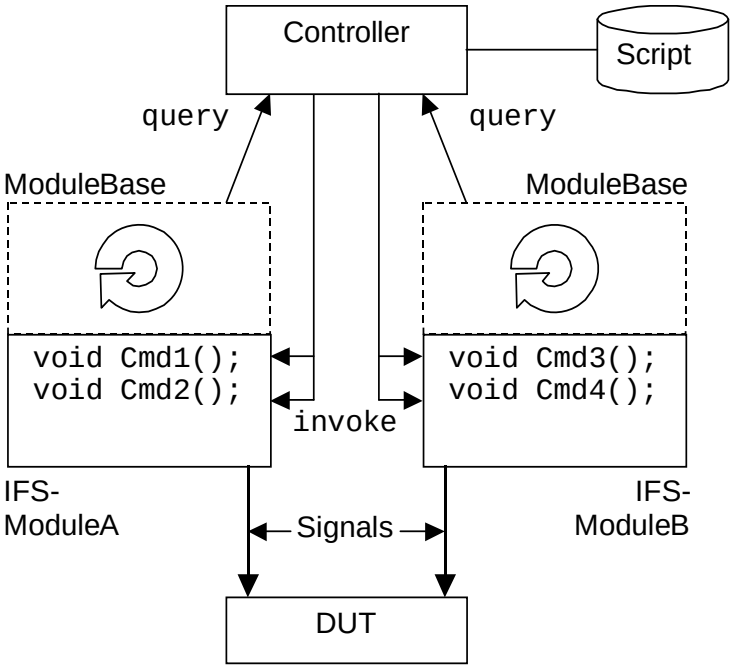

Figure 2.1: Dispatching and Execution of IFS modules

Commands where the target module is `ALL` or `IFS` (both are synonyms), such as the command: `IFS IRQ MASK #1 1` in Figure 1.2, are actually executed by every module. In special cases `IFS` can also address a module. Therefore it is advisable to place `SYNC` statements around such commands in order to ensure a consistent environment for all modules, especially when a certain setting is changed multiple times during the execution of the script. Otherwise it may happen that some 'faster modules' activate the new settings and "some slower" modules set them back because they arrive at the location of the first change of the settings.

```
moduleA Cmd1 1 2 3
moduleA print "Hello from moduleA"
moduleB Cmd3 XYZ
moduleB Cmd4
moduleB print "Hello from moduleB"
moduleA Cmd2
```

Figure 2.2: An exemplary IFS script.

2.2 Interrupts

Interrupts can be triggered by IFS modules via the method

```
void interruptOccurred ( InterruptID occurredInterrupt )
```

If an enabled interrupt is triggered, all modules branch to the corresponding interrupt handler in the IFS script (see Figure 1.2).

By means of a predefined module command it is possible to prevent that all modules branch to the interrupt handler. Actually, the occurrence of an interrupt will not interrupt the execution of a module's command (except for `WAIT simtime`, `WAIT deltime` and `SYNC`), but cause a module to fetch its next instruction from the interrupt handler. Since there is no implicit `SYNC` at the beginning or the end of an interrupt handler, modules may enter the corresponding interrupt handler at different times. It may happen that some modules are already finished with the execution of the interrupt handler whereas other modules did not even begin with the execution of the interrupt handler. When the end of an interrupt handler is reached (or a `#break` statement), a module continues its execution at the point where it was interrupted.

Note that interrupts cannot be interrupted. For further details refer to Section 3.9.

2.3 Substitutions

Three kinds of substitutions are applied before a script line is actually executed: variable substitution, random expression substitution, and arithmetic expression substitution – in this sequence. Examples are shown in Figure 2.3. Variables can be either loop variables or global variables. Their substitution is a simple string replacement; hence variables can hold any string. Arithmetic expressions can only contain operations on numerical values. Boolean values in relational expressions are also handled as numbers, i.e. using 0 and 1. Finally, there are two different kinds of random expressions: ranges and lists of arbitrary tokens. When combining these three types of expressions (variables, random expressions, and arithmetic expressions) it is important to keep the substitution order in mind. For example, it is not possible to use arithmetic expressions in ranges of random expressions due to the substitution order. However, it would be possible to assign the arithmetic expression to a variable and then use this variable as a range.

```
module1 ASSIGN myVariable = "Hello, world"
module1 print The value of myVariable is : #myVariable
module1 print The result of 3*2 is : $(3*2)
module1 print A random number between 1 and 10 : ${ 1:10 }
```

Figure 2.3: Substitutions in IFS script

3 IFS Command Script

This chapter describes the syntax and semantics of the IFS scripts. The script syntax of the SystemC implementation of IFS is (should be) fully backwards compatible to the VHDL implementation of IFS, but provides additional features such as:

- Random expressions
- Nested loops

- Unified checking of expected values
- Arbitrary length of module names
- Static consistency checks

3.1 Notation

The syntax is described below, using a mixture of regular expressions and the EBNF (Extended Backus-Naur Form). This section will explain the semantics of the used notation.

For example, the expression

Script ::= <InterruptHandler_List> <Command_List> <InterruptHandler_List>

means: A script is a sequence of an interrupt handler list followed by a command list followed by an interrupt handler list.

Symbols printed in *italics* are non-terminal symbols. The left hand side of a so called production rule is always a single non-terminal symbol. Occurrences of non-terminal symbols on the right hand side are always enclosed in angled brackets (<>), meaning they could be replaced by the right hand side of the production rule that has this non-terminal symbol on its left hand side. For example, the non-terminal *<Command_List>* could be replaced by the right hand side of this production rule:

Command_List ::= (<Command> <Newline>)+

Parentheses are used to group sequences of commands. The plus operator means that the preceding symbol (in this case the sequence of *<Command>* *<Newline>*) can be repeated one to n times. An asterisk (*) means that the preceding symbol can be repeated zero to n times.

Alternatives are expressed by a vertical bar (|). For example:

InterruptCommand ::= *<IrqEnableClause>* |
<IrqPriorityClause>

So an *InterruptCommand* may be either an *IrqEnableClause* or an *IrqPriorityClause*.

In addition to non-terminal symbols there are so called terminal symbols. These terminal symbols are the text that is actually written in the script. Terminal symbols are printed in `underlined courier font`.

ALLCommand ::= `SYNC` *(* *<ModuleName>** | `ALL` *)* |
`CHECK` *<ModuleCommand>* *<ReferenceValue>*

Hence, an *ALLCommand* is either the word ‘SYNC’ followed by a (possibly empty) list of module names or the word ‘ALL’, or it is the word ‘CHECK’ followed by a *ModuleCommand* and a *ReferenceValue*.

3.2 Lexical Elements

The basic lexical elements of a command-script, i.e. the terminal symbols of its grammar, are called ‘tokens’. Tokens are sequences of characters that are separated from each other by either white-spaces (space, newline, or tab) or special non-alphabetic characters such as "`:,()${}`". Additional separation characters exist in expressions (see Chapter 3.11). Roughly speaking, a token is a word surrounded by spaces or non-alphabetic characters or it is the non-alphabetic character itself. If a token is required to contain characters, which normally would lead to a separation into multiple tokens, it has to be surrounded by quotation marks. In order to quote a quotation mark, it has to be preceded by a backslash.

Examples:

`"Hello World!"`	One token
`"Hello , World"`	One token
`"Hello \" World"`	One token
`Hello World!`	Two tokens
`Hello World!`	Two tokens
`Hello,World?`	Three tokens
`Hello, World.`	Three tokens
`Hello ,,World`	Four tokens

3.3 Pre-Processing

A pre-processing of the script is performed before it is actually parsed. During this phase comments and empty lines are removed and include directives are resolved.

3.3.1 Comments

Comments are indicated by ‘--‘, i.e., two minuses. All characters in a line after the comment delimiter are ignored. In expressions (see Chapter 3.11) a minus ‘-‘ is regarded as a sign and not as the introduction of a comment.

3.3.2 Inclusion of other Scripts

If the first token of a line is either `#inc` or `#include` the line will be replaced by the contents of the file whose path follows as the second token. If the path is relative, it is interpreted as being relative to the location where the executable has been started.

3.4 Values

Table 3.1 shows the lexical structure of some basic tokens.

*Identifier ::= <Letter> (<Letter> | <Digit> | _)** **Note:** The third alternative is the terminal '_' (underscore).
Letter ::= a-z | A-Z
Digit ::= 0-9
PositiveInteger **Note**: Follows the common notion of positive integer numbers. Leading zeros are allowed and the zero-value[1] itself is not allowed. The leading '+' (plus) is optional.
NegativeInteger **Note**: Follows the common notion of negative integers with a preceding '-' (minus). Leading zeros are allowed and the zero-value[2] itself is not allowed.
Integer ::= <NonNegativeInteger> | <NegativeInteger>
NonNegativeInteger ::= <PositiveInteger> | 0
Time ::= <IfsFloatingPointNumber> [fs | ps | ns | us | ms | s *]*

Table 3.1: Lexical structure of identifiers and numbers

3.5 General Script Structure

Table 3.2 shows the overall structure of an IFS script. Basically, it states that interrupt handlers must be at the beginning or at the end of the script (or both), but not within the body (the sequence of "normal" commands). An example can be found in Figure 1.2.

Some general remarks:

- Identifiers and keywords are handled case-sensitive.
- The non-terminals ending on the postfix '*_List*' were not absolutely required, but were introduced to increase comprehensibility.

[1] This includes a zero with leading zeros.
[2] This includes a zero with leading zeros.

- Commands may contain a line break between any two tokens, but then require an extra backslash as the last token of all but the last line in order to allow continuation of the command.

Script ::= <InterruptHandler_List> <Command_List> <InterruptHandler_List>
Command_List ::= <Command>+
Command ::= <ModuleCommand> | (see 3.6) *<ControlCommand>* | (see 3.7) *<ControlFlowCommand>*
ControlFlowCommand ::= <LoopCommand> | (see 3.8.1) *<IfCommand>* | (see 0) *<SwitchCommand>* | (see 3.8.3) *<CriticalCommand>* (see 3.8.4)

Table 3.2: The structure of an IFS script

3.6 Module Commands

Table 3.3 shows the syntax of commands addressed to registered modules. There are three kinds of module commands: user-defined commands, predefined commands, and legacy commands. User-defined commands are the commands that are defined and registered in SystemC-based IFS modules. Predefined commands are commands that every module understands without defining them explicitly in the module. Legacy commands are more of a different way of handling a command in a script[3] rather than a new style of commands.

ModuleCommand ::= <ModuleName> (<UserCommand> <Parameter> | <PredefinedCommand> | <LegacyCommand>)* **Note**: These commands refer to a specific module.
*LegacyCommand ::= <LegacyCommandName> <Parameter>** **Note**: Legacy commands are commands of existing IFS modules in VHDL and need not be registered.

[3] These commands are intended to allow the reuse of existing (legacy) IFS modules written in VHDL. See also chapter 1.

UserCommand **Note**: A <*UserCommand*> shall be the name of a command that has been registered by the module. In general there are no restrictions in the choice of the name of the command. But if the command shall be named like the method in the IFS module it is of course subject to the C++ rules for valid identifiers.
ModuleName **Note**: This is an identifier of a module. It has to be unique in its context to allow unambiguous addressing. It is equal to the name of the instance of the referenced interface module. Valid identifiers in SystemC may contain all characters except '.' (dot), white-space[4] or '*' (asterisk).
Parameter ::= <*UserDefinedParameter*> | <*VariableReference*>
UserDefinedParameter **Note**: An arbitrary sequence of characters (a string), which is further processed within the command it belongs to.
VariableReference ::= #<*Identifier*> **Note**: The *Identifier* has to be defined by a previous command to be valid. An invalid reference is replaced with an empty string and a warning is issued.
PredefinedCommand ::= <*WaitClause*> | <*PrintClause*> | <*GetValue*> | (see **Fehler! Verweisquelle konnte nicht gefunden werden.**) <*Title*> | (see 3.6.1) <*IRQM*> | (see 3.6.2) <*IFSM*> | (see 3.6.3) <*AssignCommand*>

Table 3.3: The syntax of module commands

[4] According to the man-page of the function isspace() of the C standard library.

3.6.1 TITLE

Each module contains five predefined signals which can be used to display string information in a signal trace. With the command `TITLE` it is possible to set these signals. The <*index*> indicates which signal the string is assigned to. The length of the string itself is limited to 100 characters. The names of the signals are "title< *index* >". Legacy modules provide their own title implementation, which might differ from this one.

Title ::= `TITLE` < *index* > <*String*>*   **Note**: The *String*s may not contain a " (quotation mark). If quotation marks shall be used they have to be prefixed with the escape character. Multiple parameters are concatenated with an additional space between each element. Zero string parameters clear the title signal.   **Note**: The *index* must be a number between 0 and 4.

Table 3.5: The syntax of the TITLE command

3.6.2 IRQ for Modules

By means of this command it is possible to disable specific interrupts for a module. The syntax of this command is shown in Table 3.6.

<*IRQM*> *::=* `IRQ MASK` <*MaskSpecifier* >   **Note**: The <*MaskSpecifier*> has got the same semantics as described in Section 3.9. By default, all IRQs are enabled.

Table 3.6: The syntax of the IRQ command for modules

```
-- After this command the IRQs one and four are disabled for the module
ifsmodule1 IRQ MASK 0110
```

Figure 3.2: Example for the usage of the IRQ command for modules

3.6.3 IFS_M

The command makes it possible to enable or to disable all interrupts for the module. Table 3.7 shows the syntax of this command.

<IFSM> ::= IRQ IFS_M (1 | 0)

Note: '1' means that all interrupts are enabled and '0' means that all interrupts are disabled.

Table 3.7: The syntax of the IFS_M command

3.7 Controller Commands

Table 3.8 shows the syntax of commands concerning the controller. These commands are not directed toward specific modules, but configure the IFS system or initiate more global activities such as printing output or causing a synchronisation of modules.

ControlCommand ::= (IFS | ALL) *<ALLCommand>*

Note: The keywords IFS and ALL are synonyms. The keyword IFS was introduced, because the keyword ALL suggests that all modules will be affected by this command, which is not necessarily the case.

Note: It is possible to use the name IFS as module name. In that case the *<PrintClause>* and the *<WaitClause>* are regarded as module commands. The rest of the control commands are interpreted as control commands. Commands, which are not control commands, are always regarded as module commands.

ALLCommand ::=	SYNC (*<ModuleName>** | ALL)	| (see 3.7.20)
	CHECK *<ModuleCommand> <ReferenceValue>*	| (see 3.7.3)
	IRQ *<InterruptCommand>*	|
	<PrintClause>	| (see 3.7.4)
	<WaitClause>	| (see 3.7.1)
	QUIT [MODULES]	|
	REPORTLEVEL (INFO | DEBUG | ERROR | WARNING)[NO_IFS]	
	| EXPRESSION_ACCURACY	
	[<WordLength> [<IntegerWordLength>]]	(3.11)

Note: QUIT will cause the simulation to end as soon as a module requests this command. The optional parameter MODULES will shutdown the IFS modules only, allowing the DUT to continue.

Note: ALL WAIT is deprecated. Please use <ModuleName> WAIT followed by ALL SYNC ALL instead. ALL WAIT cycles may cause unexpected behaviour for the case of multiple clock domains.

Note: REPORTLEVEL will control the verbosity of the output messages. The option NO_IFS (with an underscore between NO and IFS) will suppress IFS internal messages coming from the controller.

ReferenceValue ::= <Parameter>			
InterruptCommand ::=	*<IrqEnableClause>*	|	(see 3.9.20)
	<IrqPriorityClause>		(see 3.9.1)

Table 3.8: The syntax of controller commands

3.7.1 WAIT

Table 3.9 shows the syntax of a wait command. Such a command suspends the execution of further commands for the addressed module. The two variants `WAIT simtime` and `WAIT deltime` can be interrupted by an interrupt. Upon completion of the execution of the interrupt, the interrupted wait command will be continued. A warning will be issued if the targeted simulation time has been exceeded in the meantime.

The `wait` command can also be used as a control command (in the form '`ALL` <*WaitClause*>'). This variant is deprecated (and a warning will be issued), but for compatibility reasons it will still cause every module to wait upon reaching this command. Hence, the keyword `ALL` in this case can be regarded as wildcard matching every module.

WaitClause ::= `WAIT` *(*	`cycles` <*PositiveInteger*>	|		
	`simtime` <*Time*>	|		(see 3.3)
	`deltime` <*Time*>		*)*	(see 3.3)

Note: In order to use `WAIT cycles`, an clock event has to be registered. This variant will wait the given number of clock cycles. When the `WAIT cycles` command is executed simultaneously with the occurrence of the event it is waiting for, it depends on the delta cycle in which the command is executed and on the delta cycles in which the event was notified, whether or not the event will be counted. For example, a `WAIT cycles` following a `WAIT simtime` or `WAIT deltime` will count the first event, even if it occurs exactly at the resulting wait time. This is because the preceding `WAIT` command will cause the module to fetch its instruction in the first delta cycle of the given simulation time, whereas the signal update (for example for the clock signal) will happen not until the next delta cycle. On the other hand, a `WAIT cycles` following a `WAIT cycles` will not count the event that occurs at the same time as the wait is executed, because this event was already consumed by the previous `WAIT` command (and the command fetch of the second `WAIT` happens in a delta cycle after the occurrence of the event).

Note: `WAIT simtime` waits until the specified simulation time is reached. A warning is issued if this time has already exceeded.

Note: `WAIT deltime` waits for the given duration.

Table 3.9: The syntax of a wait statement

3.7.2 SYNC

A SYNC command will suspend each module that reaches this command and that is an element of the given list. When all specified modules have reached the SYNC, execution is continued. This ensures that all modules will execute their next commands simultaneously.

A SYNC command can be interrupted by an interrupt. The SYNC will be re-executed upon completion of the interrupt handler.

3.7.3 CHECK

A CHECK-command will call the given module command and compare the value returned from this command with the specified reference. In order to be able to use a checked command call, it is necessary to use the macro `IFS_SET_CHECKVALUE(value)` within the implementation of the command. An example is shown in Figure 3.3.

```
-- call the Read command (with one parameter: 0x100) of module1 and
-- check if the result is 42
IFS CHECK module1 Read 0x100 42
```

Figure 3.3: Example for the usage of a checked command call.

An error is issued when no value to check has been set in the called command and a warning, if the values do not match. Otherwise, a message stating the equality will be issued on the ‘info’-level.

3.7.4 Print

The syntax of the print command is given in Table 3.10. This command can be used for user-defined output.

PrintClause ::= `print` <*Parameter*>*
Note: It is advisable to put the parameters in quotation marks. Otherwise the spaces will be lost.

Table 3.10: The syntax of the print command

3.8 Control Flow Commands

3.8.1 #loop

Figure 3.4 shows an example for the usage of the `#loop` command. In this example, the outer loop defines a loop variable called `outerLoopVariable`. The outer loop will make 3 iterations with `#outerLoopVariable` running from 0 to 2. The inner loop will make 5 iterations per iteration of the outer loop. The expression `#innerVar` will be successively replaced by the values 3, 2, 1, 0, -1 and `#someSymbol` will be replaced by `"Hello World"`, `1234`, `asdf`, `“Hello World”` and `1234`.

```
#loop outerLoopVariable 3
  module1 print "outerLoopVariable = " #outerLoopVariable
  #loop innerVar (3, -1) someSymbol var { "Hello World" 1234 asdf }
    module1 print "innerVar = " #innerVar ", someSymbol = " #someSymbol
  #eol
#eol
```

Figure 3.4: Example for the usage of the `#loop` command

Table 3.11 shows the syntax of the `#loop` statement. Note, that

- Loops can be nested to any depth.
- Loop variable names in nested loops must be unique. Otherwise the inner variable names will hide the outer variable names.
- There is no implicit `SYNC` in a loop, which means that different modules may be in different iterations of the same loop. A warning will be issued if there is a loop that contains commands for more than one module, but does not contain `SYNC`.

LoopCommand ::= `#loop` *<IterationDefinition> [<EnumerationDefinition>] <Newline>*
<Command_List>
`#eol`

Note: Variables defined in a loop command can be referenced within the enclosed *<Command_List>* to obtain their actual value.

Note: The identifiers of the iteration variable and the enumeration variable must be different.

IterationDefinition ::= [<Identifier>] (<NumberOfIterations> | (<IntervalStart> `,` *<IntervalEnd>* `)` *)*

Note: If the *<Identifier>* for the variable is omitted, the name `i` is assumed. In that case referencing occurs via `#i`.

If a *<NumberOfIterations>* is provided, the value of the variable iterates from zero to the *<NumberOfIterations>* minus one, else from *<IntervalStart>* to *<IntervalEnd>*.

EnumerationDefinition ::= [<Identifier>] `var` `{` *<EnumerationValue>** `}`

Note: If the *Identifier* for the variable is omitted, the name `var` is assumed. In that case referencing occurs via `#var`.

Note: The *Identifier* must not be named `var`.

Note: An empty sequence of *<EnumerationValue>* will issue a runtime warning.

The value of the variable will iterate (from left to right) through the values provided in the enumeration *<EnumerationValue>+* and starts again with the first value after reaching the last value.

Table 3.11: The syntax of the `#loop` command

3.8.2 #if

Figure 3.5 shows an example for the usage of an `#if` command. The `#if` command works as in other programming languages such as C/C++. Depending on the condition either the 'then'-part or the optional ‘elseif’-part/'else'-part is executed. A condition is interpreted as false if it is equal to the string literal '`0`' (zero) and true otherwise.

```
#if 1
  module1 print "This is always true"
#endif

#if 0
  module1 print "This is wrong."
#else
  module1 print "This is OK."
#endif

module1 ASSIGN cond = 3

#if $(#cond)
  module1 print "This is OK."
#endif
#if $(#cond-5)
  module1 print "This is OK, too."
#endif

#if $(#cond-3)
  module1 print "This is wrong."
#elseif $(#cond*0)
  module1 print "This is also wrong."
#else
  module1 print "This is OK."
#endif
```

Figure 3.5: Example for the usage of the `#if` command

Table 3.12 shows the syntax of the `#if` command. Note, that it is legal to use expressions (3.11) in the condition, to have empty 'then'- ‘elseif-’, and 'else'-part, and to nest `#if` commands.

IfCommand ::= `#if` *<Condition> <Newline>* *[<ThenPart>]* *[<ElseifPart>]* *[<ElsePart>]* `#endif` **Note**: *<Condition>* can be any token. A `0` is interpreted as false, everything else as true.
ThenPart ::= <Command_List>
ElseifPart ::= `#elseif` *<Condition> <Newline>* *<Command_List>*
ElsePart ::= `#else` *<Newline>* *<Command_List>*

Table 3.12: The syntax of the `#if` command

3.8.3 #switch

Figure 3.6 shows an example for the usage of a `#switch` command. The `#switch` command works as in other programming languages such as C/C++. Depending on the expression following the keyword `#switch`, the first matching `#case` block will be executed. A `#case` block matches if the string following the `#case` statement equals the string following the `#switch` statement. The `#cases` are checked in their order of appearance in the script, i.e., top to bottom. A `#case` statement without an additional string always matches and therefore can be used as a 'default' case.

```
-- a loop with 6 iterations:
-- #i=0 and #var = symbol1
-- #i=1 and #var = 8
-- #i=2 and #var = "X Y"
-- #i=3 and #var = qwer
-- #i=4 and #var = symbol1
-- #i=5 and #var = 8
#loop 6 var { symbol1 $(2*4) "X Y" qwer }

  #switch #var

    #case symbol1
      module1 print "1. iteration " #i
    #case 8
      module1 print "2. iteration " #i
    #case "X Y"
      module1 print "3. iteration " #i
    #case

      #if $(#i-3)

        #switch #i

          #case 4
            module1 print "5. iteration " #i #var
          #case 5
            module1 print "6. iteration " #i #var
        #endswitch

      #else
        module1 print "4. iteration " #i #var
      #endif

  #endswitch
#eol
```

Figure 3.6: Example for the usage of the `#switch` command

Table 3.13 shows the syntax of the `#switch` command. Note, that it is legal to use expressions (3.11) as switch value, to have empty case blocks, and to nest `#switch` commands.

SwitchCommand ::= `#switch` <*SwitchValue*> <*Newline*>
<*CaseBlock*>*
`#endswitch`

Note: <*SwitchValue*> can be any token.

Note: Case blocks do not 'fall through' as in C/C++. Instead, the command following the `#endswitch` is executed when the end of a case block is reached.

CaseBlock ::= `#case` *[<CaseLabel>] <Newline>*
[<CommandList>]

Note: <*CaseLabel*> can be any token. The <*CaseLabel*> is compared with the <*SwitchValue*> of the surrounding <*SwitchCommand*>. If they match, the <*CaseBlock*> is executed. A missing <*CaseLabel*> always matches. The use of variables and expressions in <*CaseLabel*> is also possible.

Table 3.13: The syntax of the `#switch` command

3.8.4 #critical

Sometimes it is desirable that commands are not evaluated by any module. For this purpose the control flow command #critical is introduced. By means of this command it is possible to exclude modules from code parts in the script. Or in other words, the modules which are excluded never reach the commands in the critical section. Figure 3.7 shows an example.

```
--Only module2 can reach the print command
#critical EXCLUDED module1
      ALL print "condition"
#endcritical

module1 print module1
module2 print module2
IFS SYNC ALL

- Only module1 can reach the print command
#critical INCLUDED module1
      ALL print "condition"
#endcritical

module1 print module1
module2 print module2
IFS SYNC ALL
IFS QUIT
```

Figure 3.7: Example of the command #critical

The Table 3.14 shows the syntax of the command critical.

CriticalCommand ::= `#critical` *(*`INCLUDED` | `EXCLUDED`*)* *<ModuleName>+ <Newline>* *[<CommandList>]* `#endcritical` *<Newline>* **Note:** `INCLUDED` means that all modules (listed behind the `INCLUDED`) can reach the *CommandList* in the critical section. `EXCLUDED` means that all modules (listed behind the `EXCLUDED`) can not reach the *CommandList* in the critical section.

Table 3.14: The syntax of the #critical command

3.9 Interrupts

An occurred interrupt will trigger the execution of the associated interrupt-handler in the currently executed script. The syntax of interrupt handlers is shown in Table 3.15 and an example can be found in Figure 1.2.

Further interrupts cannot interrupt the execution of an interrupt handler. Hence, nesting interrupts is not possible. Further interrupts are stored and handled when the active interrupt-handler is left. The scheme for selecting the next interrupt is described in Chapter 2.2.

User-defined commands cannot be interrupted, only `SYNC` and `WAIT` can. When a `SYNC` command is interrupted, it will be finished after the execution of the interrupt handler. When an interrupted `WAIT` is continued, it will (a) wait until the simulation time it would have waited for without an interrupt (if this point in time is not yet reached) or (b) continue immediately, if the time is already exceeded.

Note: There is no implicit `SYNC` in an interrupt handler. So, in general, the execution of the interrupt handler will start at different simulation times for different modules, unless there is an explicit `SYNC` at the beginning.

*InterruptHandler_List ::= (<InterruptHandler> <Newline>)**
InterruptHandler ::= `#isr` *<InterruptID> <Newline> <InterruptHandlerCommand_List>* `#end`
InterruptID ::= <NonNegativeInteger>
*InterruptHandlerCommand_List ::= (<InterruptHandlerCommand> <Newline>)**

InterruptHandlerCommand ::= <*Command*> | `#break` (see Chapter 2.2)

Table 3.15: The syntax of interrupt handlers

3.9.1 Priorities

The syntax for setting interrupt priorities is shown Table 3.16; an example is shown in Figure 3.8. In this example, the priority of interrupt 1 is set to 3 and is therefore higher than the priority for interrupt 2, which is -5.

```
IFS IRQ PRIO 1 3
IFS IRQ PRIO 2 -5
```

Figure 3.8: Setting priorities of interrupts

Multiple pending interrupts are handled sequentially according to their priority. If two pending interrupts have the same priority, the interrupts are handled in their order of occurrence. If two pending interrupts have the same priority and occur simultaneously, the execution order of the corresponding interrupt handlers is undefined. If no priority was assigned to an interrupt, the priority defaults to 0.

IrqPriorityClause ::= `PRIO` <*InterruptID*> <*InterruptPriority*>
InterruptPriority ::= <*Integer*> **Note**: A lower number indicates a lower priority. The default value of a priority is 0.

Table 3.16: The syntax of priorities for interrupts.

3.9.2 Masking

The syntax for interrupt masking, i.e. enabling and disabling interrupts, is shown in Table 3.17; an example is shown in Figure 1.2.

When an interrupt is disabled, interrupt requests will be ignored completely, i.e., they won't be stored until the interrupt is enabled.

Attention: Masking an already pending interrupt will not remove it from the list of pending interrupts. As a result it is possible that even after masking an interrupt its handler gets executed.

IrqEnableClause ::= `MASK` <*IrqMaskSpecifier*> **Note**: By default, all interrupts are enabled.
IrqMaskSpecifier ::= `b#(` `0`|`1` `)`+ | `d#(`0-9`)`+ | `h#(`0-9 | a-f | A-F`)`+ | `(` `.`|`0`|`1` `)`+ | `#`<*InterruptID*> `(` `0`|`1` `)` **Note**: The first alternative uses a string of bits (1s and 0s), where each bit indicates whether the interrupt with the ID identical to the index[5] of this bit is enabled (1) or disabled (0). If the number of letters in the specification-string is smaller than the number of available interrupts, the masking of interrupts who have an ID equal to or higher than this number remains unchanged. The second and third variant is similar to the first one, but instead of a binary representation, the number is specified in decimal or hexadecimal notation. The fourth variant works identical to the first one, but also allows a dot indicating interrupts whose masking remains unchanged. The fifth approach explicitly names the ID of an interrupt and specifies whether it is enabled (1) or disabled (0). **Note**: The bit vectors used must be a single token, i.e., they must not contain spaces.

Table 3.17: The syntax for masking interrupts

3.9.3 #break

The `#break` command must only occur within an interrupt handler and will pass control back to the point executed prior to the interrupt, just like `#end`. The next occurrence of this interrupt, however, causes the execution to resume with the statement following the `#break` statement, which caused leaving the handler last time.

3.10 Randomisation

A <*RandomExpression*> can occur at each location within the script in place of a normal token. This expression will be evaluated once each time a line is processed. The result of the evaluation will replace the <*RandomExpression*>. Note that random expressions can be nested. An example is shown in Figure 3.9; the syntax of random expressions is shown in Table 3.18.

[5] The index is counted from right to left and begins with one.

The randomisation is based on the randomisation features of the SystemC Verification Library (SCV). Hence, the SCV seed functions can be used to control the randomisation in order to create reproducible tests. It should be noted that the random values generated can differ between different platforms. Hence, test sequences on a 32bit machine can (and often do) differ from those obtained on a 64bit machine.

```
My_module1 ${ Command1, Command2 }
My_module1 Command1 ${ ${ X 10%, Y 9%, Z } 90%, Q }
My_module1 ${ Command1, Command2 } ${ X 90%, Y } 42
My_module1 Command2 ${ -8 :-3 75%, "Hello World" } 23
```

Figure 3.9: Exemplary usage of randomisation

RandomExpression ::= $[!]{ *<RandomParameter_List>* }

Note: The meaning of the optional symbol ! (exclamation mark) is described in Section 3.10.1.

Note: A *RandomExpression* is evaluated by the controller and replaced by the result.

Note: If no parameter is provided, a warning is issued and the expression will be replaced by an empty string.

*RandomParameter_List ::= <RandomParameter> (, <RandomParameter>)**

RandomParameter ::= <RandomValue> [<Probability>]

Note: If the *Probability* is omitted, the remaining probability is equally distributed onto the parameters with omitted probabilities. Since probabilities are handled as integers, any discretisation will add the value in decimal places to the probability of the first parameter with omitted probability.

Example:
```
${ red 10% , blue , lightblack }
```
equals
```
${ red 10% , blue 45% , lightblack 45% }
```

Example:
```
${ red 27% , blue , darkwhite, mauve }
```
equals
```
${ red 27% , blue 25% , darkwhite 24% , mauve 24% }
```

Note: If all probabilities are specified, but their sum is not equal to 100%, an error is issued.

RandomValue ::= <Token> | <Range> **Note**: For an explanation of *Token* see the Section 2.2.
Range ::= <Integer> `:` *<Integer>* Example: `-20:14`
Probability ::= `0-100 %` Example: `42%`

Table3.18: The syntax of random expressions

3.10.1 Randomisation without Repetition

Sometimes it is required that the values of a randomisation must not be repeated. For this case the <*RandomExpression*> can be extended by the symbol '`!`' (exclamation mark). If this symbol occurs, no <*RandomParameter*> will be repeated until all values have occurred once. **Note:** Ranges and nested randomizations are regarded as one parameter.

When all parameters have occurred once, the randomisation restarts. A <*RandomExpression*> with a probability of zero will never be selected.

```
-- We have the following randomization
#loop 6
  module print $!{A 10% , ${AA ,BB}, C 10%}
#eol

--The output could be

-- One of the values is selected
[module]: AA

-- AA and BB can't occur now, since the parameter ${AA,BB} has been used once
[module]: A

-- A, AA and BB can't occur
[module]: C
```

```
-- All three parameters have occurred. Now all parameters are available
[module]: BB

-- The second parameter has occurred, hence only C and A are possible
[module]: C

-- Only A is possible
[module]: A
```

Figure 3.10: Exemplary usage of randomization without repetition

It is important to note that there is a difference between module commands and common commands. When a module command is executed, the selected <*RandomParameter*> (for the randomizations without repetition) is only stored for that module. In other words, only the requesting module has an effect of the randomization without repetition.

But, when a control-flow command or a control command is executed, the selected parameters are stored for all modules. Hence, all modules have an effect of the randomization without repetition.

Furthermore, one exception is to mention. If a particular module command (see Figure 3.11) is requested, the parameters of all randomizations in this line are stored for all modules.

```
$!{module1, module2, module3} print “TEST”
```

Figure 3.11: Randomization without repetition

3.11 Expressions

An <*Expression*> can occur at each location within the script in place of a normal token. This expression will be evaluated once each time a line is processed (after the random expression evaluation and after the variable substitution). The result of the evaluation will replace the <*Expression*>. Note, that expressions cannot be nested. An example is shown in Figure 3.12; the syntax of random expressions is shown in Table 3.19.

Expressions can be arithmetic expressions, bit operations, or logic (Boolean) expressions. The accuracy of the expression evaluation can be determined by the command `EXPRESSION_ACCURACY`.

For Boolean expressions the number `0` means false and anything else true. If the result of such an expression is true, it will be substituted by `1` and by `0` otherwise.

```
module1 print $( 1 )
module1 print $( h#ff )
module1 print $( 1+1 )
module1 print $(1)
module1 print $(3*2)
module1 print "The value is : " $(3*2) A B C
module1 print "The value is : " $(3*3) " with text behind the expression."
module1 print "The value is : " $(3*2) " and a second value is: " $(42)
module1 print $(1+2*3) $((1+2)*2) $((((+1)*2+3)*(-11--21)))
module1 print The previous line should be: 7 6 50
module1 print $( b#00100 )
module1 print $(b#00100)
module1 print $(h#10/b#010) " (16/2 = 8)"
module1 print $(h#ff) $(h#0ff) "(255 255)"
module1 print $(h#100/+2+-b#10) "(256/2+(-2)=126)"
module1 print $(h#f) $(h#ff) $(h#fff) $(b#1) $(b#11) $(b#111) "(15 255 4095
1 3 7)"
IFS EXPRESSION_ACCURACY 8
module1 print $(-1) $(--1) $(---1) $(----1) "(255 1 255 1)"
module1 print $(-1) $(1/2) $(257) "(255 0 1)"
module1 print $(60/3/5) "(4)"
IFS EXPRESSION_ACCURACY 8 4
module1 print $(1/2) $(15+15/8/2) "(0.5 15.9375)"
IFS EXPRESSION_ACCURACY
module1 print $(-1) $(1/2) "(18446744073709551615 0)"

# relational operators
module1 print $( 5<10 ) (1)
module1 print $( 10<5 ) (0)
module1 print $( 1<0+2 ) (1)
module1 print $( 3+4*5 < 22*1 ) (0)
module1 print $( 3+4*5 == 22+1 ) (1)
module1 print $( 3+4*5 != 20+22+1 ) (1)
module1 print $( (1+4>3)*5 ) (5)
module1 print $( (1+4>3)*5-(2>1) ) (4)
module1 print $( (1+4>3)*5-2>1 ) (1)

# Boolean negation
module1 print $( !1 ) (0)
module1 print $( !100 ) (0)
module1 print $( !0 ) (1)
module1 print $( !(1-1) ) (1)
module1 print $( !(1*1) ) (0)

# bit-wise negation
module1 print $( ~0 ) ($(-1))
module1 print $( ~1 ) ($(-2))

# Boolean operators
module1 print $( 1 && 1 ) (1)
module1 print $( 0 && 1 ) (0)

module1 print $( 1 || 0 ) (1)
module1 print $( 0 || 0 ) (0)

module1 print $( 1+3 || 42 ) (1)
module1 print $( 3-(2+1) || 256*0 ) (0)
```

```
# bit operations
module1 print $(1<<2) (4)
module1 print $(4>>2) (1)

module1 print $(2|1)  (3)
module1 print $(2&1)  (0)

# conversion
IFS EXPRESSION_ACCURACY 8

module1 print $(1<<2)'b# (b#00000100)
module1 print $(1<<2)'h# (h#04)
```

Figure 3.12 : Exemplary usage of expressions

Expression ::= `$(` *<ExpressionContent>* `)[`'h#|'b#*]*

Note: An *Expression* is evaluated by the controller and replaced by the result.

Note: After the evaluation of the expression, it is possible to convert the result in a binary or hex representation. This can be accomplished by using the optional terminals `'h#` and `'b#`.

ExpressionContent ::= <LogicalAndExpression> (`||` *<LogicalAndExpression>)** (or)

LogicalAndExpression ::= <RelationalExpression> (`&&` *<RelationalExpression>)** (and)

RelationalExpression ::= *<AdditiveExpression> (*

`>=`	*<AdditiveExpression>*	| (greater equal)
`<=`	*<AdditiveExpression>*	| (lower equal)
`<`	*<AdditiveExpression>*	| (lower)
`>`	*<AdditiveExpression>*	| (greater)
`==`	*<AdditiveExpression>*	| (equal)
`!=`	*<AdditiveExpression>*	(unequal)

*)**

AdditiveExpression ::= MultiplicativeExpression (`+` *MultiplicativeExpression* |
`&` *MultiplicativeExpression* |
`|` *MultiplicativeExpression* |
`<<` *MultiplicativeExpression* |
`>>` *MultiplicativeExpression* |
`-` *MultiplicativeExpression)**

MultiplicativeExpression ::= Factor (`*` *Factor* | `/` *Factor)**

Factor ::= *IfsNumber* |
`(` *Expression* `)` |
`-` *Factor* | (unary -)

+ *Factor* | (unary +) ! *Factor* | (Boolean negation) ~ *Factor* | (bit-wise negation) *IfsNumber ::= (+ | -)(* b#(0|1)+ | h#(0-9 | a-f | A-F)+ | *[d#](0-9)+* *)* *IfsFloatingPointNumber::=[(+ | -)](* b#(0|1)+ *[. (0|1)+]* | h#(0-9 | a-f | A-F)+ *[.* (0-9 | a-f | A-F)+ *]* | *[d#](0-9)+ [. (0-9) +]* *)*
Note: The intermediate rules are to ensure the correct precedence of the operators, that is from highest to lowest priority: unary - + ! ~, * /, - +,<<,>>,&,|, >= <= < > == !=, &&, ||. **Note**: The underlying data type of the expressions is sc_ufix. The accuracy of (intermediate) expressions during the evaluation can be controlled by the controller command IFS EXPRESSION_ACCURACY *[<WordLength> [<IntegerWordLength>]]*. The two parameters *<WordLength>* and *<IntegerWordLength>*, which must be *<NonNegativeInteger>*, are passed as arguments to the underlying sc_ufix. If the *<IntegerWordLength>* is omitted, it will be set to *<WordLength>*, i.e., the number of digits behind the binary point is zero in this case. If both numbers are omitted, the accuracy is set to the default accuracy (64, 64). **Note:** In Boolean expressions a 0 is interpreted as false and anything else as true. **Note:** The *IfsFloatingPointNumber* is not used in the expressions. This string representation is used in the IFS Utilities (see 2.3) to convert a string into a double.

Table 3.19: The syntax of expressions

3.12 Variables

The controller holds a set of global variables. These variables can be referenced anywhere in the script (e.g. in expressions, as parameters to commands, as loop index boundaries, etc.) by preceding the variable name with a '#'. In this case the token with the variable reference is replaced by the variable's value. In order to create such a variable (or to set it to different value) the predefined module command ASSIGN is used. An example is shown in Figure 3.13 the syntax of random expressions is shown in Table 3.19.

Note that ASSIGN is a predefined module command and not a controller command. The reason for this is the fact that controller commands are executed by every module. Hence, an assignment would be executed multiple times if it was realised as controller command. Especially for assignments like x=x+1 this would result in unexpected results. The advantage of making ASSIGN a module command is that assignments are executed only once.

```
module1 ASSIGN testVar = 42
module1 ASSIGN testVar2 = $(1+1)
module1 ASSIGN testVar3 = "Hello, world"

module1 print #testVar " (42)"
module1 print #testVar2 " (2)"
module1 print #testVar3 "  (Hello, world)"

-- variable re-assignment
module1 ASSIGN testVar2 = $(2*4)
ALL SYNC ALL
module2 print #testVar2 " (8)"

-- shadowing of global variables by loop variables
#loop testVar2 2
  module1 print #testVar " (42)"
  module1 print #testVar2
  module1 print #testVar3 "  (Hello, world)"
#eol

module1 ASSIGN someVar = h#ff
ALL SYNC ALL
module2 print $(#someVar +1) "(256)"
module2 print $(2+#someVar) "(257)"

-- Use an expression as loop range
#loop ($(#testVar2+5), $(#testVar2+4))
  module2 print #i
#eol
```

Figure 3.13: Exemplary usage of variables

AssignCommand ::= `ASSIGN` <*VariableName*> `=` <*Value*>

Note: The `ASSIGN` command assigns <*Value*> to a variable called <*VariableName*>. Value can be any kind of token. <*VariableName*> can be any name (without the leading '#'), that is unambiguous when preceded with the '#'. For example '`loop`' is not a valid variable name, because a reference to this variable ('`#loop`') would be a reserved keyword. Variables are created on demand, i.e., they need not be declared.

Note: If unknown variables or an empty sequence of <*EnumerationValue*> are referenced, an error will be issued.

Note: The *AssignCommand* can also be used to initialise arrays. The syntax of arrays is described in the Section 3.13.

Note: It is forbidden to use the characters '[' and ']' in the <*VariableName*>.

Table 3.20: The syntax of variables

3.13 Arrays

In many cases it is desirable to use arrays. For this reason the concept of the variables is extended by arrays. Figure 3.14 shows examples of their usage. Similar to variables, the controller holds a list of global arrays. The elements of the arrays can be referenced anywhere in the script. The access to an array is introduced by a '#', followed by the variable name and '[<*NonNegativeInteger*>]'. Between the brackets no expressions or randomisations are allowed, only variables are possible. If you need expressions or randomisations you have to evaluate them in an extra command line and store the result in a variable. The variable can then be used in the array expression. The first element of an array has the index 0 (zero). If the access is not within the bounds of the array, an error will be issued.

The procedure of the replacement is similar to that of variable replacement. An array is initialised by the assign command. The syntax of the assign command for arrays is described in Table 3.21.

```
-- The first variation to initialise an array
module1 ASSIGN test = a b c

-- Access to the array content
module1 print "Array test[0]: " #test[0]
module1 print "Array test[1]: " #test[1]
module1 print "Array test[2]: " #test[2]

-- Only a variable is possible between the brackets.
#loop i 3
      module1 print "loop: " #test[#i]
#eol
```

```
-- Assignment of values to the array
module1 ASSIGN test[0] = aa
module1 ASSIGN test[1] = bb
module1 ASSIGN test[2] = cc

#loop i 3
      module1 print "loop: " #test[#i]
#eol

-- The second variation to initialise an array
module1 ASSIGN [10]myArray = "All The Same!"
#loop 10
      module1 print "Content: " #myArray[#i]
#eol
```

Figure 3.14: Exemplary usage of arrays

AssignCommand ::= ASSIGN *((<VariableName> = <Value><Value>+) |*
([<NonNegativeInteger>]<VariableName> = <Value>) |
(<VariableName>[<NonNegativeInteger>] = <Value>))

Note: There are two possibilities to initialise an array. The first one defines a variable name with an initialisation list. The values behind the '=' are the initialisation list. The second one is to initialise all elements of an array to the same value. The size of the array is specified in front of the variable name (between the brackets). All elements of the array are initialised to the value standing on the right hand side of the '='.

Note: It is also possible to change the value of a specific element of an array. Behind the variable name the index (a *<NonNegativeInteger>*) is specified between brackets. On the right hand side of the '=' is the new value. If the index is not valid, an error will be issued.

Table 3.21: The syntax of arrays

Quick Reference

Commands

```
-- <ModuleName> <Command> <Parameter>*
clk SetPeriod 100 ns

-- pre-defined module commands:
-- ASSIGN, print, WAIT
clk ASSIGN x = 42
clk print "Hello, World!"
clk WAIT simtime 200 ns
```

Variables

```
m ASSIGN testVar = 42
m ASSIGN testVar2 = $(1+1)
m ASSIGN testVar3 = Symbol
m ASSIGN testVar3 = "Hello, world"
m ASSIGN testVar4 = a b c
m ASSIGN [10]testVar4 = a
m ASSIGN testVar4[5] = b
m print #testVar #testVar2 #testVar3
```

Expressions

```
-- (256/2+(-2)=126)
m print $(h#100/+2+-b#10)
-- relational/Boolean expression
m print $(2*5+#x < 100 && #y >= 3)
-- binary/hex conversion
m print $(1 +1)'b#
m print $(1 +1)'h#
```

Random Expressions

```
m print ${ "Hello, world!", Something }
m print ${ "A B C" 30%, ${ x, y }, Z 30% }
${m1, m2 95%, m3} print "It's me."
m print ${ -3:-6 20%, ${ -10:-7, X}, Z}
```

Control Flow

```
#if $(#i == 3)
  m print True

#elseif $(#i == 4)

 m print True
#else
  m print False
#endif

#switch #var
  #case symbol
    m print A
  #case 8
    m print B
  #case "X Y"
    m print C
  #case
    m print Default
#endswitch

#loop myVar 3
  m print myVar = #myVar
#eol

#loop myVar (4, -2)
  m print "myVar =" #myVar
#eol

#loop myVar (-3, 3) var { red green blue }
  m print myVar = #myVar , var = #var
#eol

#loop 3 X var { red green blue }
  m print i = #i , X = #X
#eol

#critical INCLUDED m
  m print True
#endcritical

#critical EXCLUDED m
  m print True
#endcritical
```

Controller Commands

```
-- IFS and ALL are synonyms
-- sync all modules
ALL SYNC ALL
-- sync modules m1 and m2
ALL SYNC m1 m2

-- compare IFS_SET_CHECKVALUE(..) with Ref
IFS CHECK <ModuleCommand> <ReferenceValue>
-- check if m1 cmd1 'returns' 100
IFS CHECK m1 cmd1 100

-- enable IRQ 3, disable 2 and 1
IFS IRQ MASK b#100
-- disable IRQ 3
IFS IRQ MASK #3 0

-- set PRIO of IRQ 2 to 5
IFS IRQ PRIO 2 5

IFS print Hello, World!

-- stop simulation/shut down modules
IFS QUIT [MODULES]

-- set amount of displayed messages
IFS REPORTLEVEL ( INFO | DEBUG | ERROR |
WARNING )

-- set evaluation accuracy
IFS EXPRESSION_ACCURACY [<WL> [<IWL>]]
```

IFS Modules

```
-- wait for 20*IFS_REGISTER_CLOCK_EVENT(X)
m WAIT cycles 20
-- wait until simulation time is 100 ns
m WAIT simtime 100 ns
-- wait for 10 ns
m WAIT deltime 10 ns

-- TITLE command
m TITLE 0 “My Title”
-- IRQ commands
m IRQ MASK 0110
m IRQ IFS_M 1
```