# Saber® MAST Language Reference Manual

Version Z-2007.03, March 2007



SYNOPSYS®

# Copyright Notice and Proprietary Information



## Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

## Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

## Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIM$^{plus}$, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

## Service Marks ($^{SM}$)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Saber® MAST Language Reference Manual, Z-2007.03

# Contents

**Contents**

# Preface

This manual defines the MAST language accurately and completely. Its primary audiences are the tool implementers and advanced users. It is not intended as a user guide on how to use MAST for modeling. Other resources such as books, tutorials, and classes are better resources for learning the language and how to model with it.

This document contains

- the formal syntax and semantics of all MAST constructs
- the definition of the elaboration and simulation semantics
- the predefined language environment
- the definition of the external interface

## Related Documents

1. A. Mantooth and M. Fiegenbaum, *Modeling with an Analogy Hardware Description Language*, Kluwer, Dordrecht, The Netherlands, 1995
2. P. Duran, *A Practical Guide to Analog Behavioral Modeling for IC System Design*, Kluwer, Dordrecht, The Netherlands, 1998
3. R. Cooper, *The Designer's Guide to Analog & Mixed-Signal Modeling*, Avant! Corporation, Fremont, California, 2001

## Conventions

This manual uses the following conventions:

**Syntactic Description**

The syntax of the MAST language is described using an extended Backus-Naur form with the following conventions.

1. Lowercase words in regular font, some containing embedded underline characters, are used to denote syntactic categories, for example:

   lowercase_character

When the name of a syntax category is used other than in a syntax rule the underline characters are replaced by space characters, i.e. in ordinary text the syntax category of the example is called “lowercase character”.

2. Lowercase words in bold font are used to denote keywords in the MAST language, for example:

   **template**

3. A production consists of a left-hand side, the symbol "::=", and a right-hand side. The left-hand side is always the name of a syntactic category; the right-hand side is a replacement text.

4. A production is a rule for textual replacement: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

5. Alternative items on the right-hand side of a production are either separated by a vertical bar:

   identifier ::= basic_identifier | extended_identifier

   or presented as a list of single-character alternatives introduced by the words “one of”:

   digit ::= one of 0 1 2 3 4 5 6 7 8 9

   In a production of this second form each alternative is interpreted literally. A vertical bar that is part of the syntax defined by a production is enclosed within quotation marks, i.e. "|".

6. Optional items on the right hand side of a production are enclosed in italicized square brackets. The following two productions are equivalent:

   unary_primary ::= *[* unary_operator *]* primary

   unary_primary ::= primary | unary_operator primary

   A left bracket or right bracket that is part of the syntax defined by a production is enclosed within quotation marks, i.e. "[" or "]".

7. Repeated items on the right-hand side of a production are enclosed in italicized braces. The items may appear zero or more times. Repetitions are left-recursive. The following two productions are equivalent:

   or_expression ::= and_expression *{* "|" and_expression *}*

   ```
   or_expression ::=
           and_expression
           | or_expression "|" and_expression
   ```

A left brace or right brace that is part of the syntax defined by a production is enclosed within quotation marks, i.e. "{" or "}".

8. A syntactic category whose name starts with an italicized part is equivalent to the category named by the part in regular font. The italicized part intends to convey some semantic information. For example, *val*_name and *parameter*_name are syntactically identical and equivalent to just name.

**Semantic Description**

Narrative text is used to describe the meaning of a particular construct and to introduce concepts. An italicized term indicates a new concept that is subsequently defined. Finally, a term in uppercase characters refers to an item in the predefined language environment.

A number of sections provide additional information; they are not part of the language definition. Examples serve to illustrate a construct. Conclusions and additional information specific to the definition are listed in a Notes section, and cross References to other sections of the manual are given where useful. Justifications for some design decisions are described in a Rationale section. Finally, the Commentary section gives explanations that put a definition in perspective with the rest of the language and describes hints, conventions etc. that may be useful for users of the MAST language to know.

In the description the following terms are used:

error

The condition described represents an ill-formed description. An implementation is required to detect the condition and report it to the user.

erroneous

The condition described represents an ill-formed description. However, an implementation is not required to detect and report the condition. Usually it is not possible to detect an erroneous condition during the processing of the language, for example a division by zero.

illegal

A synonym for error.

legal

The condition described represents a well-formed description.

undefined

The result is not specified by this language definition. Therefore, it may be different for different implementations.

# 1

# Design Entities

*The chapter describes* MAST *design entities and their template definitions.*

A *design entity* represents a portion of a hardware design that performs a well-defined function and has a well-defined interaction with the rest of the design. It may represent an entire system, a subsystem, a chip, an off-the-shelf part, or anything in between.

A design entity may be composed of interconnected components, each bound to a design entity that defines the behavior or structure of the component. Thus, each design entity is at the root of a hierarchy of components, and its immediate descendents are the components of which it is composed. The design entity that represents a complete hardware design is at the root of the *design hierarchy*.

In the MAST language a design entity is defined by a template definition. The *root template* is the design entity at the root of the design hierarchy.

## Template Definitions

A template definition defines the interface of, and the function performed by, a design entity.

template_definition ::=
    template_header "{" template_body "}" eos

root_template ::=
    template_body

The template definition acts as the template declaration.

## Template Header

The template header defines the interface of a design entity, that is, the way a design entity interacts with the rest of a design.

template_header ::=
template_header_definition *{* template_header_sentence *}*

template_header_definition ::=
*{* template_attribute *}* **template** identifier
*[* connection_list *] [* = *template*_argument_list *]* eos

template_attribute ::=
**encrypted**
| **element**
| **component**

The identifier following the keyword **template** is the name of the design entity.

A template definition whose template header definition decorates the template with the **element** attribute implicitly decorates each analog system variable whose declaration is a declarative item in the corresponding template body with the export attribute.

**Notes**

The **encrypted** and **component** template attributes have no meaning in the MAST language. See Special Attributes.

The name of the root template is defined by the implementation, not by the text of the model.

The root template cannot be decorated with any template attributes.

**References**

Argument Lists, Special Attributes

## Template Connections

Template connections allow a design entity to interact with the rest of the design.

connection_list ::=
connection_element *{ [* , *]* connection_element *}*

connection_element ::=
connection_definition *[* : connection_specification *]*

connection_definition ::=
identifier
| decimal_literal

connection_specification ::=
identifier
| decimal_literal

Each connection element defines a formal connection of the template. The class and type of the formal connection must be defined by a subsequent (possibly implicit) declaration; the class must be either a pin, or an analog system variable of kind **var** or **ref**, or a state. A connection element without a connection specification is equivalent to a connection element whose connection specification is the same as the connection definition.

The connection definition defines a simple name or a decimal name that is said to be the *external name* of the formal connection. Similarly, the connection specification defines a simple name or a decimal name that is called the *internal name* of the formal connection. It is an error if two formal connections have the same external name. It is also an error if the external name of a formal connection denotes the predefined pin 0.

If the internal name of a formal connection is different from the external name of any formal connection in the connection list, and if the internal name is different from the internal name of any formal connection whose connection element precedes the formal connection in the connection list, and if the internal name does not denote the predefined pin 0, then the connection element implicitly declares an object in the template body whose name is the internal name of the formal connection and whose class, subclass (if any), kind (if any), and type or unit is the same as that of the object denoted by the external name of the formal connection. The declaration is deemed to precede any template body sentence.

If the internal name of a formal connection is different from the external name of the formal connection, then the connection element also defines an implicit collapse specification for the two objects denoted by the internal name and the external name.

### References

Simple Names, Decimal Names

### Rationale

The external name of a formal connection cannot denote the predefined pin 0 to avoid the confusion caused by the common assumption that 0 is always the name of the reference pin.

## Template Header Declarations

Template header declarations declare the formal connections and formal arguments of a template, as well as objects decorated with the **external** or **export** attributes.

```
template_header_sentence ::=
    template_header_declarative_item
  | eos

template_header_declarative_item ::=
    type_declaration
  | unit_declaration
  | pin_type_declaration
  | parameter_declaration
  | state_declaration
  | analog_variable_declaration
  | pin_declaration
```

The identifier in the declaration of a formal connection must be the external name of the corresponding template connection element.

An alias is an alternate name for an entity.

The declaration of a formal argument of a template implicitly declares an alias for the base type of the formal argument. The name of the alias is a qualified name whose prefix is the name of the design entity and whose suffix is the simple name of the formal argument. The declaration of a formal argument of a template implicitly decorates the object with the **export** attribute.

### Examples

The following two template definitions are equivalent:

```
template one out:value = amplitude
state nu out              # declares external name and internal
                          # name of connection element
number amplitude          # declares argument amplitude of type
                          # number and the qualified name
                          # one..amplitude as an alias for
                          # number
{ ... }

template two out = amplitude
state nu out              # declares external name of
                          # connection element
number amplitude          # declares amplitude as before
{
     state nu value       # declares object of the same
                          # class and type as out
     control_section {    # collapse specification
          collapse(out, value)
               }
          ...
}
```

### References

Type Declarations, Unit Declarations, Pin Type Declarations, Objects, Argument Lists, Qualified Names

---

## Template Bodies

A template body defines the behavior or structure of a design entity.

template_body ::=
*{* template_body_sentence *}*

template_body_sentence ::= sentence

sentence ::=
declarative_item
| statement
| eos

declarative_item ::=
declaration_statement
| alter_specification
| control_section_specification

It is an error if a variable declaration is a declarative item in a template body. It is also an error if a return statement is a statement in a template body.

### References

Declarations, Function Definitions, Specifications, Statements

# 2

# Functions

*This chapter describes MAST functions.*

Functions define algorithms for the computation of one or more values. They may be used to improve the code organization of a design unit (see Design Units) and support code re-use. Some functions may also be used to resolve the value of a net, and for limiting the change of an independent variable from one iteration to the next during the determination of an analog solution point.

There are two kinds of functions: MAST functions that are written using statements of the MAST language, and *foreign functions* that are written in a language other than MAST, for example C or Fortran. A foreign function is a function that is decorated with the **foreign** attribute.

Functions can be called recursively.

## Function Definitions

A function definition defines an MAST function.

function_definition ::=
function_header "{" function_body "}" eos

In the absence of a function declaration the function definition acts as the declaration.

### Function Header

The function header defines the name of the function, its formal arguments (if any), and the values returned by the function.

function_header ::=
function_header_definition *{* function_header_sentence *}*

function_header_definition ::=
*{* function_attribute *}* **function** result_indication =

identifier ( *[* function_argument_list *]* ) eos

function_attribute ::=
**encrypted**
| **foreign**

result_indication ::=
untyped_result_indication
| typed_result_indication

untyped_result_indication ::=
identifier
| inline_group

typed_result_indication ::=
variable_declaration
| ( variable_declaration_list )

variable_declaration_list ::=
variable_declaration *{* , variable_declaration *}*

function_argument_list ::=
argument_list
| variable_declaration_list

function_header_sentence ::=
function_header_declarative_item
| eos

function_header_declarative_item ::=
variable_declaration

Each variable declaration in a typed result indication or a variable declaration list must declare exactly one variable.

The identifier in the function header definition denotes the function. The function attributes, if present, are said to *decorate* the function. It is an error if a function is decorated in a function header with the **foreign** attribute.

The result indication of the function header definition defines the profile of the result of the function (see Common Types). If the result indication is an untyped result indication that is an identifier, or if it is a typed result indication that is a variable declaration, then the result of the function is a single value of the type of the object. Otherwise, the result of the function is a group. If the result indication is a typed result indication that is a variable declaration list enclosed in parentheses, then the group constituents of the group defined by the typed result indication are the variables declared by the variable declarations of the variable declaration list.

If the result indication of the function header definition is an untyped result indication that is an identifier, then this identifier must be declared by a variable declaration that is a function header declarative item. Similarly, if the result indication is an untyped result indication that is an inline group, then each simple name that is a group constituent of the inline group must be declared by a variable declaration that is a function header declarative item. It is an error if a variable declaration that declares an object that is part of the result indication includes an initial value expression.

If the function argument list is a variable declaration list, then each variable declaration declares a formal argument of the function. The initial value expression, if any, of the variable declaration must be a locally constant expression.

A function is said to be *pure* if a function call calling the function with the same values as actual arguments always returns the same result. The function is *impure* otherwise.

### Examples

```
function r = f1(n)              # A function with a single
number r, n                     # argument returning a
{...}                           # single value

function number r =             # An alternative way to
       f1(number n)             # define the same function
{...}

struc s1 { number r, i; }       # A function with a single
function (a, b) = f2(c)         # argument of an array type
number a, b                     # returning two scalar
struc s1 c[2]=[(0,0),(1,1)]     # values
{...}
```

### Notes

1. The function attribute **encrypted** has no meaning in the MAST language. See Special Attributes.
2. The function attribute **foreign** can only appear in a function declaration.
3. A variable declaration that is a function header declarative item can declare more than one variable.

### References

Variable Declarations, Argument Lists, Inline Groups

### Rationale

Since type compatibility is defined by name equivalence, it does not make sense to declare types and units in a function header.

## Function Body

The function body defines the algorithm implemented by the function and the result of the function.

```
function_body ::=
        function_sentence { function_sentence }

function_sentence ::=
          function_body_declarative_item
        | function_statement
        | eos

function_body_declarative_item ::=
          function_declaration
        | type_declaration
        | unit_declaration
        | variable_declaration
        | group_declaration
        | function_definition

function_statement ::=
          assignment_statement
        | conditional_statement
        | compound_statement
        | return_statement
        | loop_statement
        | exit_statement
        | next_statement
```

The algorithm performed by the function is defined by the sequence of function statements in the function body. The execution of the function body consists of executing the sequence of function statements. It is an error if any subelement of the result has not been assigned a value when the execution of the function body completes. It is also an error if a name that denotes a formal argument of the function, or a subelement thereof, is assigned a value in the function body. Finally, a function is erroneous if the value of a subelement of a formal argument of the function is changed by a function call calling a foreign function.

**Notes**

1. The language does not define how function arguments are passed into the function (by value, by address, etc.).
2. A foreign function call may change the value of an object if the function call has an actual argument that is itself a function call calling the predefined function ADDR whose actual argument is the object.

### Rationale

The argument passing mechanism has not been defined to allow an implementation to choose the mechanism best suited for each argument type. As a consequence, and to provide consistent results for all types, assignments to function arguments are not allowed.

The requirement that the function result be completely defined when a function call returns helps prevent hard to find errors.

## Function Declarations

A function declaration declares a function.

function_declaration ::=
**foreign** identifier_list eos
| *{* function_attribute *}* type_indication function_declarator_list eos

identifier_list ::=
identifier *{* , identifier *}*

type_indication ::=
extended_type_mark
| ( extended_type_mark_list )

function_declarator_list ::=
function_declarator *{* , function_declarator *}*

function_declarator ::=
identifier ( *[* extended_variable_declaration_list *]* )

extended_type_mark_list ::=
extended_type_mark *{* , extended_type_mark *} [* , ... *]*
| ...

extended_type_mark ::=
type_mark *[* "[" index_constraint *{* , index_constraint *}* "]" *]*

extended_variable_declaration_list ::=
variable_declaration_list *[* , ... *]*
| ...

There are two forms of function declarations.

The first form of a function declaration declares each identifier in the identifier list as the name of a function decorated with the **foreign** attribute; the decorated identifier denotes the foreign function. The result of a foreign function declared using this form of declaration is a group with an unspecified profile (see Common Types). The argument profile of the foreign function is unspecified.

The second form of a function declaration declares the identifier in each function declarator of the function declarator list as the name of a function whose result is one or more values with a profile defined by the type indication of the function declaration. If the type indication is an extended type mark list enclosed in parentheses, and if the extended type mark list ends in an ellipsis (...), then the portion of the result profile corresponding to the ellipsis is unspecified. The identifier of each function declarator is decorated with the specified function attributes, and the decorated identifier denotes the function. If present, the extended variable declaration list of a function declarator defines the formal arguments and hence the argument profile of the function; the argument profile is unspecified otherwise. If the extended variable declaration list ends in an ellipsis, then the portion of the argument profile corresponding to the ellipsis is unspecified. It is an error if the result profile or the argument profile of a function that is not decorated with the **foreign** attribute is unspecified or contains a portion that is unspecified. A function declaration that is not decorated with the **foreign** attribute is erroneous if the profile of the result is different from the profile of the result of the corresponding function definition, or if the argument profile is different from the argument profile of the corresponding function definition.

Each extended type mark in an extended type mark list defines the type of one value of the result profile of the function. If the extended type mark does not contain an index constraint, then the type of the value is the type denoted by the type mark. Otherwise, the extended type mark defines an array type, as follows. If the type mark denotes an array type, then the element type of the array type defined by the extended type mark is the element type of the array type denoted by the type mark. Otherwise, the element type of the array type is the type denoted by the type mark. The index constraints of the extended type mark define index ranges and index types of the array type in the order in which the index constraints appear. If the type mark denotes an array type, then the array type defined by the extended type mark has at additional index positions the index ranges and index types of the array type denoted by the type mark in the order in which they have been defined for that array type. It is an error if the type mark (see Derived Units) of an extended type mark is a type definition.

## Examples

```
foreign myfunc1, myfunc2          # Declaration of two
                                  # foreign functions with
                                  # unspecified argument
                                  # profile and unspecified
                                  # result profile

number f1(number)                 # Declaration of function
                                  # f1 defined in Function Header

(number, number) =                # Declaration of function
    f2(struc s1 c[2] =            # f2 defined in Function Header
        [(0,0),(1,1)])

foreign string myfunc3()          # Declaration of a foreign
                                  # function with unspecified
                                  # argument profile returning
                                  # a single value of type
                                  # string
```

## Notes

1. The result of a foreign function declared using the first form can have a different profile in different function calls.
2. A foreign function whose argument profile includes an unspecified portion can be called with a different number of actuals in different function calls, and the actuals at a particular argument position in the unspecified portion of the argument profile can have different types in different function calls. Similarly, a foreign function whose result profile includes an unspecified portion can return a different number of values in different function calls, and the values at a particular position in the unspecified portion of the result profile can have different types in different function calls.
3. The names of foreign functions must be unique among all declarative regions in a design.
4. The following three foreign function declarations are equivalent:

   **foreign** xyz
   **foreign** (...) xyz()
   **foreign** (...) xyz(...)

5. The name of a formal argument in a function declaration may be different from the name of the corresponding formal argument in the function definition. It is arbitrary if the function declared by the function declaration is a foreign function.
6. The initial value expression of a formal argument in a function declaration may be different from the initial value expression of the corresponding formal argument in the function definition.

**References**

Index Constraints

## Special Purpose Functions

Functions for special purpose applications must satisfy certain conditions. This section describes these conditions.

### Resolution Functions

A resolution function defines how multiple values driving a net are resolved into a single value. A resolution function is declared as part of the declaration of a resolved unit (see Enumeration Unit Declarations and Derived Unit Declaration). It is called during the simulation cycle by the simulator and is typically never called by a model directly.

A resolution function must be a function whose result is a single value. The type of the result must be type compatible with the base type of the resolved unit, and the argument profile of the resolution function must be a one-dimensional unconstrained array whose element type is the type of the result of the function.

Before a resolution function is called to determine the resolved value of a net, the values driving the net are placed into the argument array in an order defined by the implementation. The length of the array is the number of values driving the net. When the execution of the resolution function completes, the value returned by the resolution function becomes the resolved value of the net.

### Limiting Functions

A limiting function is a foreign function whose purpose is to limit the change of an independent variable (see section 11.3.1) from one iteration to the next.

Limiting functions are called by the simulator during the determination of an analog solution point. The semantics of limiting functions and the protocol of calling such functions are described in section C.1.3.

### References

Nonlinearity Specification, Limiting Functions

# 3

# Types and Units

*This chapter describes MAST types, both scalar and composite, and units.*

The type is a characteristic of an object or value through which the compatibility of the object or value with other objects or values is established. Each type includes a set of operations. A unit is a set of attributes of a scalar type.

There are two categories of types. A *common type* is characterized by a set of values. A *pin type* is characterized by the absence of values.

Within each type category there are two kinds of types.

- *Scalar* types are atomic; they cannot be further decomposed. The scalar common types are the predefined types INTEGER, NUMBER and STRING and types defined by an enumeration of their values. Scalar pin types are types with a scalar across aspect and a scalar through aspect.
- *Composite* types are composed of elements of simpler types of the same type category. Composite common types are array types, structure types, and union types. Composite pin types are array pin types and structure pin types.

An *element* is a constituent of a composite type. A *subelement* is either a scalar, or an element, or an element of a subelement.

A composite type must not have subelements whose type is the composite type itself.

## Common Types

type_definition ::=
scalar_type_definition
| composite_type_definition

Each object of a common type has a value. The set of operations of a common type includes the predefined operators (see Operators), functions that have arguments or a result of the type, and the operations inherent in the following:

- an assignment (in assignment statements and declarations)
- a selected name, an indexed name, or a slice name
- a structure aggregate, a union aggregate, an array aggregate, or a structure overlay
- an implicit type conversion of a value to the corresponding value of a compatible type
- a numeric literal, a string literal, or the literals **undef** and **inf**

In this manual the term type is used in place of the term common type if no ambiguity exists.

The *profile* of an ordered collection of objects or values is the types of the objects or values in the order defined by the collection. The *cardinality* of an ordered collection of objects or values is the number of types in the profile of the collection.

### Scalar Common Types

Scalar common types consist of enumeration types and the predefined types INTEGER, NUMBER, and STRING. Enumeration types and type INTEGER are called *discrete types*. The types INTEGER and NUMBER are called *numeric types*. All scalar common types are ordered; that is, the relational operators are defined for their values.

scalar_type_definition ::=
enumeration_type_definition

Each scalar common type has a *range*, which describes the set of values that are representable by the type.

## Integer

The predefined type INTEGER provides a subset of the integer numbers. The range of INTEGER is implementation dependent, but is guaranteed to include the range -2,147,483,647 to +2,147,483,647 and the value **undef**.

## Number

The predefined type NUMBER provides an approximation to the real numbers. The range of NUMBER is implementation dependent, but is guaranteed to include the range -1.0E+300 to +1.0E+300 and the values **undef** and **inf**.

## String

The predefined type STRING holds textual information. The range of STRING includes any sequence of zero or more graphic characters (see Character Set) or white space characters, and the value **undef**. A string value, other than the value **undef**, has an index range whose lower bound is 1 and whose upper bound is the number of characters in the string value. The index range of a string whose value is **undef** is undefined.

## Enumeration Types

An enumeration type definition defines an enumeration type.

enumeration_type_definition ::=
**enum** *[ tag*_identifier *]* "{" enumeration_literal
*{* , enumeration_literal *}* "}"

enumeration_literal ::= identifier

The optional identifier following the keyword **enum** is said to be the *tag* of the enumeration type.

The identifiers representing the enumeration literals in an enumeration type definition must be distinct within the enumeration type definition. Each identifier is the declaration of the corresponding enumeration literal.

Each enumeration literal has a corresponding *enumeration value*. The enumeration values corresponding to the enumeration literals of an enumeration type span consecutive integers in the order in which the enumeration literals appear in the enumeration type definition.

The range of an enumeration type includes the enumeration literals of the enumeration type definition and the value **undef**.

An enumeration literal is said to be *overloaded* if the corresponding identifier is specified in more than one enumeration type definition. The type of an enumeration literal that is a primary in an expression is determined by using the rules described in Overload Resolution.

**Examples**

```
enum yesno { _y, _n }   # an enumeration type whose name
                        # is enum yesno that has two
                        # enumeration literals

enum { _n, _p }         # an anonymous enumeration type
                        # with two enumeration literals.
                        # The enumeration literal _n is
                        # distinct from _n of type enum
                        # yesno
```

## Composite Common Types

Composite common types define collections of values. They include values of structure types, union types, and array types.

composite_type_definition ::=
  structure_type_definition
| union_type_definition

An object of a composite common type represents a collection of objects, one for each element of the composite common type. The elements of a composite common type may be of either a scalar common type or a composite common type. Thus, an object of a composite common type ultimately represents a collection of objects of a scalar common type: its scalar subelements.

**Note**

Array types are defined implicitly rather than by a composite type definition.

## Structure Types

A structure type is a composite common type whose elements are named. A structure type definition defines a structure type.

structure_type_definition ::=
**struc** *[ tag*_identifier *]* "{" *{* format_effector *}*
element_declaration *{* format_effector *}*
*{* element_declaration *{* format_effector *} }* "}"

element_declaration ::=
variable_declaration

The optional identifier following the keyword **struc** is said to be the *tag* of the structure type.

Each element declaration declares one element of the structure type for each declarator of the corresponding variable declaration. If the variable declaration contains an initial value expression (see Object Declarations), then the expression must be a locally constant expression (see Locally Constant Expressions). The *default value* of an element is the initial value of the corresponding variable. It is an error if a name that denotes an element of the structure type appears in an expression in an element declaration of the structure type definition.

The structure type defined by a structure type definition consists of the elements declared by the element declarations in the order in which they appear in the structure type definition. A value of a structure type is the composite of the values of its elements.

It is an error if some elements of a structure type are of a resolved unit (see Enumeration Unit Declarations and Derived Unit Declaration) and other elements are not of a resolved unit.

**Note**

It is a consequence of these rules that the default value or the size of an element of a structure type cannot depend on the value of any other element.

### Examples

```
struc complex {          # A definition for type struc
   number real, imag     # complex with two elements
  }                      # named real and imag, both of
                         # type number

struc tree {             # A definition for type struc
   string name           # tree. Its elements are:
   enum {green, red}\    #  name of type string
      color = green      #  color of an enumeration type
     }                   # color has an initial value
                         # expression

unit {"V", "Volts",      # A declaration of a physical
   "voltage"} v          # unit named v


struc bus {              # A definition for type struc
   variable v strobe     # bus whose elements have units
   variable v data[8]
  }
```

### References

Variable Declarations, Enumeration Unit Declarations, Derived Unit Declaration, Constant Expressions

## Union Types

A union type is a composite common type whose elements are named. A union type definition defines a union type.

union_type_definition ::=
**union** *[* *tag*_identifier *]* "{" *{* format_effector *}*
element_declaration *{* format_effector *}*
*{* element_declaration *{* format_effector *} }* "}"

The optional identifier following the keyword **union** is said to be the *tag* of the union type.

The elements of a union type are said to be the *alternatives* of the union type; each element declaration declares an alternative. If the declaration of an alternative contains an initial value expression, then the expression must be a locally constant expression. It is an error if a name that denotes an alternative

of the union type appears in an expression in an element declaration of the union type definition.

The union type defined by a union type definition consists of the alternatives declared by the element declarations of the union type definition. A value of a union type is the value of one of its alternatives; this alternative is said to be the *current alternative*. The value is **undef** if there is no current alternative.

**Notes**

1. It is a consequence of these rules that the initial value or the size of an alternative of a union type cannot depend on the value of any other alternative.
2. The initial value of an alternative, although allowed, does not have any effect and cannot be used when defining the value of an object of a union type.

## Array Types

An array type is a composite common type whose elements all have the same common type. The name of an element of an array object is an indexed name consisting of the name of the object and one or more index values of a discrete type.

Array types are anonymous types defined implicitly through the declaration of an array object (see Object Declarations). Each array type is characterized by the type of its elements, the number of indices and their order, and the type and range of each index. The type of the elements is said to be the *element type* of the array type. The number of indices is said to be the *dimensionality* of the array type. The range of an index is called its *index range*. The *length* of an index range is the number of values in the index range. Each index range has a corresponding *normalized index range* whose lower bound is 1 and whose upper bound is the length of the index range.

An array type is said to be *unconstrained* if one of its index ranges is assumed (see Index Constraints).

A one-dimensional array type has a distinct element for each value in the range of its index. A multi-dimensional array type has a distinct element for each possible sequence of index values formed by selecting one value in the index range of each index.

A value of an array type is the composite of the values of its elements.

**Index Constraints** An index constraint defines the type and index range of an index of an array object or a slice.

index_constraint ::=
*[* lower_bound : *]* upper_bound
| *enumeration*_type_mark

lower_bound ::=
expression

upper_bound ::=
expression
| *

For an index constraint specified by a type mark, the type mark must denote an enumeration type.

For an index constraint specified with an upper bound and lower bound, if the upper bound is specified as an expression, then the expression must be a simple expression of either a numeric type or an enumeration type. The type of the upper bound is INTEGER if the expression is of a numeric type, or the type of the expression otherwise. An upper bound specified by an asterisk (*) is said to be *assumed*.

The expression specifying the lower bound, if present, must be a simple expression of either a numeric type or an enumeration type. The type of the lower bound is INTEGER if the expression is of a numeric type, or the type of the expression otherwise. If the lower bound is not specified, then it is determined as follows:

- if the upper bound is either assumed or of type INTEGER, then the type of the lower bound is INTEGER and its value is 1.
- if the upper bound is of an enumeration type, then the type of the lower bound is the same enumeration type and its value is the enumeration literal with the smallest enumeration value of the enumeration type.

It is an error if the upper bound and lower bound are both specified by an expression and their types are different.

The type of an index constraint is determined as follows. If the index constraint is specified by a type mark, then its type is the type denoted by the type mark. Otherwise, the type of the index constraint is the type of the lower bound.

The index range of an index constraint is determined as follows.

- if the index constraint is specified by a type mark, then its index range is the range of the enumeration type denoted by the type mark.
- if the upper bound is specified by an expression, then the index range of the index constraint extends from the lower bound to the upper bound of the index constraint, both inclusive.
- if the upper bound is assumed, and if the index constraint is part of a slice name that is a primary in an expression, then the index range of the index constraint extends from the lower bound of the index constraint to the upper bound of the object denoted by the prefix of the slice name, both inclusive.
- otherwise, the index range of the index constraint is said to be *assumed*.

It is an error if the value of the upper bound or lower bound is **undef**.

**References**

Slice Names

## Units

A unit is a scalar common type decorated with attributes specific to the kind of the unit. Each unit has a *base type* through which its type compatibility is established.

### Physical Units

A physical unit definition defines a physical unit. Its base type is NUMBER.

physical_unit_definition ::=
**unit** "{" *string*_expression , *string*_expression , *string*_expression "}"

Each string expression in the physical unit definition defines the value of an attribute of the unit. The expression must be a locally constant expression.

**Note**

The attributes of a physical unit have no meaning in the MAST language. Their values are intended to be used as follows. The first attribute is an abbreviation for the symbol associated with the physical unit (e.g. "V"). The second attribute is the name of the symbol associated with the physical unit (e.g. "Volt"). The

third attribute is the name of the dimension associated with the physical unit (e.g. "voltage").

### Predefined Physical Units

The predefined physical units are ACROSS and THROUGH.

## Enumeration Units

An enumeration unit definition defines an enumeration unit. The base type of an enumeration unit is the corresponding enumeration type (see Enumeration Unit Declarations).

enumeration_unit_definition ::=

**unit state** "{"
enumeration_literal , string_literal , string_literal , string_literal
*{* , enumeration_literal , string_literal , string_literal , string_literal *}*
"}"

The identifiers representing the enumeration literals in an enumeration unit definition must be distinct within the enumeration unit definition. Each identifier is the declaration of the corresponding enumeration literal.

Each string literal following an enumeration literal defines the value of an attribute of the enumeration literal.

**Note**

The attributes of an enumeration unit have no meaning in the MAST language. Their values are intended to be used as follows. The first attribute specifies how the enumeration value is to be used in boolean expressions (outside the MAST language); its value must be either "0", or "1", or "X", or "x". The second attribute specifies how the enumeration value is to be printed. The third attribute specifies how the enumeration value is to be plotted; its value is case insensitive and must be either "low", or "high", or "middle", or "unknown".

## Derived Units

A derived unit definition defines a unit that is derived from an existing unit or from a type.

derived_unit_definition ::=
**unit** unit_mark

unit_mark ::=
*unit*_name
| type_mark

type_mark ::=
**integer**
| **number**
| **string**
| type_definition
| type_reference

type_reference ::=
**enum** *enumeration*_tag
| **struc** *structure*_tag
| **union** *union*_tag
| qualified_name

If the unit mark of a derived unit definition denotes a unit, then the unit must be a physical unit or a derived unit. The base type of the derived unit is the base type of the unit denoted by the unit mark, and the attributes of the derived unit are the same as the attributes of the unit denoted by the unit mark. Otherwise, the base type of the derived unit is the type denoted by the unit mark. The type must be a scalar type other than type STRING.

## Pin Types

An object of a pin type has no value. The set of operations of a pin type includes the operations inherent in a selected name, an indexed name, a slice name, or a branch name.

Each pin type implies a *branch type*, which is a common type.

**References**

Selected Names, Branch Names, Indexed Names, Slice Names

### Scalar Pin Types

A scalar pin type definition defines a scalar pin type.

scalar_pin_type_definition ::=
across_aspect through_aspect
| through_aspect across_aspect

across_aspect ::=
**across** *physical_unit*_name

through_aspect ::=
**through**

*_physical_unit*name

The unit denoted by the physical unit name of the across aspect is called the across unit of the scalar pin type. Similarly, the unit denoted by the physical unit name of the through aspect is called the through unit of the scalar pin type. It is an error if the across unit and the through unit of a scalar pin type are the same.

The branch type implied by a scalar pin type is the predefined type NUMBER.

## Predefined Scalar Pin Types

The only predefined scalar pin type is pin type NEUTRAL. Its across unit is ACROSS and its through unit is THROUGH.

---

## Composite Pin Types

Composite pin types define collections of pins. They include structure pin types and array pin types.

An object of a composite pin type represents a collection of pin objects, one for each element of the composite pin type. The elements of a composite pin type may be of either a scalar pin type or a composite pin type. An object of a composite pin type thus ultimately represents a collection of objects of a scalar pin type, its scalar subelements.

## Structure Pin Types

A structure pin type is a composite pin type whose elements are named. A structure pin type definition defines a structure pin type.

structure_pin_type_definition ::=
**struc** *[ tag*_identifier *]* "{" *{* format_effector *}*
*element*_pin_declaration *{* format_effector *}*
*{* element_pin_declaration *{* format_effector *} }* "}"

The optional identifier following the keyword struc is said to be the tag of the structure pin type.

Each element pin declaration declares an element of the structure pin type.

The structure pin type defined by a structure pin type definition consists of the elements declared by the element pin declarations in the order in which they appear in the structure pin type definition.

The branch type implied by a structure pin type is an anonymous structure type. For each element of the structure pin type there is a matching element of the structure type with the same name; the type of the matching element is the branch type implied by the pin type of the element of the structure pin type.

## Array Pin Types

An array pin type is a composite pin type whose elements all have the same pin type. The name of an element of an array pin object is an indexed name consisting of the name of the array pin object and one or more index values of a discrete type.

Array pin types are anonymous pin types defined implicitly through the declaration of an array pin object (see Pin Type Declarations). Each array pin type is characterized by the pin type of its elements, the number of indices and their order, and the type and range of each index. The pin type of the elements is said to be the *element pin type* of the array pin type. The number of indices is said to be the *dimensionality* of the array pin type.

A one-dimensional array pin type has a distinct element for each value in the range of its index. A multi-dimensional array pin type has a distinct element for each possible sequence of index values formed by selecting one value in the index range of each index.

The branch type implied by an array pin type is an array type whose element type is the branch type implied by the element pin type of the array pin type and whose index ranges and index types are the same as the index ranges and index types of the array pin type.

# Type and Unit Compatibility

This section describes the rules that define when two or more common types, units or pin types are compatible.

Each type has an associated *base name*. The base name of a predefined type is the name of the type. The base name of an enumeration type, structure type or union type whose type definition contains a tag is the name of the type. The base name of a scalar pin type is the name of the pin type. The base name of a structure pin type whose structure pin type definition contains a tag is the name of the structure pin type. The base name of an anonymous type is a unique identifier.

**References**

Type Declarations, Pin Type Declarations

## Compatibility of Common Types

A numeric type is type compatible with any other numeric type. The predefined type STRING is type compatible with itself. Two enumeration types are type compatible if their enumeration type definitions define the same enumeration literals in the same order.

Two structure types are type compatible if they have the same fully qualified base name. Two union types are type compatible if they have the same fully qualified base name. Two array types are type compatible if they have the same dimensionality, the normalized index ranges at corresponding index positions are either the same or one of them is assumed, and the element types of the two array types are type compatible.

Two objects are type compatible if their types are type compatible.

**References**

Qualified Names

## Unit Compatibility

A physical unit is unit compatible with itself and with the predefined physical units ACROSS and THROUGH. An enumeration unit is unit compatible with itself. Additionally, each unit is unit compatible with its base type.

Two structure types are unit compatible if they are type compatible. Two union types are unit compatible if they are type compatible. Two array types are unit compatible if they are type compatible and their elements are unit compatible.

Two objects are unit compatible if they are of a composite type and their types are unit compatible, or if they are of a scalar type, both have a unit and their units are unit compatible, or if they are of a scalar type, one or both have no units, and the objects are type compatible.

**Note**

Two structure types or two union types that are type compatible are also unit compatible because type compatibility by name implies that they have the same elements.

## Pin Type Compatibility

Two scalar pin types are pin type compatible if they have the same fully qualified base name. Additionally, a scalar pin type is pin type compatible with the predefined pin type NEUTRAL.

Two structure pin types are pin type compatible if they have the same fully qualified base name. Two array pin types are pin type compatible if they have the same dimensionality, the normalized index ranges at corresponding index positions are the same, and the element pin types of the two array pin types are pin type compatible.

Two pins are pin type compatible if their pin types are pin type compatible.

**References**

Qualified Names

## Supertypes

The *supertype* implied by the two types is defined as follows.

The supertype implied by two numeric types is INTEGER if both types are INTEGER, and NUMBER otherwise. The supertype implied by two string types is STRING. The supertype implied by two array types that are type compatible is an array type that is type compatible with the two array types and whose index range at each index position is normalized and not assumed. The supertype implied by two enumeration types, structure types or union types that are type compatible is the type denoted by the base name of the types. The supertype implied by a numeric type and a composite type with scalar

subelements of a numeric type is a composite type that has, for each subelement of the implying composite type, a matching subelement with the same name, if applicable, and the type of each scalar subelement of the implied composite type is the supertype implied by the numeric type and the type of the corresponding scalar subelement of the implying composite type.

# 4

# Declarations

*This chapter describes MAST declarations of named entities.*

The language defines several kinds of named *entities* that are declared by declarations, either explicitly or implicitly.

```
declaration_statement ::=
    function_declaration
  | type_declaration
  | unit_declaration
  | pin_type_declaration
  | object_declaration
  | group_declaration
  | function_definition
  | template_definition
```

Each form of declaration, except a type declaration and a structure pin type declaration, associates an identifier or a decimal literal with an entity. The identifier or decimal literal is said to be *declared* by the declaration. Within the scope of a declaration (see Scope of Declarations) there are places where the identifier or decimal literal is sufficient to refer to the entity; these places are defined by the visibility rules (see Visibility). At such places the identifier or decimal literal is said to be the *name* of the entity. The name is said to *denote* the entity. The name of a type is formed using the rules described in Type Declarations. The name of a structure pin type is formed using the rules described in Pin Type Declarations.

Template definitions are described in Design Entities. Function declarations and function definitions are described in Functions. All other declaration statements are described in this section.

**Note**

A decimal literal can be associated with an entity only by an implicit declaration.

## Type Declarations

A type declaration declares a type. Only enumeration types, structure types and union types can be declared by type declarations.

type_declaration ::=
type_definition eos

It is an error if the type definition in a type declaration does not contain a tag.

The name of an enumeration type whose enumeration type definition contains a tag is **enum** followed by the tag. Similarly, the name of a structure type whose structure type definition contains a tag is **struc** followed by the tag. Finally, the name of a union type whose union type definition contains a tag is **union** followed by the tag. An enumeration type, structure type, or union type whose corresponding type definition does not contain a tag is anonymous.

## Unit Declarations

A unit declaration declares a unit.

### Physical Unit Declarations

unit_declaration ::=
physical_unit_declaration
| enumeration_unit_declaration
| derived_unit_declaration

A physical unit declaration declares a physical unit.

```
physical_unit_declaration ::=
   physical_unit_definition identifier eos
```

The simple name declared by a physical unit declaration denotes the physical unit.

### Examples

unit { "A", "Ampere", "current" } i

unit { "m/s", "meter/second", "velocity" } vel

unit { "mph", "miles per hour", "velocity" } mph

## Enumeration Unit Declarations

An enumeration unit declaration declares an enumeration unit and defines and declares an enumeration type corresponding to the enumeration unit.

enumeration_unit_declaration ::=
enumeration_unit_definition identifier = enumeration_default_value
*[* "{" resolution_indication "}" *]* eos

enumeration_default_value ::= simple_name

resolution_indication ::=
**conflict_resolution** : function_declaration

The identifier declared by an enumeration unit declaration denotes the enumeration unit. The enumeration default value must denote one of the enumeration literals of the enumeration unit definition.

The enumeration type corresponding to the enumeration unit is defined by an equivalent enumeration type definition. The tag of the equivalent enumeration type definition is the identifier of the enumeration unit declaration, and the name of the enumeration type is **enum** followed by the tag. The enumeration literals of the equivalent enumeration type definition are the enumeration literals of the enumeration unit in the order in which they appear in the enumeration unit definition.

A resolution indication declares a resolution function and associates it with a unit. It is an error if the function declaration of a resolution indication declares more than one function. A resolution indication is erroneous if the function does not meet the requirements of a resolution function (see Resolution Functions).

A unit declaration that includes a resolution indication is said to be a *resolved unit*. Each state declared to be of that unit will be resolved, if necessary, by the resolution function declared by the resolution indication.

### Derived Unit Declaration

A derived unit declaration declares a derived unit.

derived_unit_declaration ::=
derived_unit_definition identifier *[* = default_value *]*
*[* resolution_indication *]* eos

default_value ::= literal

The identifier declared by a derived unit declaration denotes the derived unit. The default value, if present, must be a literal whose type is type compatible with the base type of the derived unit.

**Examples**

```
enum l4 { l4_0, l4_1, l4_x, l4_z }     # a resolved unit
unit enum l4 logic_4 = l4_x \          # whose base type
      conflict_resolution: \           # is an enumera-
      foreign l4cnfr                   # tion type

unit vel vel_r \                       # a resolved unit
      conflict_resolution: \           # derived from
      foreign number nucnfr()          # unit vel
```

## Pin Type Declarations

A pin type declaration declares a pin type.

pin_type_declaration ::=
scalar_pin_type_declaration
| structure_pin_type_declaration

### Scalar Pin Type Declarations

A scalar pin type declaration declares a scalar pin type.

scalar_pin_type_declaration ::=
**pin** identifier scalar_pin_type_definition eos

The simple name declared by a scalar pin type declaration denotes the scalar pin type.

### Structure Pin Type Declarations

A structure pin type declaration declares a structure pin type.

structure_pin_type_declaration ::=
**pin** structure_pin_type_definition eos

The name of a structure pin type whose structure type definition contains a tag is **struc** followed by the tag. A structure pin type whose structure pin type definition does not contain a tag is anonymous.

## Objects

An *object* is an entity of a given common type or pin type. Objects of a common type also have a value. An object is one of the following:

- An object declared by an object declaration, whether explicitly or implicitly
- A formal argument of a foreign function

In addition, an element or slice of an object is itself an object but not an entity.

There are six classes of objects: parameters, variables, states, analog variables, pins, and simulator variables. Analog variables are divided into two subclasses: analog system variables and analog local variables. States have two subclasses: event-driven states, and assigned states. The class of an explicitly declared object is specified as part of the object declaration. The class of an implicitly declared object is derived from the use of the object. The class and subclass of a subelement of an object of a composite type or pin type is the class and subclass of the object.

### Object Declarations

An object declaration declares an object of a specified class and type or unit. Such an object is said to be *explicitly declared*. An object declaration may also include, either explicitly or implicitly, the specification of one or more attributes. Such attributes are said to *decorate* the objects declared by the object declaration and, if an object is of a composite type, all subelements of the object. Finally, an object declaration may also define an array type or array pin type.

object_declaration ::=
parameter_declaration
| variable_declaration
| state_declaration
| analog_variable_declaration
| pin_declaration
| simulator_variable_declaration

declarator_list ::=
declarator *{* , declarator *}*

declarator ::=
identifier *[* "[" index_constraint *{* , index_constraint *}* "]" *]*
*[* = expression *]*

An object declaration whose declarator list contains two or more declarators is equivalent to a sequence of object declarations in the same order as the declarator list, each object declaration in the sequence containing the text of the object declaration preceding the declarator list followed by a single declarator.

An object declaration (other than a pin declaration) whose type mark is a type definition implicitly declares the type defined by the type definition.

An object declaration (other than a pin declaration) whose declarator contains one or more index constraints defines an array type. If the object declaration contains a unit name or a unit mark that denotes a unit, then the element type of the array type is the base type of the unit denoted by the unit name. Otherwise, if the type mark denotes an array type, then the element type of the array type defined by the object declaration is the element type of the array type denoted by the type mark. Otherwise, the element type of the array type is the type denoted by the type mark. The index constraints of the declarator define index ranges and index types of the array type in the order in which the index constraints appear. If the type mark of the object declaration denotes an array type, then the array type defined by the object declaration has at additional index positions the index ranges and index types of the array type denoted by the type mark in the order in which they have been defined for that array type.

The type of an object (other than a pin) declared by an object declaration is determined as follows. If the object declaration defines an array type, then the type of the object declared by the object declaration is the array type. Otherwise, if the object declaration contains a unit name or a unit mark that

denotes a unit, then the type of the object is the base type of the unit denoted by the unit name. Otherwise, the type of the object is the type denoted by the type mark.

If the declarator of an object declaration contains an expression, then the expression must be a simple expression that is argument constant and type compatible with the type of the object. Such an expression is said to be an *initial value expression*; it defines the *initial value* associated with the object.

In the absence of an initial value expression a default initial value applies. The default initial value of an object of an enumeration unit is the enumeration default value of the enumeration unit. The default initial value of an object of a derived unit is the default value of the derived unit, if present; if the derived unit declaration does not define a default value then the default initial value is dependent on the object class. The default initial value of a scalar analog variable or a scalar branch through variable (see Branch Variable Declarations) is the value 0.0. The default initial value of any other scalar object of class parameter, variable, state, or analog variable, or of an object of a union type is **undef**. The default initial value of an object of a structure type or an array type is the composite of the default initial values of its elements.

**Example**

Given a template definition

```
template type = arg
number arg[10]
{}
```

and the declaration

```
type..arg a20by10[20]
```

a20by10 is now a two-dimensional array whose element type is **number** and whose index ranges are 1:20 and 1:10.

**Note**

In some contexts an initial value expression is required to be a locally constant expression.

## Parameter Declarations

A parameter declaration declares a parameter.

parameter_declaration ::=
*{* parameter_attribute *} [* **parameter** *]* type_mark
declarator_list eos
| *{* parameter_attribute *}* **parameter** unit_name
declarator_list eos

parameter_attribute ::=
**export**
| **external**
| **const**

The value of a parameter cannot be modified after elaboration.

A parameter declaration that decorates a parameter with the **export** attribute must be a template header declarative item. A parameter declaration that decorates a parameter with the **external** attribute should be a template header declarative item. A parameter declaration that decorates a parameter with the **const** attribute must not be a template header declarative item.

The identifier associated with a parameter that is decorated with the **external** attribute must not appear in the argument list of a template header definition. Conversely, the identifier associated with a parameter that is decorated with the **export** attribute must appear in the argument list of a template header definition.

The type of a parameter whose declaration is a template header declarative item must be a relaxed locally constant type. The type of a parameter whose declaration is a template body sentence must be a relaxed argument constant type.

A parameter declaration that decorates a parameter with the **const** attribute must include an initial value expression that is a locally constant expression. Similarly, the initial value expression of a parameter that is a template header declarative item must be a locally constant expression. It is an error if a parameter declaration that decorates a parameter with the **external** attribute includes an initial value expression.

### Examples

```
number a, b=3,          # declares three parameters a,
       c[2:4]           # b, c. The initial value of
                        # b is 3. c is of an array
                        # type; its index range is
                        # 2 through 4, its index type
                        # is integer

unit { "m","meter",     # declares a parameter d whose
       "length" } m     # initial value is 2, whose
parameter m d=2         # unit is m and whose type is
                        # the base type of m: number

const number \          # declares a parameter whose
       pi = 3.14159     # value is constant
```

### Note

A parameter declaration that decorates a parameter with the **external** attribute may be a declarative item in a template header or a template body, with the template header the preferred location.

### Rationale

The **const** attribute allows a parameter to be declared as a constant whose value cannot be modified later in an assignment statement or an alter specification.

## Variable Declarations

A variable declaration declares a variable.

variable_declaration ::=
*[* variable_attribute *] [* **variable** *]* type_mark declarator_list eos
| *[* variable_attribute *]* **variable** *unit*_name declarator_list eos

variable_attribute ::=
**const**

A variable declaration that decorates a variable with the const attribute must not be a function header declarative item.

The type of a variable whose declaration appears in a template body must be a globally constant type. The type of a variable whose declaration is a function

header declarative item or a function body declarative item must be a relaxed argument constant type.

## State Declarations

A state declaration declares a state.

state_declaration ::=
*{* state_attribute *} [* mode *]* state unit_mark declarator_list eos

state_attribute ::=
**external**
| **foreign**

mode ::= **input** | **output** | **inout**

A state declaration that decorates a state with the **external** attribute should be a template header declarative item. The identifier associated with a state that is decorated with the **external** attribute must not be the connection definition of a template connection element. A state declaration that decorates a state with the **foreign** attribute must appear as a declarative item in a template body.

It is an error if a state declaration that includes a mode declares a state that is neither a template connection element nor decorated with the **foreign** attribute. It is also an error if a state declaration declares a state that has a subelement of a union type or of the predefined type STRING.

The type of a state that is decorated with the **external** attribute must be a relaxed locally constant type. The type of a state that is a connection element must be a relaxed argument constant type. The type of a state whose declaration is a template body sentence must be a globally constant type.

The state denoted by the longest constant prefix of the second argument of function SCHEDULE_EVENT is said to be a *driving state*. A state is a *scheduled state* if it contains a subelement that is a scheduled state or a driving state or that is the connection actual part in a connection association element whose connection formal part is a scheduled state. A state is an *observed state* if it contains a subelement that is denoted by the first argument of function EVENT_ON or that is the connection actual part in a connection association element whose connection formal part is an observed state. It is an error if a state that is decorated with the **foreign** attribute is neither a driving state nor an observed state.

A state is an *event-driven state* if it is a scheduled state, or an observed state, or a formal connection of a template, or the connection actual part in a

connection association element, or if it is decorated with the **external** attribute. A state is an *assigned state* if its value is updated by the execution of an assignment statement or by an implicit assignment. It is an error if a state is both an event-driven state and an assigned state.

A state is said to be *read* if it is an observed state, or if the name of one of its subelements appears in an expression, or if the name of one of its subelements is denoted by the first argument of one of the functions LAST_VALUE, RAMP, or SLEW.

If a state declaration includes a mode, then one of the following conditions must be true:

- The mode is **input** and the state declared by the state declaration is not a scheduled state.
- The mode is **output** and the state declared by the state declaration is a scheduled state that is not read.
- The mode is **inout** and the state declared by the state declaration is a scheduled state that is also read.

If a state declaration that declares a template connection element or that decorates a state with the **foreign** attribute does not include a mode, then the mode is implied according to these rules.

**Notes**

1. A state declaration that decorates a state with the **external** attribute may be a declarative item in a template header or a template body, with the template header the preferred location.
2. A state may have subelements that are driving states and others that are not.
3. Some states are neither event-driven states nor assigned states.

**Rationale**

The modes are based on a state being scheduled or read rather than scheduled or observed because if the name of a state appears in an expression or as the argument of one of the functions LAST_VALUE, RAMP, or SLEW we are interested in the (possibly resolved) value from the net. This value might not be available (e.g. in the context of conversion models) if the mode did depend on whether the state is scheduled or observed. In the case of the DRIVEN function, however, we are interested in the “local” value, i.e. the interface is not involved.

## Analog Variable Declarations

An analog variable declaration declares an analog variable.

analog_variable_declaration ::=
var_declaration
| ref_declaration
| val_declaration
| branch_variable_declaration

analog_attribute ::=
**export**
| **external**

It is an error if an analog variable declaration declares an analog variable that has a subelement of a union type. It is also an error if any scalar subelement of an analog variable is not of the predefined type NUMBER.

An analog variable declaration that decorates an analog variable with the **export** attribute must be a template header declarative item. An analog variable declaration that decorates an analog variable with the **external** attribute should be a template header declarative item.

An *analog system variable* is an analog variable declared by a var declaration or a ref declaration, or a branch through system variable. An *analog local variable* is an analog variable declared by a val declaration, a branch across variable, or a branch through local variable.

It is an error if an analog local variable or a branch through variable is decorated with the **external** attribute.

**Note**

An analog variable declaration that decorates an analog variable with the **external** attribute may be a declarative item in a template header or a template body, with the template header the preferred location.

**References**

Branch Variable Declarations

**Var Declarations** A var declaration declares an analog system variable of kind var.

var_declaration ::=
*{* analog_attribute *}* var_indication unit_mark declarator_list eos

var_indication ::= *[* **output** *] [* **var** *]*

It is an error if a declarator in the declarator list of a var declaration contains an initial value expression. It is also an error if a var indication includes neither the keyword **output** nor the keyword **var**.

The type of an analog system variable of kind **var** that is decorated with the external attribute must be a relaxed locally constant type. The type of an analog system variable of kind var that is decorated with the **export** attribute must be an argument constant type. The type of an analog system variable of kind **var** that is a connection element must be a relaxed argument constant type. The type of an analog system variable of kind **var** whose declaration is a template body sentence must be a globally constant type.

**Note**

By convention, the keyword **output** is only used in the declaration of template connection elements, where it is often used without the keyword **var**.

**Ref Declarations** A ref declaration declares an analog system variable of kind ref.

ref_declaration ::=
*{* analog_attribute *}* ref_indication unit_mark declarator_list eos

ref_indication ::= *[* **input** *] [* **ref** *]*

It is an error if a declarator in the declarator list of a ref declaration contains an initial value expression. It is also an error if a ref indication includes neither the keyword **input** nor the keyword **ref**.

The type of an analog system variable of kind ref that is decorated with the external attribute must be a relaxed locally constant type. The type of an analog system variable of kind ref that is decorated with the export attribute must be an argument constant type. The type of an analog system variable of kind ref that is a connection element must be a relaxed argument constant type. The type of an analog system variable of kind ref whose declaration is a template body sentence must be a globally constant type.

**Note**

By convention, the keyword input is only used in the declaration of template connection elements, where it is often used without the keyword ref.

**Val Declarations** A val declaration declares an analog local variable of kind **val**.

```
val_declaration ::=
   [ analog_attribute ] val unit_mark declarator_list eos
```

It is an error if a declarator in the declarator list of a val declaration contains an initial value expression.

The type of an analog local variable of kind **val** that is decorated with the **export** attribute must be an argument constant type. The type of an analog local variable of kind val whose declaration is a template body sentence must be a globally constant type.

**Branch Variable Declarations** A branch variable declaration declares a branch variable.

branch_variable_declaration ::=
*[* analog_attribute *]* **branch** branch_definition
*[* , branch_definition *]* eos

branch_definition ::=
identifier = branch_name

A branch variable declaration that contains two or more branch definitions is equivalent to a sequence of branch variable declarations in the same order as the list of branch definitions, each branch variable declaration in the sequence containing the keyword **branch** followed by a single branch definition.

The unit of the branch variable is the unit of the branch denoted by the branch name. Similarly, the type of the branch variable is the type of the branch denoted by the branch name. The type of a branch variable that is decorated with the **export** attribute must be an argument constant type. The type of a branch variable whose declaration is a template body sentence must be a globally constant type.

A branch variable is a *branch across variable* if the branch denoted by the branch name is an across branch. Similarly, a branch variable is a *branch through variable* if the branch denoted by the branch name is a through branch. A branch through variable is a *branch through local variable* if its name, or the names of any of its subelements, appears as the target of an assignment statement or as the name in a contribution statement, but does not appear as a primary in an expression. Otherwise, the branch through variable is a *branch through system variable*.

### References

Assignment Statement, Contribution Statement

## Pin Declarations

A pin declaration declares a pin.

pin_declaration ::=
*{* pin_attribute *}* pin_type_mark declarator_list eos

pin_attribute ::=
**export**
| **external**

pin_type_mark ::=
*pin_type*_name
| structure_pin_type_definition
| pin_type_reference

A pin declaration whose declarator contains one or more index constraints defines an array pin type. The element pin type of the array pin type is the pin type denoted by the pin type mark. The index constraints of the declarator define the index types and index ranges of the array pin type in the order in which the index constraints appear.

A pin declaration whose pin type mark is a structure pin type definition implicitly declares the structure pin type defined by the structure pin type definition.

The pin type of a pin declared by a pin declaration is determined as follows. If the pin declaration defines an array pin type, then the pin type of the pin declared by the pin declaration is the array pin type. Otherwise, the pin type of the pin is the pin type denoted by the pin type mark.

A pin declaration that decorates a pin with the **export** attribute must be a template header declarative item. A pin declaration that decorates a pin with the **external** attribute should be a template header declarative item. The identifier associated with a pin that is decorated with the export or external attribute must not denote a connection element.

The pin type of a pin that is decorated with the **external** attribute must be a relaxed locally constant pin type. The pin type of a pin that is decorated with the **export** attribute must be an argument constant pin type. The pin type of a pin that is a connection element must be a relaxed argument constant pin type. The pin type of a pin whose declaration is a template body sentence must be a globally constant pin type.

It is an error if the declarator of a pin declaration contains an initial value expression.

**Note**

A pin declaration that decorates a pin with the **external** attribute may be a declarative item in a template header or a template body, with the template header the preferred location.

**Predefined Pins** The only predefined pin is the reference pin 0. Its pin type is the predefined pin type NEUTRAL.

## Simulator Variable Declarations

A simulator variable declaration declares a simulator variable.

simulator_variable_declaration ::=
**simvar** identifier_list eos

Each identifier in the identifier list of a simulator variable declaration must denote one of the simulator variables described in Simulator Variables.

---

# Arguments

## Argument Lists

Argument lists provide a way to parameterize a design unit.

argument_list ::=
identifier_list

Each identifier in the argument list defines a formal argument of the design unit.

Each identifier in a *template* argument list must be declared in a parameter declaration that is a template header declarative item. Each identifier in a *function* argument list must be declared in a variable declaration that is a function header declarative item. The initial value expression, if any, of a parameter declaration or variable declaration that declares a formal argument of a design unit must be a locally constant expression.

The *argument profile* is the profile of the formal arguments in an argument list.

**References**

Common Types

### Rationale

The initial value expression of a formal argument must be locally constant to allow for an ordered and efficient elaboration.

## Argument Association Lists

An argument association list establishes a correspondence between the formal arguments of a design unit and actual arguments.

argument_association_list ::=
    argument_association_element *{* , argument_association_element *}*

argument_association_element ::=
    *[* formal = *]* actual

formal ::=
    *_argument*simple_name

actual ::=
    expression

Each argument association element associates one or more simple expressions with a corresponding number of formal arguments of a template argument list or a function argument list. Additionally, an argument association element of an instance argument association list may associate an expression with a parameter of the template denoted by the prefix of the instance name; such a parameter must be decorated with the **external** attribute. The corresponding arguments or parameters are determined either by position or by name.

An argument association element is said to be *named* if the element is specified explicitly by its simple name defined in the template declaration or function declaration; it is said to be *positional* otherwise. For a positional association element the formal argument is implicitly specified by the textual position of the argument association element in the argument association list.

Named association elements may appear in any order, but if named and positional association elements appear in the same argument association list, then any named association element must follow all positional association elements.

The actual expression of an argument association element must be a simple expression, unless the argument association element appears in the argument association list of a function call that calls a function whose argument profile is

unspecified. If the actual expression is not a simple expression, then the argument association element is implicitly substituted by a sequence of argument association elements; the length of the sequence is equal to the cardinality of the actual expression.

After any substitutions, the type of the actual expression in an argument association element must be type compatible with the type of the corresponding formal argument. If a subelement of the formal argument is of an unconstrained array type, then the upper bound of each assumed index range of the subelement is defined by the association such that the length of the index range is equal to the length of the index range of the corresponding subelement of the actual at the same index position.

An argument association list need not include an argument association element for a formal argument if the declaration of the formal argument includes an initial value expression. If the argument association element is not included in the argument association list, then an implicit argument association element associates the initial value expression with the formal argument. It is an error if a formal argument is associated more than once with an actual in an argument association list.

**Note**

A parameter decorated with the **external** attribute cannot be associated with an expression by a positional association element since it is not a member of the template argument list.

## Implicit Declarations

In the absence of a declaration statement an object may be declared by using its simple name or decimal name in a statement that is a template body sentence. Such objects are said to be *implicitly declared*.

## Implicit Declaration of Branch Variables

The appearance of a branch name in a statement implicitly declares a branch variable according to the following rules:

- If the branch name is an across branch name, then a single branch variable is implicitly declared whose name is the branch name and whose unit and type is the unit and type, respectively, of the branch denoted by the branch name.
- If the branch name is a through branch name the following rules hold:
  - If the branch name appears as a primary in an expression, then a single branch variable is implicitly declared whose name is the branch name and whose unit and type is the unit and type, respectively, of the branch denoted by the branch name.
  - If the through branch name appears as the target in an assignment statement or as the name in a contribution statement, then each such appearance implicitly declares a distinct through branch variable whose name is a unique name.

The corresponding declaration is deemed to occur in the major declarative region formed by the template, immediately preceding the statement that contains the branch name or the outermost minor declarative region, if any, that encloses the statement.

**Note**

It is a consequence of these rules that each implicitly declared branch through variable defines a distinct branch between the plus pin and minus pin of its branch name.

**References**

Assignment Statement, Contribution Statement

## Implicit Declaration of Imported Objects

The appearance of an imported name or of a name whose prefix contains a portion that is an imported name in a statement implicitly declares an *imported object* of the same object class, subclass (if any), type or pin type, and unit (if any) as the object denoted by the prefix of the imported name in the instance denoted by the instance name of the imported name. If the object is an analog system variable, then its kind is ref. The imported object is associated with the object denoted by the simple name of the imported name in the instance denoted by the instance name of the imported name. If the object is of class parameter, then the declaration includes an initial value expression that is the

value of the object associated with the imported object. The corresponding declaration is deemed to occur in the major declarative region formed by the template, immediately preceding the statement that contains the reference to the imported object or the outermost minor declarative region, if any, that encloses the statement.

### Other Implicit Declarations

The following language elements may also implicitly declare objects:

- An object declaration may declare a type or a pin type. See Object Declarations and Pin Declarations.
- A connection association element may declare the object denoted by the connection actual part. See Instantiation Statement.
- An assignment statement may declare the object denoted by the target of the assignment statement. See Assignment Statement.

## Group Declarations

A group declaration declares a group, which is an ordered collection of objects.

group_declaration ::=
**group** "{" group_constituent_list "}" identifier eos

group_constituent_list ::=
group_constituent *{* , group_constituent *}*

group_constituent ::= name

The simple name declared by a group declaration denotes the group. The name of each group constituent must denote an object or a group that has been declared in the same major declarative region.

The *canonical group* corresponding to a group is obtained by replacing each group constituent in the group constituent list of the group that itself denotes a group by the canonical group of the group constituent. Thus, a group ultimately represents a collection of objects in an order defined by its corresponding canonical group. It is an error if the same object appears more than once in the canonical group corresponding to a group.

The profile of a group is the profile of its canonical group.

The objects denoted by the group constituents of the canonical group corresponding to a group must meet one of the following conditions:

- The class of each object must be parameter or variable. An object of class variable must be declared either in the major declarative region associated with a function or in a minor declarative region decorated with the **parameters** attribute. Such a group is said to be a *parameter group*.
- The class of each object must be state, or variable, or analog variable, or simulator variable. An object of class variable must be declared in a minor declarative region decorated with the **states** or **values** attribute. Such a group is called a *nonparameter group*.

**Note**

Further restrictions on the constituents of a group exist if the group is the target of an assignment statement. See Assignment Statement.

## Inline Groups

An inline group is an unnamed group.

inline_group ::=
( group_constituent_list )

An inline group is declared by its appearance in a sentence. It is an error if a group constituent of an inline group has not been declared.

# 5

# Specifications

*This chapter describes MAST specifications.*

A specification associates additional information with an entity that either has been previously declared or that is implied or declared by the specification. The specification is said to *relate* to that entity.

A specification that relates to a design unit must appear within the declarative region associated with the design unit. A specification that relates to a unit must appear in a declarative region where the unit is visible. Any other specification must appear in the same major declarative region as the declaration of the entity to which it relates.

## Alter Specification

An alter specification specifies the initial value expression of an object to which it relates.

alter_specification ::=
**alter** specifier_list eos

specifier_list ::=
specifier *{* , specifier *}*

specifier ::=
simple_name = expression

An alter specification whose specifier list contains two or more specifiers is equivalent to a sequence of alter specifications in the same order as the specifier list, each alter specification in the sequence containing the reserved word **alter** followed by a single specifier.

The simple name in a specifier must denote an explicitly declared object for which an initial value expression is a legal part of the declaration and whose object declaration appears in a template body or a function body. The expression must be an argument constant expression whose type is type compatible with the type of the object denoted by the simple name. The value of the expression replaces the initial value of the object denoted by the simple name.

## Control Section Specifications

Control section specifications are specifications that must be decorated with the **control_section** attribute. It is an error if a control section specification is decorated with any other statement attribute.

control_section_specification ::=
dc_help_specification
| noise_source_specification
| collapse_specification
| start_value_specification
| initial_condition_specification
| restart_specification
| device_type_specification
| nonlinearity_specification
| sample_point_specification
| newton_step_specification
| partial_derivative_specification
| small_signal_specification
| stress_measure_specification
| variable_range_specification
| unit_range_specification
| range_set_specification

A control section specification is said to *apply* to an entity if it relates to the entity and has been elaborated.

### DC_Help Specification

The dc_help specification relates to a through branch implied by the dc_help specification and specifies a contribution to the through branch.

dc_help_specification ::=
*[* **control_section** *]*
**dc_help** ( *plus*_pin_aspect , *minus*_pin_aspect ) eos

The plus pin aspect and the minus pin aspect of a dc_help specification must satisfy the rules for the plus pin aspect and the minus pin aspect of a branch name.

The branch name of the through branch implied by the dc_help specification is THROUGH ( plus_pin -> minus_pin ), where plus_pin is the pin denoted by the plus pin aspect and minus_pin is the pin denoted by the minus pin aspect. The contribution specified by the dc_help specification is:

```
THROUGH(plus_pin->minus_pin) += G * ACROSS(plus_pin,
minus_pin)
```

where G is a scalar parameter of type NUMBER whose value is controlled by the simulator. The default value of G is zero.

It is an error if more than one dc_help specification applies to the same through branch.

**References**

Branch Names

## Noise Source Specification

A noise source specification specifies a contribution for a noise analysis only.

noise_source_specification ::=
*[* **control_section** *]* **noise_source** ( *source*_simple_name,
*plus*_pin_aspect *[* , *minus*_pin_aspect *]* ) eos
| *[* **control_section** *]* **noise_source** ( *source*_simple_name,
*asv*_simple_name ) eos

A noise source specification of the first form relates to a through branch implied by the noise source specification. A noise source specification of the second form relates to an analog system variable of kind var.

The source simple name must denote an analog local variable of kind val.

For a noise source specification of the first form, the plus pin aspect and the minus pin aspect must satisfy the rules for the plus pin aspect and the minus pin aspect of a branch name. A noise source specification without a minus pin aspect is equivalent to a noise source specification whose minus pin aspect is

the literal 0. The branch name implied by the noise_source specification is THROUGH ( plus_pin -> minus_pin ), where plus_pin is the pin denoted by the plus pin aspect and minus_pin is the pin denoted by the minus pin aspect. The noise analysis contribution specified by the noise source specification is:

```
THROUGH(plus_pin->minus_pin) += source
```

where source is the analog local variable denoted by the source simple name.

For a noise source specification of the second form, the asv simple name must denote an analog system variable of kind var or a branch through system variable. The noise analysis contribution specified by the noise source specification is:

```
asv -= source
```

where asv is the analog system variable denoted by the asv simple name and source is the analog local variable denoted by the source simple name. It is an error if the asv simple name is not the label of a labeled equation statement.

**Note**

The definitions imply that the source denoted by the source simple name must be type compatible with the through branch (for a noise source specification of the first form) or the analog system variable denoted by the asv simple name (for a noise source specification of the second form).

**References**

Branch Names

## Collapse Specification

A collapse specification specifies an implicit connection between two objects.

collapse_specification ::=
*[* **control_section** *]*
**collapse** ( *plus*_pin_aspect, *minus*_pin_aspect ) eos
| *[* **control_section** *]* **collapse** ( simple_name, simple_name ) eos

A collapse specification of the first form relates to the pin denoted by the plus pin aspect. The plus pin aspect and the minus pin aspect of a collapse specification must satisfy the rules for the plus pin aspect and the minus pin aspect of a branch name.

A collapse specification of the second form relates to the object denoted by the first simple name. Each simple name must denote either a state or an analog system variable of kind var or ref. Both objects must belong to the same object

class, and they must be unit compatible. It is an error if a state whose name appears in a collapse specification is an assigned state.

**References**

Branch Names

## Start Value Specification

A start value specification relates to an analog variable and specifies the initial value of that analog variable.

start_value_specification ::=
*[* **control_section** *]* **start_value** ( analog_name, expression ) eos

analog_name ::=
simple_name
| branch_name

The analog name must denote either a pin or an analog variable whose scalar subelements are independent variables (see Statements Decorated with the Values Attribute). If the analog name denotes a pin, then it implies an across branch name whose plus pin is the pin and whose minus pin is the predefined pin 0, and the start value specification relates to the corresponding branch across variable. The expression must be a globally constant expression that is type compatible with the type of the object denoted by the simple name.

It is an error if more than one start value specification applies to the same analog variable.

## Initial Condition Specification

An initial condition specification relates to an analog variable and specifies the initial condition for that analog variable.

initial_condition_specification ::=
*[* **control_section** *]*
**initial_condition** ( analog_name, expression ) eos

The analog name must denote either a pin or an analog variable whose scalar subelements are independent variables (see Statements Decorated with the Values Attribute). If the analog name denotes a pin, then it implies an across branch name whose plus pin is the pin and whose minus pin is the predefined pin 0, and the initial condition specification relates to the corresponding branch

across variable. The expression must be a globally constant expression that is type compatible with the type of the object denoted by the simple name.

It is an error if more than one initial condition specification applies to the same analog variable.

## Restart Specification

A restart specification relates to one or more states and specifies that the value of these states must be adjusted when a time domain analysis starts (see The Time Domain Initialization Phase).

restart_specification ::=
*[* **control_section** *]* **adjust_on_restart** ( simple_name , simple_name *}* ) eos

Each simple name must denote a state. The corresponding state is said to be an adjustable state.

It is an error if more than one restart specification applies to the same state.

## Device Type Specification

A device type specification relates to a template and specifies two attributes of an instance of the template.

device_type_specification ::=
*[* **control_section** *]* **device_type** ( *device_class*_expression,
*device_subclass*_expression ) eos

Both expressions must be globally constant expressions of type STRING. The device class expression defines the device class attribute of an instance of the template. Similarly, the device subclass expression defines the device subclass attribute of an instance of the template.

It is an error if more than one device type specification applies to the same instance.

**Note**

The device class attribute and the device subclass attribute have no meaning in the MAST language.

## Nonlinearity Specification

A nonlinearity specification specifies the dependent variables and independent variables (see Statements Decorated with the Values Attribute) of a nonlinearity and optionally a limiting function and constant actual arguments for calling the limiting function.

nonlinearity_specification ::=
*[* **control_section** *]* **pl_set** ( *dependent*_set, *independent*_set
*[* , *limiting_function*_name *[* , expression *] ]* ) eos

set ::= analog_name | inline_group

A *set* is an unordered collection of objects. If a set is defined by an inline group or by a simple name that denotes a group, then each group constituent of the group or inline group is a *member* of the set. Otherwise, the set must be defined by a simple name that denotes an object, and the object is the only *member* of the set.

A nonlinearity specification relates to each member of the dependent set.

Each member of the dependent set must denote an analog variable whose scalar subelements are dependent variables. Each member of the independent set must denote either a pin or an analog variable whose scalar subelements are independent variables. If a member denotes a pin, then it implies an across branch name whose plus pin is the pin and whose minus pin is the predefined pin 0. The limiting function name, if present, must denote a foreign function that meets the requirements for a limiting function (see Limiting Functions). The expression, if present, must be a globally constant expression that is either of type NUMBER or of a one-dimensional array type whose element type is NUMBER.

It is an error if more than one nonlinearity specification applies to the same analog variable.

## Sample Point Specification

A sample point specification specifies sample points for independent variables (see Statements Decorated with the Values Attribute). It relates to each scalar subelement of each member of a set.

sample_point_specification ::=
*[* **control_section** *]*
**sample_points** ( *independent_*set, expression ) eos

Each member of the independent set must denote either a pin or an analog variable whose scalar subelements are independent variables. If a member denotes a pin, then it implies an across branch name whose plus pin is the pin and whose minus pin is the predefined pin 0, and the sample point specification relates to the corresponding branch across variable. A sample point specification whose independent set is defined by a group is equivalent to a sequence of sample point specifications. For each sample point specification in the equivalent sequence the independent set denotes one group constituent of the canonical group corresponding to the group, and the expression is the same as the expression of the original sample point specification.

The expression must be a globally constant expression of a one-dimensional array type whose element type is either the predefined type **struc** BREAKPOINT or a user-defined structure type with two elements of type NUMBER whose first element is considered equivalent to element BP of type **struc** BREAKPOINT and whose second element is considered equivalent to element INC of type **struc** BREAKPOINT. The value of the expression must satisfy the following conditions:

- The value of the subelement BP of any array element must be greater than the value of the subelement BP of the immediately preceding array element, if any.
- The value of the subelement BP of one array element must be zero.
- The value of the subelement INC of each array element, except the one whose index value equals the upper bound of the index range of the array, must be positive.
- The value of the subelement INC of the array element whose index value equals the upper bound of the index range of the array must be zero.

For each sample point specification in the equivalent sequence, the sample points of each scalar subelement of the independent variable denoted by the independent set are specified by the expression as follows:

- The value of the subelement BP of each array element, except the array element whose index value equals the upper bound of the index range of the array, defines the left end of an interval.
- The value of the subelement BP of each array element, except the array element whose index value equals the lower bound of the index range of the array, defines the right end of the interval whose left end is defined by the immediately preceding array element.
- Within each interval there is a sample point at bp + k * inc, where bp is the value of the subelement BP of the array element that defines the left end of the interval, inc is the value of the subelement INC of the same array element, and k is any nonnegative integer.

It is an error if more than one sample point specification applies to the same independent variable. If no explicit sample point specification applies to an independent variable, then an implicit sample point specification applies to that variable with an expression whose value is defined by the implementation.

## Newton Step Specification

A newton step specification specifies newton steps for independent variables (see Statements Decorated with the Values Attribute). It relates to each scalar subelement of each member of a set.

newton_step_specification ::=
*[* **control_section** *]* **newton_step** ( *independent*_set,
*increase*_expression *[* , *decrease*_expression *]* ) eos

A newton step specification without a decrease expression is equivalent to a newton step specification whose decrease expression is the same as the increase expression.

Each member of the independent set must denote either a pin or an analog variable whose scalar subelements are independent variables. If a member denotes a pin, then it implies an across branch name whose plus pin is the pin and whose minus pin is the predefined pin 0, and the newton step specification relates to the corresponding branch across variable. A newton step specification whose independent set is defined by a group is equivalent to a sequence of newton step specifications. For each newton step specification in

the equivalent sequence the independent set denotes one group constituent of the canonical group corresponding to the group, and the increase expression and decrease expression are the same as the corresponding expressions of the original newton step specification.

The increase expression and the decrease expression must be globally constant expressions of a one-dimensional array type whose element type is either the predefined type **struc** BREAKPOINT or a user-defined structure type with two elements of type NUMBER whose first element is considered equivalent to element BP of type **struc** BREAKPOINT and whose second element is considered equivalent to element INC of type struc BREAKPOINT. The value of the increase expression and the decrease expression must satisfy the following conditions:

- The value of the subelement BP of any array element must be greater than the value of the subelement BP of the immediately preceding array element, if any.
- The value of the subelement INC of each array element, except the one whose index value equals the upper bound of the index range of the array, must be nonnegative.

For each newton step specification in the equivalent sequence, the increase newton steps of each scalar subelement of the independent variable denoted by the independent set are specified by the increase expression, as follows:

- The value of the subelement BP of each array element of the increase expression defines the right end of an interval and the left end of the subsequent interval. Additionally, the value **-inf** defines the left end of the interval whose right end is the value of the subelement BP of the array element whose index value equals the lower bound of the index range of the array. Similarly, the value **inf** defines the right end of the interval whose left end is the value of the subelement BP of the array element whose index value equals the upper bound of the index range of the array.
- The *increase newton step* of the interval whose left end is the value **-inf** is 0. The increase newton step of all other intervals is the value of the subelement INC of the array element that defines the left end of the interval.

Similarly, for each newton step specification in the equivalent sequence, the decrease newton steps of each scalar subelement of the independent variable

denoted by the independent set are specified by the decrease expression, as follows:

- The value of the subelement BP of each array element of the decrease expression defines the right end of an interval and the left end of the subsequent interval. Additionally, the value **-inf** defines the left end of the interval whose right end is the value of the subelement BP of the array element whose index value equals the lower bound of the index range of the array. Similarly, the value **inf** defines the right end of the interval whose left end is the value of the subelement BP of the array element whose index value equals the upper bound of the index range of the array.
- The *decrease newton step* of the interval whose right end is the value **inf** is 0. The decrease newton step of all other intervals is the value of the subelement INC of the array element that defines the right end of the interval.

It is an error if more than one newton step specification applies to the same independent variable.

## Partial Derivative Specification

A partial derivative specification relates to a variable declared by the partial derivative specification and specifies the value of that variable.

partial_derivative_specification ::=
*[* **control_section** *]* **ss_partial** ( identifier, expression, *wrt*_name ) eos

The partial derivative specification declares the identifier as a variable of type NUMBER in the minor declarative region immediately enclosing the specification. If the partial derivative specification does not occur immediately within a minor declarative region, then the declaration and the specification together form a minor declarative region that is decorated with the statement attribute of the partial derivative specification.

The expression must be a simple expression of type NUMBER. It is an error if a primary in the expression denotes an imported object that is an analog local variable. The wrt name must denote a scalar analog variable or a pin of a scalar pin type. If the wrt name denotes an analog variable, then the partial derivative specification specifies the partial derivative of the expression with respect to the analog variable as the value of the variable denoted by the identifier. Otherwise, the partial derivative specification specifies the partial derivative of the expression with respect to ACROSS(pin) as the value of the variable, where pin is the pin denoted by the wrt name.

It is an error if the name of the variable denoted by the identifier appears in any expression except in the expression of a small signal specification.

## Small-Signal Specification

A small-signal specification relates to a variable declared by the small-signal specification and specifies the value and two attributes of that variable.

small_signal_specification ::=
*[* **control_section** *]* **small_signal** ( identifier, *category*_identifier,
*report*_expression, expression *[* , *wrt*_name *]* ) eos

The small-signal specification declares the identifier as a variable of type NUMBER in the minor declarative region immediately enclosing the specification. If the small-signal specification does not occur immediately within a minor declarative region, then the declaration and the specification together form a minor declarative region that is decorated with the statement attribute of the small-signal specification.

The category identifier defines the category attribute of the variable denoted by the identifier. The report expression defines the report attribute of the variable denoted by the identifier; it must be a globally constant expression of type STRING.

The expression must be a simple expression of type NUMBER. It is an error if a primary in the expression denotes an imported object that is an analog local variable. The wrt name, if present, must denote a scalar analog variable or a pin of a scalar pin type. If the wrt name is not present, then the small-signal specification specifies the value of the expression as the value of the variable. Otherwise, if the wrt name denotes an analog variable, then the small-signal specification specifies the partial derivative of the expression with respect to the analog variable as the value of the variable denoted by the identifier. Otherwise, the small-signal specification specifies the partial derivative of the expression with respect to ACROSS(pin) as the value of the variable, where pin is the pin denoted by the wrt name.

It is an error if the name of the variable denoted by the identifier appears in any expression. It is also an error if the wrt name is present in the small-signal specification and the name of a variable declared by a partial derivative specification appears in the expression.

**Note**

The category attribute and the report attribute have no meaning in the MAST language.

## Stress Measure Specification

A stress measure specification relates to a variable declared by the stress measure specification and specifies the value and two attributes of that variable.

stress_measure_specification ::=
*[* **control_section** *]* **stress_measure** (
identifier, *category*_identifier,
*report*_expression, *stress*_expression,
simple_name, *rating*_expression
*[* , *reference_rating*_expression *]* ) eos

The stress measure specification declares the identifier as a variable of type NUMBER in the minor declarative region immediately enclosing the specification. If the stress measure specification does not occur immediately within a minor declarative region, then the declaration and the specification together form a minor declarative region that is decorated with the statement attribute of the stress measure specification.

The category identifier defines the category attribute of the variable denoted by the identifier. The report expression defines the report attribute of the variable denoted by the identifier; it must be a globally constant expression of type STRING. The stress expression must be a simple expression of type NUMBER. The simple name must denote an enumeration literal of the predefined type **enum** STRESS_MEASURES. The rating expression and the reference rating expression, if present, must be simple expressions of type NUMBER; they must be globally constant. A stress measure specification without a reference rating expression is equivalent to a stress measure specification whose reference rating expression is the literal 0.

The stress measure specification specifies the value of the variable denoted by the identifier as the value of the expression

$$\frac{100 \times (\text{measure} \times (\text{stress expression}) - \text{reference})}{(\text{derate} \times (\text{rating}) - \text{reference})}$$

where measure is the function associated with the enumeration literal denoted by the simple name, reference is the value of the reference rating expression, rating is the value of the rating expression, and derate is an unspecified function.

It is an error if the name of the variable denoted by the identifier appears in any expression.

**Note**

The category attribute and the report attribute have no meaning in the MAST language.

## Variable Range Specification

A *tolerance range* is a 4-tuple of values of type NUMBER. The values are called minimum, maximum, absolute tolerance, and relative tolerance.

A variable range specification specifies the tolerance range of analog system variables. It relates to each scalar subelement of each analog system variable in a set.

variable_range_specification ::=
*[* **control_section** *]* **range_for_variable** ( *variable*_set,
*min*_expression, *max*_expression
*[* , *abs*_expression *[* , *rel*_expression *] ]* ) eos

Each member of the variable set of a variable range specification must denote an analog system variable. A variable range specification whose variable set is defined by a group is equivalent to a sequence of variable range specifications. For each variable range specification in the equivalent sequence the variable set denotes one group constituent of the canonical group corresponding to the group, and the min expression, max expression, abs expression, and rel expression are the same as the corresponding expressions of the original variable range specification.

The min expression defines the minimum of the tolerance range. The max expression defines the maximum of the tolerance range. The abs expression defines the absolute tolerance of the tolerance range. In the absence of an abs expression the value of the absolute tolerance is ABS*max(abs(minimum), abs(maximum)), where ABS is a scalar value of a numeric type defined by the implementation. The rel expression defines the relative tolerance of the tolerance range. In the absence of a rel expression the relative tolerance is defined by the implementation. All expressions must be globally constant expressions of a numeric type.

It is an error if more than one variable range specification applies to the same analog system variable.

## Unit Range Specification

A unit range specification relates to a unit and specifies the tolerance range of that unit.

unit_range_specification ::=
*[* **control_section** *]* **range_for_unit** ( *unit*_name,
*min*_expression, *max*_expression
*[* , *abs*_expression *[* , *rel*_expression *] ]* ) eos

The unit name must denote a physical unit.

The min expression defines the minimum of the tolerance range. The max expression defines the maximum of the tolerance range. The abs expression defines the absolute tolerance of the tolerance range. In the absence of an abs expression the value of the absolute tolerance is ABS*max(abs(minimum), abs(maximum)), where ABS is a scalar value of a numeric type defined by the implementation. The rel expression defines the relative tolerance of the tolerance range. In the absence of a rel expression the relative tolerance is defined by the implementation. All expressions must be globally constant expressions of a numeric type.

It is an error if more than one unit range specification applies to the same unit in an instance of a template.

**Note**

Different unit range specifications may apply to the same unit in different instances.

## Range Set Specification

A *range set* is a named collection of tolerance ranges.

A range set specification relates to a template and specifies the name of the range set of an instance of that template.

range_set_specification ::=
[ **control_section** ] **range_set** ( expression ) eos

The expression must be a globally constant expression of type STRING. Its value specifies the name of the range set of an instance of a template.

It is an error if more than one range set specification applies to the same instance.

### Notes

Range sets are defined by an implementation.

An implementation may define alternative mechanisms to specify the range set of the root instance.

# 6

# Names

*This chapter describes the different types of MAST names.*

## Names

A name denotes an explicitly declared or implicitly declared entity. Names can also denote subelements of composite objects.

```
name ::=
        simple_name
      | instance_name
      | imported_name
      | selected_name
      | branch_name
      | indexed_name
      | slice_name
      | qualified_name

extended_name ::=
        name
      | decimal_name

prefix ::= name
```

The evaluation of an extended name consists of the determination of the entity denoted by the extended name. It is an error if the extended name is ambiguous after overload resolution.

Certain forms of names include a prefix that is itself a name.

An extended name can be a *constant name* or even a *locally constant name*. Each locally constant name is a constant name.

- Simple names, decimal names, instance names, imported names, and qualified names are locally constant names.
- A selected name is a (locally) constant name if its prefix is a (locally) constant name.
- A branch name is a (locally) constant name if its plus pin aspect and minus pin aspect are both (locally) constant names.
- An indexed name is a (locally) constant name if its prefix is a (locally) constant name, and every index expression is a (locally) constant expression.
- A slice name is a constant name if its prefix is a constant name and the index range of its index constraint is globally constant. A slice name is a locally constant name if its prefix is a locally name and the index range of its index constraint is locally constant.

If the name is a constant name, then its *longest constant prefix* is the name itself. Otherwise, the longest constant prefix of the name is the longest prefix that is a constant name.

## Simple Names

A simple name is the identifier associated with an entity by a declaration.

simple_name ::= identifier

## Decimal Names

A decimal name is the textual form of a decimal literal associated with an entity by a declaration. The textual form includes any leading zeros of the decimal literal.

decimal_name ::= decimal_literal

**Note**

Decimal names are only allowed in the following places:

- As a connection definition in a connection element
- As a connection specification in a connection element

- As the connection formal part of a connection association element
- As the connection actual part of a connection association element

In addition, the decimal name 0 may be part of a pin aspect.

## Instance Names

An instance name denotes an instance of a design entity.

instance_name ::=
prefix . reference_designator

reference_designator ::=
identifier
| special_reference_designator

The evaluation of an instance name consists of the evaluation of the prefix and the determination of the design entity denoted by the prefix.

**Examples**

```
inductor.shunt              gearbox . @"5I317"

@"foo".1e5
```

**Note**

The prefix of an instance name is either a simple name or a qualified name.

## Imported Names

An imported name denotes an object whose declaration appears in an instance of another design entity.

imported_name ::= simple_name ( instance_name )

The evaluation of an imported name consists of the evaluation of the instance name followed by the determination, within the scope of the named instance, of the object denoted by the simple name. It is an error if this object has not been decorated, either explicitly or implicitly, with the export attribute.

### Examples

```
power(r.load)       i(short.sense)      c(c.10)
```

### Note

The instance name part of an imported name may create a forward reference to an instance that is defined at a textually-later place in a template.

## Selected Names

A selected name denotes an entity whose declaration appears within the declaration of a structure type, structure pin type, or union type.

selected_name ::= prefix -> simple_name

The prefix of a selected name must denote an object of a structure type, structure pin type, or union type. The simple name must denote an element of the structure type, structure pin type or union type denoted by the prefix.

The evaluation of a selected name consists of the evaluation of the prefix and the determination of the object denoted by the simple name within the object denoted by the prefix. It is an error if a selected name whose prefix denotes an object of a union type appears in an expression and the selected name does not denote the current alternative.

### Note

A selected name whose prefix is an object of a union type and that does not denote the current alternative may only appear as the target in an assignment statement.

## Branch Names

A branch name denotes a branch between two pins.

branch_name ::=
unit_aspect ( *plus*_pin_aspect
[ branch_name_separator *minus*_pin_aspect *]* )

unit_aspect ::= *unit*_name

pin_aspect ::=
*pin*_name
| ( *pin*_name )
| 0

branch_name_separator ::= ,| ->

The pin name in a pin aspect must not denote an imported object. A pin name that is a selected name must be enclosed in parentheses.

A branch name without a minus pin aspect is equivalent to a branch name whose minus pin aspect denotes the predefined reference pin 0. It is an error if the plus pin aspect and the minus pin aspect of a branch name both denote the predefined reference pin 0.

The pin denoted by the plus pin aspect of a branch name is the *plus pin* of the branch. If the plus pin of the branch is of a composite pin type, then the plus pin of each scalar subelement of the branch is the corresponding scalar subelement of the plus pin of the branch. Similarly, the pin denoted by the minus pin aspect of a branch name is the *minus pin* of the branch. If the minus pin of the branch is of a composite pin type, then the minus pin of each scalar subelement of the branch is the corresponding scalar subelement of the minus pin of the branch. For each scalar subelement of a branch, the plus pin and minus pin of the scalar subelement must either be of the same pin type, or one must be of the predefined pin type NEUTRAL.

The type of a branch is determined as follows. If the plus pin of the branch is of a composite pin type, or if both pins of the branch are of a scalar pin type, then the type of the branch is the branch type implied by the pin type of the plus pin of the branch. Otherwise, the type of the branch is the branch type implied by the pin type of the minus pin of the branch.

The unit of each scalar subelement of a branch is determined as follows.

- If the unit aspect of the branch name denotes the predefined unit ACROSS, and if the pin type of the plus pin of the scalar subelement is any pin type other than the predefined pin type NEUTRAL, then the unit of the scalar subelement is the across unit of the pin type of the plus pin of the branch.
- If the unit aspect of the branch name denotes the predefined unit ACROSS, and if the pin type of the plus pin of the scalar subelement is the predefined pin type NEUTRAL, then the unit of the scalar subelement is the across unit of the pin type of the minus pin of the branch.
- If the unit aspect of the branch name denotes the predefined unit THROUGH, and if the pin type of the plus pin of the scalar subelement is any pin type other than the predefined pin type NEUTRAL, then the unit of the scalar subelement is the through unit of the pin type of the plus pin of the branch.
- If the unit aspect of the branch name denotes the predefined unit THROUGH, and if the pin type of the plus pin of the scalar subelement is the predefined pin type NEUTRAL, then the unit of the scalar subelement is the through unit of the pin type of the minus pin of the branch.

If all scalar subelements of a branch have the same unit, then the name of this unit may be used instead of the predefined units ACROSS or THROUGH as the unit aspect of the branch name.

A branch is an *across branch* if the unit of each scalar subelement is the across unit of the pin type of the plus pin or minus pin of that scalar subelement. Similarly, a branch is a *through branch* if the unit of each of its scalar subelements is the through unit of the pin type of the plus pin or minus pin of that scalar subelement.

The evaluation of a branch name consists of the evaluation of its plus pin aspect and minus pin aspect, followed by the evaluation of its unit aspect.

## Examples

```
unit { "V", "Volt", "voltage" } v
unit { "A", "Ampere", "current" } i
unit { "K", "Kelvin", "temperature" } tk
unit { "W", "Watt", "power" } p
pin electrical across v through i
pin thermal across tk through p
electrical p1, p2
struc { electrical p1, p2; } p3
struc { electrical e; thermal t; } p4, p5

ACROSS(p1, p2)          # an across branch with plus pin p1
                        # and minus pin p2

i(p2->p1)               # a through branch with plus pin
                        # p2 and minus pin p1

v(p1)                   # an across branch with p1 as plus pin
                        # and the reference pin 0 as minus pin

v(0, p1)                # an across branch whose value is the
                        # negative of that in the previous
                        # example

THROUGH((p3->p1))       # a through branch between subelement
                        # p1 of pin p3 and the reference pin 0

THROUGH(p3->p1)         # a through branch between composite
                        # pin p3 and scalar pin p1

ACROSS(p4, p5)          # an across branch between composite
                        # pins p4 and p5. The units of the
                        # scalar subelements of the branch are
                        # v and tk
```

## Notes

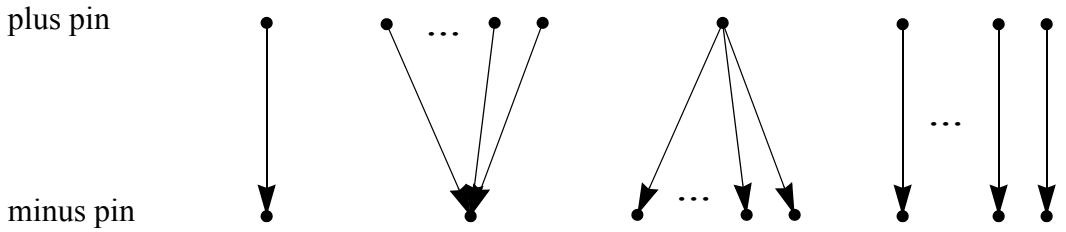
1. With the exception of the predefined reference pin 0, decimal literals used to denote pins cannot appear in branch names.
2. By convention, the comma (,) is used as the branch name separator in an across branch name, while the right arrow (->) is used in through branch names.

### References

Implicit Declaration of Imported Objects

### Commentary

The essence of these definitions is that a branch denoted by a branch name takes its shape from its pins, as shown in the following figure. The unit of each scalar subelement of the branch is the across or through unit of its plus pin or, if the pin type of the plus pin is NEUTRAL, of its minus pin.




Decimal literals cannot appear in a pin aspect of a branch name because unit names and function names are in different name spaces. This makes it impossible to determine whether v(1) is a function call or a branch name (for example, the voltage between electrical pin 1 and the reference pin 0). Therefore, a decimal literal that denotes a pin can appear only as a connection definition or a connection specification in a template connection element, or as the connection actual part or the connection formal part in a connection association element of an instantiation statement.

## Indexed Names

An indexed name denotes an element of an array.

indexed_name ::= prefix "[" expression *{* , expression *}* "]"

The prefix of an indexed name must denote an object of an array type. The expressions specify index values; they must be simple expressions. There must be one expression for each index position of the array object denoted by the prefix, and the type of the expression must be type compatible with the type of the index constraint of the corresponding index position of the prefix.

The evaluation of an indexed name consists of the evaluation of its prefix and the index expressions. It is an error if the value of any index expression is

smaller than the lower bound or larger than the upper bound of the corresponding index position of the array object denoted by the prefix.

## Slice Names

A slice name denotes a substring or a one-dimensional array consisting of consecutive elements of another one-dimensional array.

slice_name ::= prefix "[" index_constraint "]"

The prefix of a slice name must denote an object of the predefined type STRING or of a one-dimensional array type. The object class of the slice is the same as the object class of the prefix of the slice name. Similarly, if the prefix of the slice name is of an array type then the element type of the slice is the element type of the array type.

The index constraint of a slice name must include both the lower bound and the upper bound. If the prefix is of an array type, then the type of the index constraint must be type compatible with the type of the index of the prefix. Otherwise, the type of the index constraint must be the predefined type INTEGER. The slice is a null slice if the value of the lower bound exceeds the value of the upper bound.

The evaluation of a slice name consists of the evaluation of the prefix and the index constraint. It is an error if the value of the prefix is **undef**. Unless the name denotes a null slice, it is also an error if the value of the lower bound is smaller than the lower bound of the string object or array object denoted by the prefix, or if the value of the upper bound is larger than the upper bound of the string object or array object denoted by the prefix.

### Examples

```
number gain[10]
string s = "abc"

gain[3:5]# a slice of length 3

gain[6:5]# the null slice

s[3:*] # a slice of length 1 with value "c"
```

### Note

The declaration of the array object is implicit if the prefix is itself a slice name.

## Qualified Names

A qualified name denotes an entity declared in a template or contained in a design library.

qualified_name ::= prefix .. suffix

suffix ::= simple_name

If the prefix of a qualified name denotes a template, then the suffix must be a simple name that is either the external name of a formal connection of the template or that appears in the argument list of the template header definition of that template; the qualified name is an alias for the base name of the type of the connection or argument denoted by the suffix. It is an error if the template denoted by the prefix has not been analyzed.

If the prefix of a qualified name denotes a design library, then the suffix must denote a design unit contained in that design library. It is an error if the prefix of a qualified name is itself a qualified name and the prefix of that qualified name does not denote a design library.

A qualified name is said to be a *fully qualified name* if the simple name preceding the first delimiter of the qualified name denotes a design library.

The evaluation of a qualified name consists of the evaluation of its prefix and its suffix.

**Examples**

```
q..model                # if q is the name of a template,
                        # the type of its argument model

work..base              # the fully qualified name of
                        # entity base declared in library
                        # work
```

**Note**

The tag in a type reference may be a fully qualified name.

# 7

# Expressions

*An expression is a formula that defines the computation of one or more values. This section defines the syntax, the order of evaluation, and the meaning of expressions.*

## Expressions

The order of evaluation and the precedence of operators is defined by the following grammar.

expression ::=
    logical_or_expression

logical_or_expression ::=
    logical_and_expression *{* "|" logical_and_expression *}*

logical_and_expression ::=
    equality_expression *{* & equality_expression *}*

equality_expression ::=
    relational_expression *{* equality_operator relational_expression *}*

relational_expression ::=
    additive_expression *{* relational_operator additive_expression *}*

additive_expression ::=
    term *{* additive_operator term *}*

term ::=
    factor *{* multiplicative_operator factor *}*

factor ::=
    *[* unary_operator *]* primary *{* ** *[* unary_operator *]* primary *}*

An operator is said to be a *binary operator* if it has two operands. The *left operand* precedes the binary operator, and the *right operand* follows the binary operator.

In a sequence of binary operators with the same precedence level the operators associate with their operands as follows:

- The ** operator (exponentiation) associates from right to left.
- All other binary operators associate from left to right.

An expression is said to be a *simple expression* if the cardinality of its profile is one. Only simple expressions can be the operands of an operator.

The type of a simple expression depends only on the operators and the operands appearing in the simple expression. The corresponding rules are described in Operators and Primaries. Type conversions are described in Type Conversions. Constant expressions are described in Constant Expressions.

### Notes

An expression that is not a simple expression can appear only in an assignment statement and as an actual argument of a function call calling a foreign function, where the expression must match a sequence of arguments in the unspecified portion of the argument profile.

It is a consequence of the precedence and associativity rules that the following pairs of expressions are equivalent:

| | |
|---|---|
| a+b-c*d/e+f | `(((a+b)-((c*d)/e))+f)` |
| `-a--b` | `((-a)-(-b))` |
| `-a**-b**-c` | `(-(a**(-(b**(-c)))))` |

### Commentary

The expression grammar completely defines the precedence of operators in the MAST language. The following table presents a summary of this information in order from the highest to the lowest precedence. It includes the operator symbols that will be described in the following sections.

| | |
|---|---|
| exponentiation and unary operators | ** + - ~ |
| multiplicative operators | * / % |
| additive operators | + - // |

| | |
|---|---|
| relational operators | < > <= >= |
| equality operators | == ~= |
| logical and operator | & |
| logical or operator | \| |

# Operators

## Logical Operators

The logical operators are defined for operands of any numeric type. The type of the result returned by a logical operator is the predefined type INTEGER. The result is defined by the following table.

| **value of left operand** | **value of right operand** | **result of logical or (\|)** | **result of logical and (&)** |
|---|---|---|---|
| zero | zero | 0 | 0 |
| zero | nonzero | 1 | 0 |
| nonzero | zero | 1 | 0 |
| nonzero | nonzero | 1 | 1 |

The result is undefined if the value of any operand equals the value of the literal undef.

For the determination of the result of a logical operator the left operand of the logical operator is evaluated first. The right operand is evaluated only if the value of the left operand does not completely specify the result of the logical operator.

### Examples

```
number a=1string
s="abc"

a==0 | s=="abc"     # logical expression with value 1.
                    # Right operand is evaluated because
                    # value of left operand is zero

a==0 & s=="abc"     # logical expression with value 0.
                    # Right operand is not evaluated

a | s               # illegal: right operand is not of
                    # a numeric type
```

### Rationale

The result is undefined if the value of an operand is **undef** for efficiency reasons. Otherwise, each operation would require testing each operand for this value.

---

## Equality Operators

equality_operator ::= == | ~=

The equality operators are defined for operands of any type. The operands of an equality operator must have compatible types, and their expected type (see Overload Resolution) is the supertype implied by the types of the two operands. The type of the result returned by an equality operator is the predefined type INTEGER.

The == operator (equal to) returns 1 if its left operand is equal to its right operand; it returns 0 otherwise. The ~= operator (not equal to) returns 0 if its left operand is equal to its right operand; it returns 1 otherwise.

Two operands A and B are equal if:

- The operands are of a scalar type and have the same value.
- The operands are of an array type, and for each element of operand A there is a matching element of operand B and vice-versa, and the two matching elements are equal.
- The operands are of a structure type, and each element of operand A and the matching element of operand B are equal.

- The operands are of a union type, the same alternative is the current alternative of both operands, and the current alternatives are equal.
- Both operands are the literal **undef**.

For the determination of the result of an equality operator the left and the right operand of the equality operator are evaluated, in this order. It is an error if the types of the operands cannot be determined.

### Examples

```
struc st1 { number a,b; } s1=(1,2),s2=(2,3)
struc st2 { number a,b,c; } s3=(1,2,3)

s1==s2              # Equality expression with value 0

s1~=s3              # illegal: left and right operand
                    # have different types

(1,2)==(1,2)        # illegal: type of operands cannot
                    # be determined
```

### Notes

- A group whose canonical profile has a cardinality greater than 1 cannot be an operand of an equality operator because it does not have a type.
- The type of the operands cannot be determined if both operands are structure aggregates or union aggregates.

## Relational Operators

relational_operator ::= < | > | <= | >=

The relational operators are defined for operands of any ordered type and of any one-dimensional array type whose elements are of an ordered type. If both operands of a relational operator are of a one-dimensional array type, then the element types of the two operands must be type compatible; the expected type of each element is the supertype implied by the two element types. Otherwise, the two operands must have compatible types, and their expected type is the supertype implied by the types of the two operands. The type of the result returned by a relational operator is the predefined type INTEGER.

The > operator (greater than) returns 1 if its left operand is greater than its right operand; it returns 0 otherwise. For two operands A and B, operand A is greater than operand B if:

- The operands are of a numeric type and the value of the expression A - B is positive.
- The operands are of an enumeration type and the enumeration value of operand A is greater than the enumeration value of operand B.
- The operands are of the predefined type STRING, the slices A[1:i] and B[1:i] are equal for some integer value i, and either the slice A[i+1:i+1] follows the slice B[i+1:i+1] in the ISO 8859-1 collation order, or the length of operand B is i and the length of operand A is greater than i.
- The operands are of a one-dimensional array type, a slice SA of some length i whose prefix is A and whose lower bound is the lower bound of A is equal to a slice SB of length i whose prefix is B and whose lower bound is the lower bound of B, and either the element of A immediately following SA is greater than the element of B immediately following SB, or the length of B is i and the length of A is greater than i.

The >= operator (greater than or equal to) returns 1 if either the > operator or the == operator for the same two operands returns 1; it returns 0 otherwise. The < operator (smaller than) returns 1 if the >= operator for the same two operands returns 0; it returns 0 otherwise. The <= operator (smaller than or equal to) returns 1 if the > operator for the same two operands returns 0; it returns 0 otherwise.

For the determination of the result of a relational operator the left and the right operand of the relational operator are evaluated, in this order. The result is undefined if the value of any operand, or subelement thereof, equals the value of the literal undef.

### Examples

```
number a=4,b=3,c[2]=[1,2],d[2]=[3.5,2]

a < b               # Relational expression with value 0

c <= d              # Relational expression with value 1

a > c               # illegal: left and right operands
                    # have different types
```

### Notes

1. The meaning of the expression a < b < c is (a < b) < c rather than (a < b) & (b < c).
2. While the language does support comparing **undef** values for equality or inequality, it does not define the result of a relational operator if one of the operands has the value undef. Therefore, **undef** == **undef** returns 1, but the result of **undef** >= **undef** is undefined.

### References

Literals

### Rationale

The result is undefined if the value of an operand is **undef** for efficiency reasons. Otherwise, the evaluation of each operator would require testing both operands for this value.

---

## Additive Operators

additive_operator ::= + | - | //

The + operator (plus) and - operator (minus) are defined for any numeric type and for composite types whose scalar subelements are of a numeric type; the two types must be type compatible and for each scalar subelement of the left operand there must be a matching scalar subelement of the right operand. The expected type of the operands is the supertype implied by the types of the two operands, and the type of the result returned by a + operator or - operator is the expected type of its operands. The // operator (concatenation) is defined for the predefined type STRING and for all one-dimensional array types. If the types of both operands are one-dimensional array types, then the element types of the two operands must be type compatible; the expected type of each element is

the supertype implied by the two element types. Otherwise, the types of the operands must be type compatible.

The + operator and the - operator have their conventional mathematical meaning. If their operands are of a composite type, then the operator is applied to each pair of matching scalar subelements as operands. The result of the // operator whose operands are of type STRING is a value consisting of the value of the left operand followed by the value of the right operand. The result of the / / operator whose operands are of an array type is an array value whose elements are the elements of the left operand of the // operator followed by the elements of the right operand. The index constraint of the result is normalized, and its upper bound is the sum of the lengths of the two operands.

For the determination of the result of an additive operator the left and the right operand of the additive operator are evaluated, in this order. The result is undefined if its value is outside the range defined for its type, or if the value of any scalar operand of a + operator or a - operator equals the value of one of the literals **inf** or **undef**. It is an error if the value of any operand of a // operator equals the value of the literal **undef**.

**Example**

```
string s1="abc", s2="cbadef", s3

s3 = s1 // s2[4:*]  # result is the string "abcdef"
```

## Multiplicative Operators

multiplicative_operator ::= * | / | %

The multiplicative operators are defined for any numeric type. The * operator (multiplication) is also defined for one operand of a numeric type and the other operand of a composite type with scalar subelements of a numeric type. The / operator (division) and the % operator (modulus) are also defined for a left operand of a composite type with scalar subelements of a numeric type and a right operand of a numeric type. The expected type of the operands of a multiplicative operator is the supertype implied by the types of the two operands. The type of the result returned by a multiplicative operator is the expected type of its operands.

The * operator and the / operator returning type NUMBER have their conventional mathematical meaning. The result of the / operator returning type INTEGER with a left operand A and a right operand B is the integer i such that

the expression A - i * B has the same sign as A and its magnitude is less than the magnitude of B. The result of the % operator with a left operand A and a right operand B is the value A - i * B for some integer i such that the result has the same sign as A and its magnitude is less than the magnitude of B. In all cases, if one operand is of a composite type, then the operator is applied to each scalar subelement of that operand.

For the determination of the result of a multiplicative operator with scalar operands the left and the right operand of the multiplicative operator are evaluated, in this order. The result is undefined if its value is outside the range defined for its type, or if the value of any operand equals the value of one of the literals **inf** or **undef**. It is an error if the value of the right operand of a / operator or a % operator equals zero.

**Note**

The definition of the % operator is equivalent to the definition of the fmod function in the C language and to the definition of the remainder in VHDL.

## Unary Operators

unary_operator ::= + | - | ~

The unary operators are defined for any numeric type. The unary + operator (unary plus) and the unary - operator (unary minus) are also defined for any operand of a composite type with scalar subelements of a numeric type. The type of the result returned by the unary + operator or the unary - operator is the type of its operand. The type of the result of the ~ operator (not) is the predefined type INTEGER.

The unary + operator returns the value of its operand. The unary - operator returns the negative of the value of its operand. If the operand is of a composite type, then the unary - operator is applied to each scalar subelement. The ~ operator returns 1 if the value of its operand is zero; it returns 0 otherwise.

For the determination of the result of a unary operator the operand of the unary operator is evaluated. The result is undefined if the value of the operand of a unary - operator or of a ~ operator equals the value of the literal undef.

## Exponentiation Operator

The ** operator (exponentiation) is defined for any numeric type and for a left operand of a composite type with scalar subelements of a numeric type and a

right operand of a numeric type. The expected type of the operands of the exponentiation operator is the supertype implied by the types of the two operands. The type of the result returned by the exponentiation operator is the expected type of its operands.

The exponentiation operator raises its left operand to the power specified by its right operand. If the left operand is of a composite type, then the exponentiation operator is applied to each scalar subelement.

For the determination of the result of an exponentiation operator with scalar operands the right and left operand of the exponentiation operator are evaluated, in this order. The result is undefined if its value is outside the range defined for its type, or if the value of any operand equals the value of one of the literals **inf** or **undef**. It is an error if the value of the right operand is negative and the value of the left operand is zero. It is also an error if the value of the right operand is negative and the return type of the operator is INTEGER. Finally, it is an error if the value of the left operand is negative and the value of the right operand is not an integral number.

## Primaries

A primary is a basic building block of an expression.

```
primary ::=
    name
  | literal
  | function_call
  | aggregate
  | structure_overlay
  | conditional_expression
  | ( expression )
```

Each primary has one or more types and a corresponding number of values. The profile of a primary denoted by a name is the profile of the entity associated with the name. The profile of an expression enclosed in parentheses is the profile of the expression.

**Notes**

1. It is a consequence of these rules that the name of a pin is not a primary.
2. A primary may have more than one type if it is a name that denotes a group, a function call, or a group name or function call enclosed in parentheses.

## Literals

literal ::=
  numeric_literal
| string_literal
| enumeration_literal
| **inf**
| **undef**

Numeric literals include integer literals and real literals; they are described in Numeric Literals. String literals are described in String Literals. Enumeration literals are literals of an enumeration type; they are described in Enumeration Types.

The literal **inf** represents a value of the predefined type NUMBER with the following properties:

- **inf** is greater than any other value of type NUMBER
- any other value of type NUMBER is greater than **-inf**

The literal **undef** is type compatible with any scalar type and any union type, and its value is different from any other value representable by the type.

The evaluation of a literal yields the corresponding value.

**Note**

On a computer architecture that supports IEEE Std. 754 or IEEE Std. 854 for floating point number representation, a possible implementation of the literal inf is the value Infinity.

## Function Calls

function_call ::=
name ( *[ function*_argument_association_list *]* )

A function call invokes the function denoted by the name and specifies the actual arguments, if any, to be associated with the formal arguments of the function. The evaluation of the function call yields a result whose profile is defined by the corresponding function declaration. If the profile is unspecified or contains an unspecified portion, then it is determined as part of the evaluation of the function call. It is an error if the result profile of a function call cannot be determined.

For the evaluation of a function call, each association element of the function argument association list is first evaluated in the order in which the elements appear in the function argument association list. Then, the initial value expression of each remaining formal argument, defined in the function declaration, is evaluated and associated with the formal argument in the order of the corresponding declarations. Finally, if the function is an MAST function, then its body is executed. Otherwise, the foreign function is called.

**Notes**

1. Only foreign functions can have an unspecified profile or a profile with an unspecified portion.
2. A function with an unspecified result profile can only be called in an assignment statement whose expression is the function call.

**References**

Argument Association Lists, Foreign Functions Called from a Template or a MAST Function

## Aggregates

An aggregate combines one or more values into a composite value. The aggregate may be of an array type, a structure type, or a union type.

```
aggregate ::=
    array_aggregate
  | structure_aggregate
  | union_aggregate
```

The type of the aggregate must be determinable from the context in which the aggregate appears. That is, an aggregate may only appear:

- As the initial value expression in an object declaration
- As the expression in an assignment statement whose target is a declared object
- As the expression in a contribution statement
- As an expression in a labeled equation statement or a make statement, if and only if the type of the other expression is defined
- As the actual expression in an argument association element, if and only if the formal is a declared object
- As the expression in an aggregate association element

- As a subaggregate (see Array Aggregates) of an array aggregate
- As an operand of an equality or inequality operator, if and only if the type of the other operand is defined

**Note**

The formal in an argument association element is not a declared object if the argument association element corresponds to an argument in the unspecified portion of the argument profile.

**References**

Object Declarations, Argument Association Lists, Assignment Statement, Equation Statements

## Array Aggregates

An array aggregate defines a value of an array type.

array_aggregate ::=
"[" expression_list "]"

expression_list ::=
expression *{* , expression *}*

Each expression in the expression list of an array aggregate defines one element of the array value; it must be a simple expression.

The type of a one-dimensional array aggregate is determined as follows. The element type is the supertype implied by the types of the expressions in the expression list. The index range is normalized, it has an upper bound that is equal to the cardinality of the expression list, and its type is INTEGER.

For an n-dimensional array aggregate each expression in the expression list is a subaggregate that is an (n-1)-dimensional array or array aggregate. It is an error if the types of the subaggregates of an n-dimensional array aggregate are not type compatible.

The type of an n-dimensional array aggregate is determined as follows. The element type is the supertype implied by the element types of its subaggregates. The index range at the first position has a lower bound of 1 and an upper bound that is equal to the number of subaggregates; its type is INTEGER. The index ranges at positions 2 through n of the n-dimensional array aggregate are the normalized index ranges at positions 1 through n-1 of the first subaggregate.

For the evaluation of an array aggregate the expression list is evaluated, then each value that is not a subaggregate is converted to the element type of the array aggregate. For the evaluation of an expression list the expressions are evaluated in the order in which they appear in the expression list.

**Examples**

```
number a=4
enum {y, n} yn=y

[1,2.5,a+3]          # one-dimensional array aggregate
                     # whose element type is number and
                     # whose index range is 1 through 3

[[1,2,3],[4,5,6]]    # two-dimensional array aggregate
                     # whose element type is integer. Its
                     # first index range is 1 through 2,
                     # its second is 1 through 3

[2,yn,5]             # illegal: element types are not
                     # compatible

[[1,2,3],[4,5]]      # illegal: subaggregates have different
                     # types
```

## Structure Aggregates

A structure aggregate defines a value of a structure type.

structure_aggregate ::=
( *[* aggregate_association_list *]* )

aggregate_association_list ::=
aggregate_association_element *{* , aggregate_association_element *}*

aggregate_association_element ::=
*[* simple_name = *]* expression

Each aggregate association element in the aggregate association list of a structure aggregate associates an expression with an element of the structure value. The expression in an aggregate association element must be a simple expression.

The type of a structure aggregate is determined by considering the syntax rules and any rule that requires the structure aggregate to be of a certain type or type compatible with the type of an entity or an expression.

An aggregate association element is said to be *named* if the element is specified explicitly by its simple name; it is said to be *positional* otherwise. For a positional association the element of the value is implicitly specified by the textual position of the aggregate association element in the structure aggregate.

Named association elements may appear in any order, but if named and positional association elements appear in the same aggregate association list, then any named association element must follow all positional association elements. An aggregate association list containing only a single aggregate association element must be specified using a named association element.

The type of the expression in an aggregate association element must be type compatible with the type of the corresponding element of the structure type. If a subelement of the structure type is of an unconstrained array type, then the upper bound of each assumed index range of the subelement is defined by the association such that the length of the index range is equal to the length of the index range of the corresponding subelement of the expression at the same index position.

Each element of the value defined by the structure aggregate must be specified at most once in the aggregate association list of the structure aggregate. If a particular element is not specified in the aggregate association list of the structure aggregate, then its value is the default value of the corresponding element in the structure type definition of the type of the structure aggregate.

For the evaluation of a structure aggregate the aggregate association list is evaluated. For the evaluation of an aggregate association list the aggregate association elements are evaluated in the order in which they appear in the aggregate association list. For the evaluation of an aggregate association element the name, if any, and the expression are evaluated.

### Examples

```
struc s1 { number a,b=4,c=4; } p1, p2, p4
struc s2 { number d[2],e; } p3, p5

p1 = (1,2,3)          # the element values are:
                      # a=1, b=2, c=3

p2 = (3,c=3)          # the element values are:
                      # a=3, b=4, c=3

p3 = (d=[3,4])        # the element values are:
                      # d=[3,4], e=undef

p4 = (1,2,b=3)        # illegal: b specified more than
                      # once

p5 = (e=4,[5,2])      # illegal: positional association
                      # follows named association

([1,2])               # not a structure aggregate because
                      # a single element must be
                      # specified by named association
```

### Note

The rules to determine the type of a structure aggregate are similar to the overload resolution rules of Overload Resolution.

## Union Aggregates

A union aggregate defines a value of a union type.

union_aggregate ::=
( aggregate_association_element )

The type of a union aggregate is determined by considering the syntax rules and any rule that requires the union aggregate to be of a certain type or type compatible with the type of an entity or an expression.

The single aggregate association element of a union aggregate must be specified using a named association. The simple name of the aggregate association element must denote the name of an alternative of the union type, and the type of the expression must be type compatible with the type of this alternative.

For the evaluation of a union aggregate the aggregate association element is evaluated.

### Examples

```
union u1 { number a; string s; } p

p = (a=3)           # the union aggregate specifies
                    # alternative a with value 3

p = (s="abc")       # the union aggregate specifies
                    # alternative s with value "abc"

p = (d=5)           # illegal because union u1 has no
                    # alternative d
```

### Note

The rules to determine the type of a union aggregate are similar to the overload resolution rules of Overload Resolution.

## Structure Overlays

A structure overlay defines a value of a structure type.

structure_overlay ::=
prefix <- ( *[* aggregate_association_list *]* )

The prefix of a structure overlay must denote an object of a structure type; its type is the type of the structure overlay. Each aggregate association element in the aggregate association list of a structure overlay associates an expression with an element of the structure value.

Each element of the value defined by the structure aggregate must be specified at most once in the aggregate association list of the structure overlay. If a particular element is not specified in the aggregate association list of the structure overlay, then its value is the value of the corresponding element of the prefix of the structure overlay.

For the evaluation of a structure overlay the prefix of the structure overlay and its aggregate association list are evaluated.

### Examples

The following structure overlays must be considered in the context of the examples of Structure Aggregates.

```
p1<-(4)              # the element values are:
                     # a=4, b=2, c=3

p2<-(c=-1)           # the element values are:
                     # a=3, b=4, c=-1
```

## Conditional Expressions

A conditional expression defines a value that is dependent on a condition.

conditional_expression ::=
**if** condition **then** expression **else** expression

condition ::=
expression

The third expression matches the longest text that describes a syntactically correct expression.

The condition must be a simple expression of a numeric type. The second and the third operand in a conditional expression must be simple expressions of compatible types; their expected type is the supertype implied by their types. The type of a conditional expression is the expected type of its second operand.

For the evaluation of a conditional expression the condition is evaluated first. If the value of the condition is nonzero, then the second operand is evaluated and its value becomes the value of the conditional expression; otherwise, the third operand is evaluated and its value becomes the value of the conditional expression.

# Type Conversions

Type conversions are necessary if the type of a simple expression differs from the expected type of the expression. Automatic type conversion is defined between compatible types.

The conversion of a value that corresponds to the value of the literal **undef** consists of setting it to the value of the literal **undef** in the expected type.

The conversion of a value from the predefined type NUMBER to the predefined type INTEGER consists of setting the result to the integral part of the value. The behavior is undefined if the integral part of the value is outside the range defined for type INTEGER.

The conversion of a value from the predefined type INTEGER to the predefined type NUMBER consists of setting the integral part of the result to the value and the fractional part to zero.

The conversion of a value of an array type to another array type consists of the conversion of each index range and the conversion of each element of the value. For the conversion of an index range F to an index range T, if the upper bound of T is not assumed then F is replaced by T; otherwise, F is replaced by an index range whose lower bound is the lower bound of T and whose upper bound is determined such that the length of T is equal to the length of F.

## Constant Expressions

Certain expressions are said to be *constant*. There are three kinds of constant expressions. An expression is *locally constant* if it can be evaluated during the compilation of a design unit. An expression is *argument constant* if it can be evaluated after the elaboration of the arguments of an instance. An expression is *globally constant* if it can be evaluated as part of the elaboration of the design unit in which it appears.

### Locally Constant Expressions

An expression is locally constant if and only if each primary appearing in the expression is locally constant. A locally constant primary is one of the following:

1. A literal of any type
2. A simple name that denotes an object decorated with the const attribute
3. A subelement of a locally constant primary, if and only if any index expressions are locally constant expressions
4. A slice of a locally constant primary, if and only if its index constraint is locally constant (see below)

5. A conditional expression, if and only if each of its three operands is a locally constant expression
6. A locally constant expression enclosed in parentheses

A locally constant index range is either an index range specified by a type mark or an index range whose lower bound, if present, and upper bound are locally constant expressions. A locally constant index constraint is an index constraint whose index range is locally constant. A locally constant type is a scalar type other than type STRING, an array type whose element type is a locally constant type and whose index constraints are all locally constant, or a structure type or union type whose elements are of a locally constant type. A locally constant pin type is a scalar pin type, an array pin type whose element pin type is a locally constant pin type and whose index constraints are all locally constant, or a structure pin type whose elements are of a locally constant pin type.

Similarly, a relaxed locally constant index range is an index range that is either locally constant or that has an upper bound that is assumed. A relaxed locally constant index constraint is an index constraint whose index range is relaxed locally constant. A relaxed locally constant type is a scalar type, an array type whose element type is a relaxed locally constant type and whose index constraints are all relaxed locally constant, or a structure type or union type whose elements are of a relaxed locally constant type. A relaxed locally constant pin type is a scalar pin type, an array pin type whose element pin type is a relaxed locally constant pin type and whose index constraints are all relaxed locally constant, or a structure pin type whose elements are of a relaxed locally constant pin type.

## Argument Constant Expressions

An expression is argument constant if and only if each primary appearing in the expression is argument constant. An argument constant primary is one of the following:

1. A locally constant primary
2. A simple name that denotes a parameter declared in the header of a template
3. A simple name that denotes a parameter declared in the body of a template, if and only if it is neither the target of an assignment statement nor the actual argument of a function call calling the ADDR function in a statement decorated with the **parameters** attribute

4. A subelement of an argument constant primary, if and only if it is not the target of an assignment statement and any index expressions are argument constant expressions
5. A slice of an argument constant primary, if and only if it is not the target of an assignment statement and its index constraint is argument constant (see below)
6. An array aggregate, if and only if each expression in its expression list is an argument constant expression
7. A structure aggregate, if and only if the expression in each aggregate association element of its aggregate association list is an argument constant expression
8. A union aggregate, if and only if the expression in its aggregate association element is an argument constant expression
9. A function call whose function name denotes a pure function and whose actual arguments are argument constant expressions
10. An argument constant expression enclosed in parentheses

An argument constant index range is either an index range specified by a type mark or an index range whose lower bound, if present, and upper bound are argument constant expressions. An argument constant index constraint is an index constraint whose index range is argument constant. An argument constant type is a scalar type other than type STRING, an array type whose element type is an argument constant type and whose index constraints are all argument constant, or a structure type or union type whose elements are of an argument constant type. An argument constant pin type is a scalar pin type, an array pin type whose element pin type is an argument constant pin type and whose index constraints are all argument constant, or a structure pin type whose elements are of an argument constant pin type.

Similarly, a relaxed argument constant index range is an index range that is either argument constant or that has an upper bound that is assumed. A relaxed argument constant index constraint is an index constraint whose index range is relaxed argument constant. A relaxed argument constant type is a scalar type, an array type whose element type is a relaxed argument constant type and whose index constraints are all relaxed argument constant, or a structure type or union type whose elements are of a relaxed argument constant type. A relaxed argument constant pin type is a scalar pin type, an array pin type whose element pin type is a relaxed argument constant pin type and whose index constraints are all relaxed argument constant, or a structure pin type whose elements are of a relaxed argument constant pin type.

## Globally Constant Expressions

An expression is globally constant if and only if each primary appearing in the expression is globally constant. A globally constant primary is one of the following:

1. An argument constant primary
2. A simple name that denotes a parameter declared in the body of a template
3. A subelement of a globally constant primary, if and only if any index expressions are globally constant expressions
4. A slice of a globally constant primary, if and only if its index constraint is globally constant (see below).
5. An array aggregate, if and only if each expression in its expression list is a globally constant expression
6. A structure aggregate, if and only if the expression in each aggregate association element of its aggregate association list is a globally constant expression
7. A union aggregate, if and only if the expression in its aggregate association element is a globally constant expression
8. A function call whose function name denotes a pure function and whose actual arguments are globally constant expressions
9. The simulator variables STATISTICAL and WORST_CASE
10. A globally constant expression enclosed in parentheses

A globally constant index range is either an index range specified by a type mark or an index range whose lower bound, if present, is a globally constant expressions and whose upper bound is either assumed or a globally constant expression. A globally constant index constraint is an index constraint whose index range is globally constant. A globally constant type is a scalar type, an array type whose element type is a globally constant type and whose index constraints are all globally constant, or a structure type or union type whose elements are of a globally constant type. A globally constant pin type is a scalar pin type, an array pin type whose element pin type is a globally constant pin type and whose index constraints are all globally constant, or a structure pin type whose elements are of a globally constant pin type.

An expression is relaxed globally constant if and only if each primary appearing in the expression is relaxed globally constant. A relaxed globally constant

primary is a globally constant primary or a name that denotes an imported object of class parameter.

# 8

# Statements

*This chapter describes the three classes of statements.*

statement ::=
  executable_statement
| nonexecutable_statement
| generic_statement

statement_attribute ::=
  **parameters**
| **equations**
| **values**
| **states**
| **control_section**

There are three classes of statements. Executable statements define the algorithm in a function or the behavior of a design entity. Nonexecutable statements define the structure of a design entity. Generic statements can be used to define the algorithm in a function, the behavior or the structure of a design entity, or specifications in a design entity.

Some statements may include the specification of a statement attribute. Such attributes are said to *decorate* the statement. Additionally, each statement that is guarded (see Loop Statement, When Statement, and Conditional Statement) is implicitly decorated with the statement attributes of its guard. Similarly, each statement that is a sentence in a compound statement is implicitly decorated with the statement attributes of the compound statement. Implicit decorations as described in this paragraph are said to be *inherited*.

It is an error if a function statement or a nonexecutable statement is decorated with any statement attribute. It is also an error if an executable statement or a generic statement in a template body is decorated with more than one statement attribute.

**Note**

Some statements are implicitly decorated with statement attributes in additional ways.

## Executable Statements

Executable statements define the algorithm in a function or the behavior of a design entity.

executable_statement ::=
  assignment_statement
| loop_statement
| exit_statement
| next_statement
| return_statement
| when_statement
| equation_statement

Statements with the same statement attributes (including no statement attributes) execute in the order in which they appear. It is an error if an executable statement is decorated with the **control_section** attribute.

### Assignment Statement

An assignment statement either replaces the value of zero or more objects with new values specified by an expression or defines characteristic expressions (see Equation Statements).

assignment_statement ::=
*[* statement_attribute *] [* target = *]* expression eos

target ::=
  name
| inline_group

An assignment statement is said to be a *simple assignment statement* if the expression is a simple expression. The target, if present, of a simple assignment statement must be a name that denotes an object. The object must not be an imported object or a formal argument of a function. If the name has not been declared, and if the simple assignment statement has neither an explicit nor an inherited decoration, then the name must be a simple name, and

the simple assignment statement implicitly declares the simple name, as follows.

- If the expression is a globally constant expression, then the assignment statement declares an object of class parameter whose name is the simple name and whose type is the type of the expression.
- If the simple name is the external name or the internal name of a formal connection of the design entity in which the assignment statement appears, then the assignment statement declares an object of class analog system variable whose kind is **var**. The name of the object is the external name of the formal connection and the type of the object is the type of the expression.
- Otherwise, the assignment statement declares an object of class analog local variable whose kind is **val** and whose name is the simple name and whose type is the type of the expression.

It is an error if the type of the expression cannot be determined. It is also an error if a simple assignment statement whose target has not been declared appears in a function body.

If the expression is not a simple expression, then the assignment statement is equivalent to a sequence of simple assignment statements, one for each simple expression in the expression. Each simple assignment statement in the equivalent sequence is decorated with the statement attributes, if any, of the assignment statement. The target of each simple assignment statement in the equivalent sequence is determined as follows.

- If the assignment statement has no target, then each simple assignment statement in the equivalent sequence has no target.
- If the target of the assignment statement is a name that denotes an object, then this name is the target of the first simple assignment statement in the equivalent sequence. The remaining simple assignment statements in the equivalent sequence have no target.
- If the target of the assignment statement is an inline group or a name that denotes a group, then the target, if any, of each simple assignment statement in the equivalent sequence is the name of the object at the matching position in the canonical group corresponding to the group. It is an error if the cardinality of the group is greater than the cardinality of the expression.

It is an error if the target of an assignment statement that is not a simple assignment statement is a name that has not been declared. It is also an error if an assignment statement that is decorated with the **equations** attribute has

no target or is not a simple assignment statement. Similarly, it is an error if an assignment statement decorated with the **values** attribute has no target.

Some simple assignment statements in a template body are implicitly decorated with a statement attribute, as follows.

- If the object denoted by the target is of class parameter, or if the assignment statement has no target and the expression is a globally constant expression, then the simple assignment statement is implicitly decorated with the **parameters** attribute.
- If the object denoted by the target is a branch variable or an analog system variable of kind **var**, then the simple assignment statement is implicitly decorated with the **equations** attribute.
- If the object denoted by the target is an analog local variable of kind val or a simulator variable, and if the assignment statement is not decorated with the **equations** attribute (implicitly or explicitly), then the simple assignment statement is implicitly decorated with the **values** attribute.

It is an error if the target denotes an analog system variable of kind **ref**. It is also an error if the target denotes an object of class state and the assignment statement is not guarded by a when statement. Finally, it is an error if a subelement of the target of a simple assignment statement is of an unconstrained array type or of the predefined type STRING and the simple assignment statement is neither a function statement nor a statement decorated with the **parameters** attribute.

The simple assignment statements in the equivalent sequence corresponding to an assignment statement in a template body must belong to one of the following categories:

- Each simple assignment statement is decorated with only the **parameters** attribute.
- Each simple assignment statement is decorated with either the **values** attribute or the **equations** attribute, but not with any other attributes.
- Each simple assignment statement is decorated only with the **states** attribute.

It is an error if an assignment statement is decorated with the **parameters** attribute and the expression is not globally constant. It is also an error if the target of an assignment statement is decorated with the **external** or the **const** attribute.

An assignment statement that is decorated with the **equations** attribute defines the stamp expression (see Equation Statements) expression - target. The

execution of an assignment statement decorated with the **equations** attribute consists of the following steps for each scalar subelement S of the target:

1. An analog net N is selected as follows:
   - If S is a branch through variable or an analog system variable of kind var, then N is the analog net associated with S. It is an error if S has been marked.
   - Otherwise, N is an analog net that is associated with an unmarked scalar subelement of a branch through variable or an analog system variable of kind var that is not a subelement of the target of an assignment statement decorated with the equations attribute. It is an error if no analog net can be selected.
2. The characteristic expression associated with N is replaced by an expression that is the sum of the characteristic expression and the scalar subelement corresponding to S of the stamp expression.
3. The analog variable associated with N is marked.

For the execution of an assignment statement that is not decorated with the **equations** attribute, the expression is evaluated first. Then, the following steps are performed for a simple assignment statement or for each simple assignment statement in the sequence equivalent to the assignment statement:

1. The target, if present, of the simple assignment statement is evaluated; its type must be type compatible with the type of the simple expression.
2. If the statement is decorated with the **parameters** attribute, then the target, if present, of the simple assignment statement is update elaborated (see Update Elaboration of an Object) in the context of the value of the simple expression.
3. The value of the simple expression becomes the new value of the object denoted by the target, if present.

### Notes

1. The type of the expression cannot be determined if, for example, the expression is a structure aggregate or a union aggregate, or if a name appears in the expression that has not been declared.
2. The execution of an assignment statement without target has no effect unless the evaluation of the expression has a side effect (for example, writing a string).
3. Certain semantic checks can only be made during the execution of an assignment statement if the expression is a function call calling a foreign function with an unspecified profile.

### Rationale

The introduction of statement attributes provides for a consistent treatment of templates written with and without sections.

The definition of an assignment statement with optional target instead of separate assignment statement and expression statement provides for consistent treatment of statements whose expression is a group (e.g. the result of a function call) and whose target has a cardinality that is smaller than the cardinality of the group.

These definitions allow the target of an assignment statement to be an unconstrained array only if the target is of class parameter. The reason for this restriction is simulation efficiency.

---

## Loop Statement

A loop statement executes a statement repeatedly.

loop_statement ::=
*[* statement_attribute *]* iteration_scheme *{* format_effector *}* statement

iteration_scheme ::=
**while** ( condition )
| **for** ( *[ initial*_assignment *]* ; *[* condition *]* ; *[ update*_assignment *]* )

assignment ::=
simple_name = expression

The simple name in an assignment must denote an object of a discrete type or a numeric type. The type of the expression must be type compatible with the type of the object.

A loop with a **for** iteration scheme with no condition is equivalent to a loop with a **for** iteration scheme whose condition is the integer literal 1.

The statement is said to be *guarded* by the loop statement. Additionally, the statement is guarded by any statement that guards the loop statement. These statements are collectively called the *guards* of the statement.

A loop statement is implicitly decorated with the statement attributes of the statement that it guards.

The execution of a loop statement with a **while** iteration scheme consists of the execution of one or more iterations. The execution of one iteration consists of the evaluation of the condition and the execution of the statement.

The execution of a loop statement with a **for** iteration scheme consists of the execution of an initial iteration followed by the execution of subsequent iterations, if any. The execution of the initial iteration consists of the execution of the initial assignment, if present, followed by the evaluation of the condition, followed by the execution of the statement. The execution of any subsequent iteration consists of the execution of the update assignment, if present, followed by the evaluation of the condition, followed by the execution of the statement. For the execution of an assignment, the expression and the simple name are evaluated, then the value of the expression becomes the new value of the object denoted by the simple name.

Execution of an iteration is complete when the statement has been executed in its entirety, or when a next statement associated with the loop statement is executed, or when the loop statement completes. Execution of a loop statement is complete when the value of the condition is zero, when an exit statement associated with the loop statement is executed, or when a return statement appearing in the same major declarative region is executed. A loop statement is erroneous if it never completes.

### Examples

```
p2 = 1                      # Determine the smallest
while (p2 < 100)            # power of 2 larger
       p2 = 2 * p2          # than 100

fac = 1                     # Determine n factorial
for (i=2;i<=n;i=i+1)
       fac = fac * i
```

### Note

It is a consequence of these rules that a loop with a for iteration scheme is equivalent, except for the behavior of the next statement, to the following sequence of statements:

*initial*_assignment ;
**while** ( condition ) {
statement
*update*_assignment ;
}

### References

Conditional Expressions

---

## Exit Statement

An exit statement is used to complete the execution of a loop statement.

exit_statement ::=
**exit** eos

An exit statement is associated with the innermost loop statement guarding the exit statement. It is an error if an exit statement is not guarded by a loop statement.

The execution of an exit statement completes the execution of the loop statement associated with the exit statement.

### Note

Several exit statements may be associated with the same loop statement.

## Next Statement

A next statement is used to complete the execution of one iteration of a loop statement.

next_statement ::=
**next** eos

A next statement is associated with the innermost loop statement guarding the next statement. It is an error if a next statement is not guarded by a loop statement.

The execution of a next statement completes the execution of the current iteration of the loop statement associated with the next statement.

**Note**

Several next statements may be associated with the same loop statement.

## Return Statement

A return statement is used to complete the execution of a function body.

return_statement ::=
**return** eos

It is an error if a return statement appears in a template body.

The execution of a return statement completes the execution of a function body.

## When Statement

A when statement defines a portion of the event-driven behavior of a design entity.

when_statement ::=
*[* statement_attribute *]*
**when** ( condition ) *{* format_effector *}* statement

The statement is said to be *guarded* by the when statement. The when statement is called the *guard* of the statement. It is an error if a when statement is a guarded statement.

A when statement is implicitly decorated with the states attribute and with the statement attributes of the statement that it guards.

A when statement is said to be *sensitive* to a state S if its condition includes one of the following:

A function call calling the predefined function EVENT_ON whose first argument is S.

A function call calling the predefined function THRESHOLD. S is an implicit state maintained by the kernel.

A name that denotes one of the simulator variables DC_INIT, DC_START, DC_DONE, TIME_INIT, TR_START, TR_DONE, or TIME_STEP_DONE. S is the state corresponding to the simulator variable.

It is an error if a when statement is not sensitive to any state.

For the execution of a when statement the condition is evaluated first. If the condition evaluates to a nonzero value, then the statement guarded by the when statement is executed.

## Equation Statements

Equation statements, together with assignment statements decorated with the **equations** attribute, define *stamp expressions* that are used to construct the *characteristic expressions* used by the analog solver to determine the values of the analog variables. Characteristic expressions constructed from stamp expressions are called *explicit*.

equation_statement ::=
  contribution_statement
| labeled_equation_statement
| make_statement

Each equation statement is implicitly decorated with the **equations** attribute. It is an error if a name that denotes an imported object of class analog local variable appears in an expression in a statement decorated with the **equations** attribute.

## Contribution Statement

A contribution statement replaces characteristic expressions specified by a name.

contribution_statement ::=
*[* statement_attribute *]* name contribution_op expression eos

contribution_op ::= += | -=

The name must be a constant name that is one of the following:

- A branch name that denotes a through branch.
- The name of a branch through variable.
- The name of an analog system variable of kind **var** or **ref**.

The expression must be a simple expression whose type is type compatible with the type of the name, and for each scalar subelement of the name there must be a matching scalar subelement of the expression.

The stamp expression defined by the contribution statement is determined as follows. If the name is an analog system variable of kind var or ref, then the stamp expression defined by the contribution statement is the expression. Otherwise, the stamp expression defined by the contribution statement is expression - name.

The execution of a contribution statement consists of the following steps for each scalar subelement S of the object denoted by the name:

1. The analog net N associated with S is selected. It is an error if S has been marked.
2. The characteristic expression associated with N is replaced by an expression that is:
   - if the contribution operator is +=, the sum of the characteristic expression and the scalar subelement corresponding to S of the stamp expression.
   - if the contribution operator is -=, the difference of the characteristic expression and the scalar subelement corresponding to S of the stamp expression.
3. If S is a subelement of a branch through variable, then S is marked.

## Labeled Equation Statement

A labeled equation statement replaces characteristic expressions specified by a label.

labeled_equation_statement ::=
*[* statement_attribute *]* name : *lhs*_expression = *rhs*_expression eos

The name is said to be the *label* of the labeled equation statement; it must be a constant name that denotes an analog system variable of kind **var** or an explicitly declared branch through variable. It is an error if the label is a name that denotes an imported object.

Both expressions must be simple expressions of compatible constant types. Their types must be type compatible with the type of the label, and for each scalar subelement of the label there must be a matching scalar subelement of the lhs expression and of the rhs expression.

The stamp expression defined by the labeled equation statement is *lhs*_expression - *rhs*_expression.

The execution of a labeled equation statement consists of the following steps for each scalar subelement S of the label:

1. The analog net N associated with S is selected. It is an error if S has been marked.
2. The characteristic expression associated with N is replaced by an expression that is the sum of the characteristic expression and the scalar subelement corresponding to S of the stamp expression.
3. S is marked.

**Note**

It is a consequence of these definitions that a labeled equation statement replaces characteristic expressions in an identical way as the contribution statement name += *lhs*_expression - *rhs*_expression.

## Make Statement

A make statement replaces unspecified characteristic expressions.

make_statement ::=
*[* statement_attribute *]* **make** *lhs*_expression = *rhs*_expression eos

Both expressions must be simple expressions of compatible constant types with scalar subelements of a numeric type, and for each scalar subelement of the lhs expression there must be a matching scalar subelement of the rhs expression.

The stamp expression defined by the make statement is *rhs*_expression - *lhs*_expression.

The execution of a make statement consists of the following steps for each scalar subelement S of the stamp expression:

1. An analog net N that is associated with an unmarked scalar subelement of a branch through variable or an analog system variable of kind **var** is selected. It is an error if no analog net can be selected.
2. The characteristic expression associated with N is replaced by an expression that is the sum of the characteristic expression and S.
3. The analog variable associated with N is marked.

**Note**

The order in which analog nets are selected is not defined by the language.

## Nonexecutable Statements

Nonexecutable statements define the structural composition of a design entity.

nonexecutable_statement ::=
instantiation_statement

It is an error if a nonexecutable statement is a guarded statement.

### Instantiation Statement

An instantiation statement defines a component of the design entity in which it appears and specifies the actual connections, if any, to be associated with the formal connections of the component and the actual arguments, if any, to be associated with the formal arguments of the component. The component is one instance of the design entity denoted by the prefix of the instance name.

instantiation_statement ::=
    instance_name connection_association_list
        *[* = *instance*_argument_association_list *]* eos

connection_association_list ::=
    *{* connection_association_element *}*

connection_association_element ::=
    *[* connection_formal_part : *]* connection_actual_part

connection_formal_part ::= extended_name

The instantiation statement declares the instance name by associating it with the component. It is an error if the design entity denoted by the prefix of the instance name has not been analyzed.

The connection association list establishes a correspondence between the formal connections, the pins decorated with the **external** attribute, and the analog system variables decorated with the **external** attribute of an instance of a design entity, and actual pins, states, and analog system variables declared in the design entity in which the instantiation statement appears. Each connection association element associates one actual object with a formal connection of the instance, or a pin or analog system variable decorated with the **external** attribute of the instance, or a subelement thereof. The corresponding formal connections are determined either by position or by name.

An connection association element is said to be *named* if it contains a connection formal part; it is said to be *positional* otherwise. For a named connection association element the connection formal part must specify either the external name of a formal connection of the design entity denoted by the prefix of the instance name, or a subelement, a slice, or a slice of a subelement, of such an external name. For a positional connection association element the formal connection is implicitly specified by the textual position of the connection association element in the connection association list.

Named connection association elements may appear in any order, but if named and positional connection association elements appear in the same connection association list, then any named connection association element must follow all positional connection association elements.

The connection formal part and the connection actual part of a connection association element must both be constant names.

If the connection actual part of a connection association element is specified by a simple name or a decimal name that has not been declared, then the connection association element implicitly declares an object:

- If the simple name or decimal name is the external name or internal name of a template connection element of the design entity in which the instantiation statement appears, then the connection association element declares an object whose name is the external name of the template connection element. The corresponding declaration is deemed to occur following any previously declared header declarative item.
- Otherwise, the connection association element declares an object whose name is the simple name or the decimal name. The corresponding declaration is deemed to occur in the major declarative region formed by the template in which the instantiation statement appears, immediately before the instantiation statement or any compound statement whose declarative region encloses the instantiation statement.

The class, subclass, and type or unit of the implicitly declared object is the class, subclass, and type or unit, respectively, of the object denoted by the connection formal part of the connection association element. If the object is of class analog system variable, then its kind is **ref**. It is an error if a connection association element implicitly declares an object and the type of the connection formal part is not locally constant. It is also an error if the object denoted by the connection actual part has not been declared and the name is neither a simple name nor a decimal name.

The objects denoted by the connection actual part and the connection formal part of a connection association element must satisfy one of the following rules.

- Both objects are of class pin and are pin type compatible.
- The object denoted by the connection formal part is an analog system variable of kind **ref**, the object denoted by the connection actual part is an analog system variable, and the two objects are unit compatible.
- The object denoted by the connection formal part is an analog system variable of kind **var**, the object denoted by the connection actual part is an analog system variable of kind **ref** or a branch through variable, and the two objects are unit compatible.
- Both objects are of class state and are unit compatible. If the object A denoted by the connection actual part has a mode, then one of the following conditions must be true:
    - The mode of the object denoted by the connection formal part is **input** and the mode of A is either **input** or **inout**.

- The mode of the object denoted by the connection formal part is **output** and the mode of A is either **output** or **inout**.
- The mode of the object denoted by the connection formal part is **inout** and the mode of A is **inout**.

If the connection formal part of a connection association element denotes an object of a composite type or pin type, then the connection association element associates each scalar subelement of the connection formal part with the corresponding scalar subelement of the connection actual part. If a subelement of the formal connection is of an unconstrained array type or an unconstrained array pin type, then the upper bound of each assumed index range of the subelement is defined by the association such that the length of the index range is equal to the length of the index range of the corresponding subelement of the actual connection at the same index position.

If the name of the connection formal part denotes an entity declared in the template denoted by the prefix of the instance name, then the formal connection is said to be associated in whole. Otherwise, the formal connection is said to be associated individually. For a formal connection that is associated individually, each scalar subelement must be associated exactly once with an actual connection, or a subelement thereof, in a connection association list.

A connection association element need not include a connection association element for a formal connection. It is an error if a formal connection is associated more than once with an actual connection in a connection association list.

The instance argument association list associates actual values with the formal arguments of the instance and with parameters declared in the template denoted by the prefix of the instance name that are decorated with the **external** attribute. It is an error if the actual expression in any argument association element of the instance argument association list is not a relaxed globally constant expression.

### Examples

```
state number s1, s2, s3[2]
electrical dc
...
template driver bus
state number bus[*]
external electrical vcc
{ ... }

driver.1 s3                    # bus has length 2

driver.2 bus[1]:s1 \           # bus has length 4
       bus[2]:s2 bus[3:4]:s3

driver.3 s3 vcc:dc             # bus has length 2, ex-
                               # ternal pin associated
                               # explicitly

driver.4 s1                    # illegal: type mismatch

driver.5 bus[2:3]:s3           # illegal: no association
                               # for bus[1]
```

### Notes

1. A formal connection denoted by a decimal name can only be associated in whole.
2. For a formal connection associated individually, the requirement that each scalar subelement be associated exactly once uniquely defines the index ranges of any subelement of the object denoted by the connection formal part that is of an array type or an array pin type.
3. The upper bound of a subelement of a formal connection that is of an unconstrained array type cannot be determined until the model is elaborated if the corresponding subelement of the actual connection is itself of an unconstrained array type.
4. Pins and analog system variables decorated with the **external** attribute can only be associated in a named connection association element since they are not part of the connection list.

### Rationale

The type or unit of the connection formal part of a connection association element that implicitly declares an object must be locally constant to allow the

elaboration of a design hierarchy to proceed in a top-down fashion. Without this restriction certain instances would have to be elaborated bottom-up, and complicated rules would have to define what is legal. This has an implication on the specification of array connections: An actual object associated with a formal connection that has a subelement of an array type must be declared explicitly unless the index ranges of the subelement of the object denoted by the connection formal part are locally constant.

Since the connection actual part of a connection association element can be specified as a decimal name, it is not possible to support the association of a formal connection with an expression.

The requirement that the design unit denoted by the prefix of the instance name be analyzed, not just declared, makes it illegal to write an instantiation statement that instantiates a design unit that is being analyzed (i.e., recursive instantiations).

## Generic Statements

Generic statements can be used to define the algorithm in a function, the behavior or the structure of a design entity, or specifications of a design entity.

```
generic_statement ::=
    conditional_statement
  | compound_statement
```

A generic statement decorated, either explicitly or implicitly, with the **parameters**, **states**, **values**, or **equations** attribute defines the behavior of a design entity. A generic statement decorated, either explicitly or implicitly, with the **control_section** attribute defines specifications of a design entity. A generic statement decorated with no attributes defines the structure of a design entity or the algorithm in a function.

### Conditional Statement

A conditional statement selects for execution or elaboration one or none of the statements, based on the value of a condition.

conditional_statement ::=
*[* statement_attribute *]* **if** ( condition ) *{* format_effector *}*
then_part *[* else_part *]*

then_part ::=
*[* **then** *{* format_effector *} ]* conditional_sentence

else_part ::=
**else** *{* format_effector *}* conditional_sentence

conditional_sentence ::=
statement
| control_section_specification

The else part is associated with the closest conditional statement in the declarative region that does not have an else part. The statements of the then part and the else part are said to be *guarded* by the conditional statement. Additionally, the statements of the then part and the else part are guarded by any statement that guards the conditional statement. These statements are collectively called the *guards* of the statements of the then part and the else part.

A conditional statement is implicitly decorated with the statement attributes of the statements that it guards. It is an error if a conditional statement is decorated with the **parameters** or **control_section** attribute and the condition is not globally constant. It is also an error if a name that denotes an imported object of class analog local variable is a primary in the condition of a conditional statement that is decorated with the **equations** or **values** attribute. Finally, it is an error if the conditional sentence in the then part or else part of a conditional statement decorated with the **control_section** attribute is an executable statement or a nonexecutable statement, or if the conditional sentence in the then part or else part of a conditional statement not decorated with the **control_section** attribute is a control section specification.

For the evaluation of a conditional statement, the condition is evaluated first. If the value of the condition is nonzero, then the statement of the then part is selected. Otherwise, if the conditional statement includes the else part, then the statement of the else part is selected. Otherwise, no statement is selected.

### Example

```
if (a > 0)
   if (b > 0)c = 1
else                    # Associated with if statement
   if (b > 0)c = 0      # on preceding line
   elsec = -1           # Associated with if statement
                        # on preceding line
```

After evaluation of the statement, if the value of a is greater than 0, then the value of c is either 1 or -1, but never 0. If the value of a is equal to or less than 0, then the value of c is undefined.

### References

Conditional Expressions

## Compound Statement

A compound statement groups a sequence of sentences into a single syntactic unit.

compound_statement ::=
*[* statement_attribute *]* "{" *{* compound_sentence *}* "}" eos

compound_sentence ::=
  compound_declarative_item
| statement
| eos

compound_declarative_item ::=
  type_declaration
| unit_declaration
| variable_declaration
| simulator_variable_declaration
| group_declaration
| control_section_specification

Each statement that is a sentence in a compound statement is guarded by any statement that guards the compound statement.

The compound statement is implicitly decorated with the statement attributes of any statement that is a sentence in the compound statement. Conversely, each

statement that is a sentence in a compound statement is implicitly decorated with the statement attributes of the compound statement.

It is an error if a compound declarative item or a statement that is not allowed in the declarative region enclosing the compound statement is a compound sentence. It is also an error if an executable statement, a nonexecutable statement, or a compound declarative item other than a control section specification appears in a compound statement that is decorated with the control_section attribute.

### Note

It is a consequence of these rules that the statement attributes of a compound statement and of all statements that are sentences in the compound statement are the same.

# 9

# Scope and Visibility

*This chapter describes the rules defining the scope of a declaration and the rules defining which identifier is visible at various places in the text of a description.*

## Declarative Regions

A declarative region is a portion of the text of a description. The following each form a single declarative region:

- The predefined language environment
- A compilation unit
- A template header together with the template body
- A function header together with the function body
- A structure type declaration
- A structure pin type declaration
- A union type declaration
- All statements in a template that are decorated with the **control_section** attribute. The declarative region includes all such statements in the order in which the statements occur.
- A compound statement that is not decorated with the control_section attribute.

In each of these cases the declarative region is said to be *associated* with the corresponding statement or declaration.

There are two kinds of declarative regions. A compound statement is a *minor declarative region*, all other declarative regions are *major declarative regions*.

Declarative regions can be nested. A declaration is said to occur *immediately within* a declarative region if this region is the innermost declarative region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

Although some declarative regions include disjoint parts, they are considered logically continuous portions of text.

## Scope of Declarations

The *scope* of a declaration is the portion of a description over which that declaration is active. The scope of the declaration is also the scope of any entity declared by the declaration and the scope of any identifier associated with an entity by the declaration. It is legal to refer to an entity only at certain places within its scope. These places are defined by the rules of visibility and overloading.

The *immediate scope* of a declaration is a portion of the declarative region immediately enclosing the declaration. The immediate scope extends from the beginning of the declaration to the end of the declarative region. In addition, the scope of the following declarations extends beyond the immediate scope to the end of the scope of the enclosing declaration:

- A declaration decorated, either explicitly or implicitly, with the **export** attribute
- A declaration of the type or pin type of a formal connection
- A declaration of a function argument
- An element declaration in a structure type declaration
- An element declaration in a union type declaration
- An element pin declaration in a structure pin type declaration

**Note**

The scope rules apply to all kinds of declarations, including implicit declarations.

## Visibility

The visibility rules and, in the case of overloaded declarations, the overloading rules, determine the meaning of a given identifier appearing at a particular place in the text of a description.

A declaration is said to be *directly visible* if the identifier associated with an entity by the declaration is sufficient to denote the entity. A declaration is *visible* wherever it is directly visible, and additionally at places where some name (such as a selected name or a qualified name) can denote the entity.

A declaration is directly visible within certain parts of its scope. For a template declaration this part starts at the end of the template header definition; for all other declarations the part starts at the end of the declaration element that associates the identifier with an entity. Additionally, a declaration is visible at the following places:

1. For a design unit contained in a design library: at the place of the suffix in a qualified name whose prefix denotes the library.
2. For the external name of a template connection element: in the connection formal part of a connection association element of an instantiation statement whose instance_name prefix denotes the template; also at the place of the suffix of a qualified name whose prefix denotes the template.
3. For the name of the type of a formal connection: in the implicit declaration of the actual connection associated with the formal connection.
4. For a template argument: at the place of the formal in an argument association element of the argument association list of a corresponding instantiation statement; also at the place of the suffix of a qualified name whose prefix denotes the template.
5. For a function argument: at the place of the formal in an argument association element of the argument association list of a corresponding function call.
6. For a declaration decorated, either explicitly or implicitly, with the **export** attribute: at the place of the simple name in an imported name whose instance name has a prefix that denotes the template containing the declaration.

7. For a variable declaration appearing immediately within a structure type declaration: at the place of the simple name in a selected name whose prefix denotes an object of the structure type; also at the place of the simple name in an association element of a structure aggregate or structure overlay of the type.
8. For a pin declaration appearing immediately within a structure pin type declaration: at the place of the simple name in a selected name whose prefix denotes an object of the structure pin type.
9. For a variable declaration appearing immediately within a union type declaration: at the place of the simple name in a selected name whose prefix denotes an object of the union type; also at the place of the simple name in the single association element of a union aggregate of the type.

A declaration can be *hidden* within certain parts of its scope. For a declaration appearing immediately within a declarative region the following rules apply:

1. A group declaration or an object declaration other than a simulator variable declaration is hidden in any nested major declarative region.
2. A simulator variable declaration is hidden in any nested major declarative region that is associated with a function definition.
3. A declaration, template definition, or function definition is hidden within the immediate scope of another declaration that associates the same identifier with an entity in the same overloading class.

It is an error if a declaration is hidden by another declaration and the immediate scope of the hidden and the hiding declarations end at the same place.

### Examples

```
template x p m = y, z   # legal since the declaration for
electrical p, m         # template x is visible at the
number y                # end of line 1
x..y z = 0
{...}

template t1 a b         # error: a and b have no legal
{val v vab              # interpretation, since object
vab = v(a,b)            # class and type are unknown

x.1 a b = 1k
}

template t2 a b         # per 8.2.1 a and b are declared
{val v vab              # as pins of pin type electrical
x.1 a b = 1k            # legal: a and b now have a legal
                        # interpretation
vab = v(a,b)
}

unit {"a","b","c"} u    # declaration hides template x
number n = 1            # defined above
template t3
{                       # declaration hides predefined
 function r = x(a)      # simulator variable time
 number r, a            # error: although unit u is
 {...}                  # visible here, number n is not
 number time            # error: template x is not visible
 ...
 number u v = n
 ...
 x.1 a b = 1k
}
```

### Note

It is a consequence of the visibility rules that it is an error if more than one declaration associating the same identifier with an entity in the same overloading class appears in the same declarative region.

# Overload Resolution

An identifier that is associated with more than one entity in a declarative region is said to be *overloaded*. Overloading is defined for names and enumeration literals. Overload resolution determines the entity to which an identifier refers in a particular context when according to the visibility rules more than one meaning is acceptable.

## Overloading Classes

Overload resolution uses the concept of *overloading classes* (also called *name spaces*). Within each overloading class the identifiers denoting members of that class must be unique in a declarative region.

The language predefines the following overloading classes:

1. Names of design libraries, templates and functions
2. Names of units
3. Names of types and pin types
4. Names of parameters, variables, states, analog variables, pins, simulator variables, and groups
5. Instance names
6. Names of unreserved keywords

Within overloading class 1 (Names of design libraries, templates and functions) the names of foreign functions form a subclass whose members must be unique among all declarative regions in a design.

Other overloading classes may be defined as follows.

1. The definition of an enumeration type defines an overloading class containing the enumeration literals of the type.

### Examples

```
unit {"a","b","c"} x          # declares unit x
pin x across v through i      # declares pin type x

template x x                  # declares template x
x x                           # declares connection x as
{...}                         # a pin of pin type x

template y x = y, z           # declares template y
state x x                     # declares connection x as a
number y                      # state of unit x
y..y z = 1                    # number y and template y are
{...}                         # in different overloading
                              # classes, so both are visible

template z z = z              # illegal: the objects declared
...                           # in a template header are all
                              # in the same overloading class
```

### Notes

1. The entity associated with an unreserved keyword is the keyword itself. Similarly, the entity associated with a control_section specification is the specification itself.
2. It is a consequence of these definitions and the definitions in Compilation Units that a template or function cannot be named WORK.

### Rationale

Functions and templates are design units that can be compiled separately. The ability to uniquely identify a design unit by its name mandates that templates and functions are in the same name space. Design libraries are in the same name space because a qualified name is used to specify both an entity in a template and a design unit in a design library.

## Overload Resolution

When the visibility rules have determined that an identifier appearing at a particular place in a sentence may refer to more than one entity, all visible declarations that associate this identifier with an entity are considered. It is an error if there is not exactly one interpretation for the identifier at that place.

When considering possible interpretations, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below.

1. Any rule that requires the entity associated with the identifier to be of a certain class, or of the same class as another name.
2. Any rule that requires the entity associated with the identifier to be of a certain type, the *expected type*, or of the same type as another name or expression.

**Note**

If there is only one interpretation for an identifier appearing at a particular place, the identifier denotes the corresponding entity. However, the appearance may still be illegal because of other rules that are not considered for overload resolution, for example, whether an expression must be a constant expression.

# 10

# Design Units and Their Compilation

*This chapter describes the organization and compilation of a description.*

## Design Units

A *design unit* is a portion of code that can be independently compiled and inserted into a design library.

```
design_unit ::=
        library_unit

library_unit ::=
          root_template
        | template_definition
        | function_definition
```

The name of a library unit other than a root template is the name of the template or function. The name of a root template is defined by the implementation.

**Note**

It is a consequence of these rules that a template definition that is a declarative item in a template body is not a design unit. Similarly, a function definition that is a declarative item in a template body or in a function body is not a design unit.

**References**

Design Entities, Functions

## Contexts

A *context* is a (possibly empty) collection of declarations.

context ::=
*{* context_sentence *}*

context_sentence ::=
context_declarative_item
| eos

context_declarative_item ::=
function_declaration
| type_declaration
| unit_declaration
| pin_type_declaration
| parameter_declaration
| state_declaration
| ref_declaration
| pin_declaration
| template_definition

## Compilation Units

The text of a description is compiled one compilation unit at a time in a compilation environment defined by one or more contexts.

compilation_unit ::=
*{* named_library_unit } [ root_template *]*

named_library_unit ::=
template_definition
| function_definition

A compilation unit consists of an arbitrary number (including zero) of design units in one or more source files. A compilation unit consisting of more than one source file is formed by concatenating the source files in the order in which the source files are presented. The sentences in the compilation unit are compiled in the order in which they appear.

The compilation environment is formed by the concatenation of one or more contexts. The implementation must establish the compilation environment in such a way that the declarations of the contexts that form the compilation environment are visible prior to the first sentence of the compilation unit.

**Notes**

1. This specification does not define how an implementation should manage contexts and how it should establish the compilation environment.
2. An implementation may restrict the number of source files per compilation unit.

## Design Libraries

A *design library* is an implementation-dependent container of compiled design units. Each design library has an associated symbolic name that is an identifier. An implementation may support any number of design libraries.

There are two kinds of design libraries: working libraries and *resource libraries*. A design unit may contain references to library units contained in resource libraries. For each compilation there is a single *working library* denoted by the symbolic name WORK into which the result of a successful compilation is entered. For a particular compilation the same library may be both the working library and a resource library.

**Note**

The means of associating a symbolic name with a design library are defined by the implementation.

## Order of Compilation

The order in which design units can be compiled is a direct consequence of the visibility rules. In particular, a design unit A that is referenced in a given design unit B must be compiled prior to the compilation of design unit B. Further, if design unit A changes, then design unit B must be recompiled since it depends on design unit A. Similarly, if a context changes, any design unit whose compilation environment includes the context must be recompiled.

If an error is encountered during the compilation of a design unit, then the compilation of that design unit has no effect. Otherwise, the resulting compiled design unit is entered into the working library, thereby replacing any design unit with the same name that may be present in the working library.

# 11

# Elaboration

*This chapter describes the process by which a sentence takes effect, called elaboration. Elaboration is defined for a design hierarchy and for the declarative items (including implicit declarations) and statements of the language.*

## Elaboration of a Design Hierarchy

Elaboration of a design hierarchy consists of the elaboration of each context that forms the compilation environment of a root template followed by the elaboration of the template body defining the root template. The instance corresponding to the root template is called the *root instance*.

The result of elaborating a design hierarchy is a *simulatable model* that consists of a tree of instances, a collection of when statements in these instances connected by nets, and a set of *characteristic expressions*. The behavior of the design can be simulated by executing the when statements to determine the values of the nets, and evaluating the characteristic expressions while determining the values of the analog variables.

**Note**

The means of specifying the root template for a design hierarchy are defined by an implementation.

## Elaboration of Declarative Items

### Contexts

The elaboration of a context consists of the elaboration of the declarations in the context in the order in which they occur. If the declarations include an object

declaration, then the resulting object is created in the root instance. It is an error if an object with the same name already exists in the root instance.

## Template Headers

Elaboration of a template header consists of two parts.

Part 1 of the elaboration of a template header consists of the elaboration of each unelaborated context in the compilation environment of the template, followed by the elaboration of each template header declarative item and each implicit declaration in the template header that is a type declaration, a unit declaration, a pin type declaration, or a parameter declaration. The declarative items are elaborated in the order in which they occur.

Part 2 of the elaboration of a template header consists of the elaboration of the remaining template header declarative items and implicit declarations in the template header in the order in which they occur.

## Template Bodies

Elaboration of a template body consists of the following steps:

1. Elaboration of all declarations except the following declarations:
   - type declarations declaring globally constant types
   - pin type declarations
   - state declarations
   - analog variable declarations
   - pin declarations
   - group declarations of nonparameter groups

   and part 1 of the elaboration of all alter specifications, in the order in which the declarative items occur in the major declarative region associated with the template.
2. Execution of all statements decorated with the **parameters** attribute. This may cause an update elaboration of some objects.
3. Elaboration of all remaining declarations, except the declaration of imported objects, and part 2 of the elaboration of all alter specifications, in the order in which the declarative items occur in the major declarative region associated with the template.

4. Reordering of the instantiation statements such that an instantiation statement whose argument association list contains an argument association element whose actual expression contains a name that denotes an imported object does not precede the instantiation statement declaring the object associated with the imported object. It is an error if no such ordering is possible.
5. Part 1 of the elaboration of each instantiation statement in the order established by Step 4.
6. Elaboration of the implicit declarations of imported objects of class pin or analog variable. Additionally, the elaboration includes one of the following:
   - If the imported object is a pin or an analog system variable, then an implicit connection association element is created whose connection actual part is the imported object and whose connection formal part is the object associated with the imported object. The connection association element is elaborated in turn.
   - If the imported object is an analog local variable, then an implicit assignment statement is created whose target is the imported object and whose expression is the object associated with the imported object. The statement is decorated with the **values** attribute and is deemed to precede any other statement decorated with the values attribute.
7. Elaboration of statements decorated with the **values** attribute.
8. Elaboration of statements decorated with the **control_section** attribute.
9. Determination of the tolerance range of each scalar subelement of each analog variable (see Determination of Tolerance Range).
10. Part 2 of the elaboration of each instantiation statement in the order established by Step 4.

**Note**

The algorithm described in Step 4 defines a partial ordering of the instantiation statements of an instance. Partial ordering is not possible if circular references exist.

**Rationale**

The elaboration order is dictated by the information necessary at each step. Step 1 makes declarations available that are necessary to execute statements decorated with the **parameters** attribute. In Step 3 the declarations that may depend on Step 2 can be elaborated. Step 5 and Step 6 make imported objects available for the subsequent steps, in an order established by Step 4 such that no forward references are necessary. Step 7 classifies the analog variables into

dependent and independent variables, which is necessary for Step 8. Step 9 defines the tolerance range of each analog variable, which may be necessary for Step 10.

### Determination of Tolerance Range

The tolerance range of a scalar subelement S of an analog variable V is determined as the first tolerance range found when using the following algorithm:

1. If V is the connection formal part in a connection association element whose connection actual part is A, then the tolerance range of S is the tolerance range of the corresponding scalar subelement of A.
2. If V is an across branch whose plus pin is P and whose minus pin is the reference pin 0, and if P is the connection formal part in a connection association element whose connection actual part is A, then the tolerance range of S is the tolerance range of an across branch whose plus pin is A and whose minus pin is the reference pin 0.
3. If a variable range specification applies to V, then the tolerance range of S is the tolerance range defined by that variable range specification.
4. If S is of unit U, and if a unit range specification applies to U, then the tolerance range of S is the tolerance range defined by that unit range specification.
5. If no range set specification applies to the instance I in which V has been declared, and if I is not the root instance, then the range set specification that applies to the parent of I also applies to I. The tolerance range of S is obtained in an implementation-dependent way from the range set specified by the range set specification that applies to I.

It is an error if this algorithm does not define a tolerance range for a scalar subelement of an analog variable.

## Type Declarations, Type Definitions, and Index Constraints

Elaboration of a type declaration consists of the elaboration of the corresponding type definition.

Elaboration of a scalar type definition creates the type.

Elaboration of a structure type definition consists of the elaboration of each element declaration in the order in which they occur, followed by the creation of the structure type.

Elaboration of a union type definition consists of the elaboration of each element declaration in the order in which they occur, followed by the creation of the union type.

Elaboration of an array type definition consists of the elaboration of each of its index constraints in an order not defined by the language, followed by the creation of the array type.

Elaboration of an index constraint consists of the evaluation of the lower bound and the upper bound.

## Unit Declarations

Elaboration of a unit declaration in general creates the unit.

Elaboration of a physical unit declaration additionally consists of the evaluation of the string expressions of the physical unit definition and the decoration of the unit with the corresponding attributes.

Elaboration of an enumeration unit declaration additionally consists of the creation of enumeration type corresponding to the enumeration unit definition, the decoration of each enumeration literal with its attributes, the association of the enumeration default value with the enumeration unit, and, if the enumeration unit declaration includes a resolution indication, of the association of the resolution function declared by the resolution indication with the enumeration unit.

Elaboration of a derived unit declaration additionally consists of the following. If the unit mark of the derived unit definition denotes a unit that is decorated with attributes, then the derived unit is decorated with these attributes. If the derived unit declaration includes a default value, then this default value is associated with the derived unit. If the derived unit declaration includes a resolution indication, then the resolution function declared by the resolution indication is associated with the derived unit.

## Pin Type Declarations and Pin Type Definitions

Elaboration of a scalar pin type declaration creates the scalar pin type.

Elaboration of a structure pin type declaration consists of elaborating the corresponding structure pin type definition.

Elaboration of a structure pin type definition consists of the elaboration of each element pin declaration in the order in which they occur, followed by the creation of the structure pin type.

Elaboration of an array pin type definition consists of the elaboration of each of its index constraints in an order not defined by the language, followed by the creation of the array pin type.

## Unit Marks, Unit Names, Type Marks and Pin Type Marks

Elaboration of a unit mark consists of the elaboration of the type mark or the unit name.

Elaboration of a unit name has no effect.

Elaboration of a type mark consists of the following:

- If the type mark is a type definition, then its elaboration consists of the elaboration of the type definition.
- If the type mark is a type reference defined by a qualified name whose prefix denotes a template, then its elaboration consists of part 1 of the elaboration of the template header of that template.
- Otherwise, the elaboration has no effect.

Elaboration of a pin type mark consists of the following:

- If the pin type mark is a structure pin type definition, then its elaboration consists of the elaboration of the structure pin type definition.
- Otherwise, the elaboration has no effect.

## Function Calls

Elaboration of a function call consists of the elaboration of the unelaborated contexts in the compilation environment of the corresponding function definition.

**References**

Function Calls

## Object Declarations

Elaboration of an object declaration consists of the following steps:

1. The type and/or unit of the object is determined.
   - If the object declaration declares a parameter or a variable, then the type mark or the unit name is elaborated.
   - If the object declaration declares a state or an analog variable of kind **var**, **ref**, or **val**, then the unit mark is elaborated.
   - If the object declaration declares a branch variable, then the unit name of the unit aspect of the branch name is elaborated.
   - If the object declaration declares a pin, then the pin type mark is elaborated.
   - If the object declaration declares a simulator variable, then its type is obtained from the predefined language environment.
2. The initial value expression, if present, is evaluated.
3. The object is created.
4. The object is decorated with its attributes.
5. If an initial value expression has been defined, then the initial value is assigned to the object.
6. If the object declaration is a template header declarative item that declares a parameter, then the alias for the type of the parameter is created.

Elaboration of some object declarations has additional effects. See below, Association of Parameters Decorated with the External Attribute and Association of Other Objects Decorated with the External Attribute.

**Rationale**

The initial value expression is evaluated before the creation of the object because it will define the size of an unconstrained array.

## Branch Variables

If the object declaration declares a branch variable, then its elaboration also creates an *analog net* for each scalar subelement S of the branch variable and associates S with the analog net. Additionally, if the object declaration declares a branch across variable, then its elaboration associates, for each scalar subelement B of the branch across variable, the characteristic expression B -

ACROSS(plus) + ACROSS(minus), where plus and minus are the plus pin and minus pin of B, with the corresponding analog net.

## Pins

The *pin flow expression* of a scalar subelement P of a pin is defined as follows. If P is the connection actual part in an implicit connection association element due to a collapse specification (see Collapse Specification), then the pin flow expression of P is:

sum of all branch through variables whose plus pin is P
- sum of all branch through variables whose minus pin is P

Otherwise, the pin flow expression of P is:

sum of all branch through variables whose plus pin is P
- sum of all branch through variables whose minus pin is P
+ sum of the pin flow expressions of all formal connections that are associated in a connection association element with P as the connection actual part

If the object declaration declares a pin that is not the connection formal part in any connection association element, then its elaboration also creates an *analog net* for each scalar subelement S of the pin and associates S with the analog net. Additionally, the pin flow expression of S is associated as a characteristic expression with the analog net.

**Note**

The analog net associated with a pin is often called a node.

## Unassociated Analog System Variables of Kind Var or Ref

If the object declaration declares an analog system variable of kind **var** or **ref** that is not a connection formal part in any connection association element, then its elaboration also creates an *analog net* for each scalar subelement S of the analog system variable and associates S with the analog net.

## Unassociated States

If the object declaration declares an event-driven state that is not a connection formal part in any connection association element, then its elaboration also creates a *net* for each scalar subelement of the state and associates the scalar subelement with the net. The type and unit of the net is the type and unit of the scalar subelement of the state. Additionally, if the state is a driving state, then the elaboration of its declaration associates the driver of each scalar subelement of the state (see Drivers) with the corresponding net.

## Update Elaboration of an Object

The update elaboration of an object in the context of a value consists of the following steps:

1. The value, if any, of the object is deleted.
2. If the object has a subelement of an unconstrained array type, then the upper bound of each assumed index range of the subelement is defined by the value such that the length of the index range is equal to the length of the index range of the corresponding subelement of the value at the same index position.

## Group Declarations

Elaboration of a group declaration creates the canonical group corresponding to the group.

## Alter Specifications

Elaboration of an alter specification consists of two parts.

Part 1 of the elaboration of an alter specification consists of the elaboration of each specifier in the specifier list whose simple name denotes a parameter, in the order in which the specifiers occur in the specifier list.

Part 2 of the elaboration of an alter specification consists of the elaboration of each specifier in the specifier list whose simple name does not denote a parameter, in the order in which the specifiers occur in the specifier list.

The elaboration of a specifier consists of the following steps:

1. The expression is evaluated.
2. The object denoted by the simple name is update elaborated in the context of the value of the expression.
3. The value of the expression is assigned to the object.

## Control Section Specifications

### DC_Help Specification

Elaboration of a dc_help specification consists of the following steps:

1. The through branch name implied by the dc_help specification is elaborated if it has not been previously elaborated.
2. For each scalar subelement of the through branch the corresponding scalar subelement of the contribution statement specified by the dc_help specification is associated with the net that is associated with the scalar subelement of the through branch.

### Noise Source Specification

Elaboration of a noise source specification of the first form (see Noise Source Specification) consists of the following steps:

1. The through branch name implied by the noise source specification is elaborated.
2. For each scalar subelement of the through branch the corresponding scalar subelement of the contribution statement specified by the noise source specification is marked as s noise contribution and associated with the analog net that is associated with the scalar subelement of the through branch.

Elaboration of a noise source specification of the second form consists, for each scalar subelement of the analog system variable to which the noise source specification relates, of marking the corresponding scalar subelement of the contribution statement as a noise contribution and associating it with the analog net that is associated with the scalar subelement of the analog system variable.

### Collapse Specification

Elaboration of a collapse specification consists of the following steps:

1. An implicit connection association element is created whose connection formal part F is the object denoted by the first pin aspect or simple name of the collapse specification and whose connection actual part A is the object denoted by the second pin aspect or simple name of the collapse specification.

2. If F is a pin, then the characteristic expression associated with the analog net that is associated with A is replaced by an expression that is the sum of the characteristic expression associated with the analog net associated with F and the characteristic expression associated with the analog net associated with A. The characteristic expression associated with the analog net associated with F is then deleted.
3. If F is a pin or an analog system variable, then the analog net associated with F is deleted.
4. The implicit connection association element is elaborated.

## Start Value Specification and Initial Condition Specification

Elaboration of a start value specification or an initial condition specification consists of the following steps:

1. The expression is evaluated.
2. For each scalar subelement of the expression, if the value of the scalar subelement is not equal to **undef**, then the value is associated with the corresponding scalar subelement of the analog variable to which the specification relates.

## Restart Specification

Elaboration of a restart specification has no effect.

## Nonlinearity Specification

Elaboration of a nonlinearity specification consists of the following steps:

1. Each scalar subelement of each member of the independent set is associated with each scalar subelement of each member of the dependent set.
2. If present, the limiting function denoted by the limiting function name is associated with the dependent set.
3. If present, the expression is evaluated, and its value is associated with the dependent set.

## Partial Derivative Specification

Elaboration of a partial derivative specification consists of the creation of the variable denoted by the identifier, followed by the association of the expression and the wrt name with the variable.

## Small-Signal Specification

Elaboration of a small-signal specification consists of the following steps:

1. The variable denoted by the identifier is created.
2. The report expression is evaluated.
3. The variable is decorated with the category attribute and the report attribute.
4. The expression and the wrt name, if present, are associated with the variable.

## Stress Measure Specification

Elaboration of a stress measure specification consists of the following steps:

1. The variable denoted by the identifier is created.
2. The report expression, the rating expression and the reference rating expression, if present, are evaluated.
3. The variable is decorated with the category attribute and the report attribute.
4. The expression, the simple name, the value of the rating expression and the value of the reference rating expression, if present, are associated with the variable.

## Variable Range Specification

Elaboration of a variable range specification consists of the evaluation of the min expression, the max expression, the abs expression (if present), and the rel expression (if present), followed by the association of the values of these expressions with each scalar subelement of each member of the variable set.

## Unit Range Specification

Elaboration of a unit range specification consists of the evaluation of the min expression, the max expression, the abs expression (if present), and the rel expression (if present), followed by the association of the values of these expressions with the unit denoted by the unit name.

### Range Set Specification

Elaboration of a range set specification consists of the evaluation of the expression followed by the association of the value of the expression with the instance.

### Other Control Section Specifications

Elaboration of any other control section specification consists of the evaluation of the expressions followed by the association of the values of the expressions with each entity to which the specification relates.

## Elaboration of Statements

### Statements Decorated with the Values Attribute

Elaboration of statements decorated with the **values** attribute establishes certain properties of analog variables. The statements are elaborated in two parts.

Part 1 of the elaboration of the statements decorated with the **values** attribute performs part 1 of the elaboration of each such statement in the order in which the statements appear in the template.

Part 2 of the elaboration of the statements decorated with the **values** attribute performs part 2 of the elaboration of each such statement in the order in which the statements appear in the template.

An *independent variable* is a scalar subelement of one of the following:

- an analog system variable
- a branch across variable
- an analog local variable that is the target of an assignment statement whose associated transformed expression (see Assignment Statements), after applying trivial symbolic identity transformations to remove parentheses, is either a name denoting an analog system variable or a branch across variable, optionally preceded by a unary plus or unary minus operator, or the difference of two names that each denotes an analog system variable, a branch across variable with one pin aspect denoting the reference pin 0, or an across branch name with one pin aspect denoting the reference pin 0. The assignment statement must not be followed by another assignment statement with the same target in the sequence of statements.

An analog local variable is an *output variable* if its name does not appear in an expression in a statement decorated with the **equations** attribute and if it was not substituted to create a transformed expression in part 1 of the elaboration of statements decorated with the **values** attribute, except in an expression of a statement whose target is an output variable. It is an error if a name that denotes an imported object of class analog local variable appears in the expression of an assignment statement whose target is not an output variable.

A scalar subelement of an analog local variable is a *dependent variable* if it is the target of an assignment statement whose associated partially transformed expression (see Assignment Statements) depends nonlinearly on independent variables and if its name, or a name of which the scalar subelement is a subelement, is a term in an expression, in a transform, or in an actual expression in a function call, in a statement decorated with the **equations** attribute. The assignment statement must not be followed by another assignment statement with the same target in the sequence of statements.

## Assignment Statements

Part 1 of the elaboration of an assignment statement decorated with the values attribute associates a *transformed expression* with each assignment statement. The transformed expression associated with an assignment statement A is obtained from the expression of A by textually substituting each analog local variable V of kind val that appears in the expression by the transformed expression associated with the closest assignment statement preceding A whose target is V.

Part 2 of the elaboration of an assignment statement decorated with the **values** attribute associates a *partially transformed* expression with each assignment statement. The partially transformed expression associated with an assignment statement A is obtained from the expression of A by textually substituting each analog local variable V that appears in the expression and is not an independent variable by the partially transformed expression associated with the closest assignment statement preceding A whose target is V.

## Conditional Statements

Elaboration of a conditional statement decorated with the **values** attribute whose condition is a constant expression consists of first evaluating the condition. If the value of the condition is nonzero, then the statement of the then part is elaborated. Otherwise, the statement of the else part is elaborated.

Part 1 of the elaboration of a conditional statement decorated with the **values** attribute whose condition is not a constant expression consists of part 1 of the elaboration of both the then part and the else part, followed by combining the elaborated parts, as follows:

- For each analog local variable V of kind **val** that is the target of an assignment statement in both the then part and the else part of the conditional statement, a single assignment statement is generated whose target and expression is V. Then, the expression

  if condition then then_t_expression else else_t_expression

  is associated as the transformed expression with the generated assignment statement, where condition is the condition of the conditional statement, then_t_expression is the transformed expression associated with the assignment statement whose target is V in the then part of the conditional statement, and else_t_expression is the transformed expression associated with the assignment statement whose target is V in the else part of the conditional statement.

- For each analog local variable V of kind **val** that is the target of an assignment statement in only the then part of the conditional statement, a single assignment statement is generated whose target and expression is V. Then, the expression

  if condition then then_t_expression else V

  is associated as the transformed expression with the generated assignment statement.

- For each analog local variable V of kind **val** that is the target of an assignment statement in only the else part of the conditional statement, a single assignment statement is generated whose target and expression is V. Then, the expression

  if condition then V else else_t_expression

  is associated as the transformed expression with the generated assignment statement.

Part 2 of the elaboration of a conditional statement decorated with the **values** attribute whose condition is not a constant expression consists of part 2 of the

elaboration of both the then part and the else part, followed by combining the elaborated parts, as follows:

- For each analog local variable V of kind val that is the target of an assignment statement in both the then part and the else part of the conditional statement, the expression

  if condition then then_pt_expression else else_pt_expression

  is associated as the partially transformed expression with the assignment statement generated in part 1, where condition is the condition of the conditional statement, then_pt_expression is the partially transformed expression associated with the assignment statement whose target is V in the then part of the conditional statement, and else_pt_expression is the partially transformed expression associated with the assignment statement whose target is V in the else part of the conditional statement.

- For each analog local variable V of kind val that is the target of an assignment statement in only the then part of the conditional statement, the expression

  if condition then then_pt_expression else V

  is associated as the partially transformed expression with the assignment statement generated in part 1.

- For each analog local variable V of kind val that is the target of an assignment statement in only the else part of the conditional statement, the expression

  if condition then V else else_pt_expression

  is associated as the partially transformed expression with the assignment statement generated in part 1.

**Note**

Part 1 and part 2 of the elaboration of a conditional statement decorated with the **values** attribute whose condition is a constant expression are identical.

## Compound Statements

Elaboration of a compound statement decorated with the **values** attribute consists of the elaboration of each compound sentence.

## Statements Decorated with the Control_section Attribute

Elaboration of statements decorated with the **control_section** selects certain control section specifications for elaboration. The statements are elaborated in the order in which they appear in the template.

### Conditional Statements

Elaboration of a conditional statement decorated with the **control_section** attribute consists of the following steps:

1. The condition is evaluated.
2. If the value of the condition is nonzero, then the conditional sentence of the then part is elaborated. Otherwise, if the else part is present, then the conditional sentence of the else part is elaborated.

### Compound Statements

Elaboration of a compound statement decorated with the **control_section** attribute consists of the elaboration of each compound sentence.

## Instantiation Statements

Elaboration of an instantiation statement consists of two parts.

Part 1 of the elaboration of an instantiation statement consists of the following steps:

1. Part 1 of the elaboration of the template header of the template denoted by the prefix of the instance name is performed.
2. The instance argument association list is elaborated (see Instance Argument Association Lists, Argument Association Elements).
3. Each unassociated parameter that is decorated with the external attribute is associated with a previously elaborated object (see Association of Parameters Decorated with the External Attribute).
4. Part 2 of the elaboration of the template header of the template denoted by the prefix of the instance name is performed.
5. The connection association list is elaborated (see Connection Association Lists, Connection Association Elements).

6. Each unassociated pin, analog system variable or state that is decorated with the external attribute is associated with a previously elaborated object (see Association of Other Objects Decorated with the External Attribute).

Part 2 of the elaboration of an instantiation statement consists of the elaboration of the template body of the template denoted by the prefix of the instance name.

The instance containing the instantiation statement is said to be the *parent instance* of the instance defined by the instantiation statement. The parent instance, its parent instance, the parent instance of its parent instance, etc., up to the root instance are collectively called the *ancestors* of the instance defined by the instantiation statement.

## Instance Argument Association Lists, Argument Association Elements

Elaboration of an instance argument association list consists of ordering the argument association elements, followed by the elaboration of each argument association element in the order in which they appear in the ordered list.

The argument association elements of an instance argument association list are ordered such that an argument association element whose actual expression contains a name that denotes an imported object that is associated with an object of the instance does not precede the argument association element, if any, whose formal denotes the object of the instance. It is an error if no such ordering is possible.

The elaboration of an argument association element consists of the following steps:

1. If the actual expression contains a name that denotes an imported object, and if the imported object does not yet exist, then the implicit declaration declaring the imported object is elaborated.
2. The actual expression is evaluated.
3. The formal is update elaborated in the context of the value of the expression.
4. The value of the expression is assigned to the formal.
5. If an imported object is associated with the formal, then an implicit alter specification is created that specifies the initial value of the imported object as the value of the formal. The alter specification is elaborated in turn.

**Note**

The algorithm described in this section defines a partial ordering of the argument association elements of an instance argument association list. Partial ordering is not possible if circular references exist.

## Connection Association Lists, Connection Association Elements

Elaboration of a connection association list consists of the elaboration of each connection association element in the order in which they appear.

Elaboration of a connection association element consists of the following:

- If the connection formal part denotes a pin, then the elaboration of the connection association element associates each scalar subelement F of the formal connection with the analog net that is associated with the corresponding scalar subelement A of the connection actual part. Additionally, the elaboration associates the characteristic expression

  ACROSS(A) - ACROSS(F)

  with F.

- If the connection formal part denotes an analog system variable, then the elaboration of the connection association element associates each scalar subelement F of the formal connection with the analog net that is associated with the corresponding scalar subelement A of the connection actual part. Additionally, the elaboration associates the characteristic expression

  A - F

  with F.

- If the connection formal part denotes a state, then the elaboration of the connection association element associates each scalar subelement F of the formal connection with the net associated with the corresponding scalar subelement A of the connection actual part. If F is a driving state, then the elaboration also associates the driver of each scalar subelement of F (see Drivers) with the corresponding net. It is an error if more than one driver is associated with a net and the unit of the net is not a resolved unit.

## Association of Parameters Decorated with the External Attribute

To determine the object with which an unassociated parameter decorated with the **external** attribute may be associated, the parent instance is searched for a

declaration that declares an entity with the same name in the same overloading class as the parameter. If no such declaration is found, the argument association list of the instantiation statement instantiating the parent instance is searched for an argument association element whose formal denotes an object whose name matches that of the parameter. Then, if no such argument association element exists, the search continues for a declaration or an argument association element in the parent instance of the parent instance, then in the parent instance of that instance, etc., up to the root instance, until a matching declaration is found. If a matching declaration has been found, and if the entity declared by that declaration is an object of class parameter that is unit compatible with the parameter decorated with the **external** attribute, then the elaboration creates an implicit argument association element whose formal is the name of the parameter decorated with the **external** attribute and whose actual is a primary that denotes the parameter found in an ancestor. The argument association element is elaborated in turn.

It is an error if this algorithm does not define an association for a parameter decorated with the **external** attribute.

## Association of Other Objects Decorated with the External Attribute

To determine the object with which a pin, analog system variable, or state decorated with the **external** attribute may be associated, the parent instance is searched for a declaration that declares an entity with the same name in the same overloading class as the decorated object. If no such declaration is found, the search continues in the parent instance of the parent instance, then in the parent instance of that instance, etc., up to the root instance, until a matching declaration is found. If a matching declaration has been found, and if the entity declared by that declaration is an object that is of the same object class as the decorated object and that is pin type compatible (if the decorated object is of class pin) or unit compatible (if the decorated object is of class analog system variable or state) with the decorated object, then the elaboration creates an implicit connection association element whose connection formal part is the name of the decorated object and whose connection actual part is the object found in an ancestor. The connection association element is elaborated in turn.

**Notes**

1. If no matching declaration is found, the effect is the same as that of a formal connection that is left unassociated.
2. It is a consequence of these rules that the actual associated with a formal state decorated with the external attribute must be an event-driven state.

## Dynamic Elaboration

The execution of certain statements may involve elaboration during the execution of a model.

### Function Calls

Execution of a function call involves the following steps:

1. Elaboration of the function header declarative items in the order in which they occur.
2. Association of the actual value or initial value with each formal (see Function Calls).
3. Elaboration of the function body declarative items in the order in which they occur.
4. Execution of the statements in the function body.

### Compound Statements

Execution of a compound statement involves elaboration of the compound declarative items in the order in which they occur.

### Inline Groups

Execution of an assignment statement whose target is an inline group involves the elaboration of the group declaration implied by the inline group.

# 12

# Simulation

*After a model has been elaborated, simulation may begin. This chapter describes the different simulation scenarios and the agents supporting them: the event-driven engine and the analog solver.*

## The Event-Driven Engine

The event-driven engine updates event-driven states and executes when statements that are sensitive to such states.

### Drivers

A *driver* is an ordered sequence of one or more *transactions*, each consisting of a time and a value and written as “value at time”. Each driving state in an instance has a driver for each scalar subelement. Additionally, the simulator variables with state semantics, the implicit break and halt states and the implicit state associated with each function call calling the THRESHOLD function each has a driver. The time element of a transaction specifies the time at which the scalar subelement of the state attains the value specified by the value element of the transaction.

A driver has exactly one transaction whose time element is not greater than the current simulation time. This transaction is said to be the *effective transaction* of the driver. Its value element is called the *current value* of the driver.

The initial content of a driver is defined by the initial value of the corresponding scalar subelement of the state. The driver is modified by the addition and deletion of transactions. When a new transaction is added to a driver, it is inserted into the sequence of transactions such that the time element of any transaction is greater than or equal to the time element of the preceding transaction. If a driver already contains a transaction with the same time element, then this transaction is first deleted from the driver unless it is the

effective transaction, in which case the new transaction is inserted immediately after the effective transaction. A model is erroneous if it adds a transaction to a driver and the current time is greater than the time element of the transaction. A driver is said to be cleared if all transactions except the effective transaction are deleted from the driver.

The drivers of simulator variables with state semantics are modified by the event-driven engine. The drivers of the implicit states associated with function calls calling the THRESHOLD function are modified by the analog solver. The drivers for all other states are modified as follows:

- For a scalar subelement of an event-driven state declared by a state declaration, a transaction “V at T” is added to the driver of the state by a function call calling the SCHEDULE_EVENT function whose second argument contains the state as a subelement. T is the first argument of the function call, and V is the scalar subelement of its third argument that corresponds to the scalar subelement of the state. The value returned by the function call is associated with the transaction.
- For the implicit break state, a transaction “1 at T” is added to its driver by a function call calling the SCHEDULE_NEXT_TIME function, where T is the argument of the function call. The value returned by the function call is associated with the transaction.
- For the implicit halt state, a transaction “1 at T” is added to its driver by a function call calling the HALT_SIMULATION function, where T is the argument of the function call. The value returned by the function call is associated with the transaction.
- A transaction is deleted from a driver by a function call calling the DESCHEDULE function with an argument that is associated with the transaction, unless the transaction is the effective transaction of the driver.

## Propagation of State Values

When during simulation the current time becomes equal to the time element of the transaction that follows the effective transaction of a driver, then the driver is said to be *active*. When a driver is active, the effective transaction is deleted from the driver and the next transaction becomes the effective transaction.

A state is *active* if it is a component of a net and at least one driver of the net is active.

The *driving value* of a scalar driving state is the current value of the driver of the state. The driving value of a composite driving state is the aggregate of the driving values of the scalar subelements of the state.

The *effective value* of an event-driven state is the value obtained when evaluating a primary denoting the state. The effective value of a scalar event-driven state is the value of the net of which the state is a component. The value of the net is determined as follows:

- If the net has a single driver and is not of a resolved unit, then the value of the net is the current value of that driver.
- If the net has a single driver and is of a resolved unit, or if it has multiple drivers, then the value of the net is the result returned by the resolution function of the net when called with an argument array whose length is the number of drivers of the net and whose value is the aggregate of the current values of each driver of the net in an order defined by the implementation.
- If the net has no driver, then the value of the net is the initial value of the state at the root of the net.

The effective value of a composite event-driven state is the aggregate of the effective values of the scalar subelements of the state.

When the event-driven engine updates an event-driven state it determines the driving value, if any, and the effective value of the state. Updating an event-driven state causes an *event* to occur on the state.

**Notes**

1. An assigned state cannot be an active state.
2. A state that is not a driving state does not have a driving value.

## The State Propagation Algorithm

State values are propagated by the state propagation algorithm, which consists of the following steps:

1. Update each active state in the model.
2. For each instance in the design and each when statement *W* in the order in which the when statements appear in the instance, if *W* is sensitive to any state that was updated in the previous step, and if the condition of *W* evaluates to a nonzero value, then execute the statement guarded by *W*.

# The Analog Solver

The analog solver determines analog solution points and updates the driver of the state associated with each threshold function.

## Analog Solution Points

The analog solver is said to determine the explicit characteristic expressions when it creates the characteristic expressions associated with each analog net and executes the statements decorated with the **equations** attribute in each instance of the simulatable model. The determination of the explicit characteristic expressions consists of the following steps for each instance of the model:

1. For each analog net N that is associated with a scalar subelement of a branch through variable or an unassociated analog system variable of kind var or **ref**, disassociate and delete any characteristic expression that may be associated with N, then create the characteristic expression 0 (zero) and associate it with N.
2. Unmark all scalar subelements of each branch through variable and each analog system variable of kind **var**.
3. Execute the conditional statements, labeled equation statements, and contribution statements decorated with the **equations** attribute.
4. Execute the conditional statements and assignment statements decorated with the **equations** attribute.
5. Execute the conditional statements and make statements decorated with the **equations** attribute.

It is an error if any scalar subelement of a branch through variable or an analog system variable of kind **var** is unmarked after Step 5.

## Threshold Detection

The difference of the waveform and the reference waveform of a function call calling the THRESHOLD function is said to be the *threshold expression* of the function call.

The simulatable model contains an implicit state for each function call calling the THRESHOLD function. The implicit state, whose type is

**struc** { **integer** before, after ; }

is associated with the threshold expression of the corresponding function call.

When the analog solver has determined an analog solution point at time $T_n$, it also determines the earliest time $T_t$ in the interval $[T_c, T_n)$ not including Tn such that the sign of any threshold expression changes in an interval $(T_t-\Delta,T_t]$ not including $T_t-\Delta$. It then adds the transaction "(before, after) at $T_t$" to the driver of the implicit state associated with that threshold expression, where "before" is the sign of the threshold expression at a time $T_t-\Delta$ and "after" is the sign of the threshold expression at a time $T_t+\Delta$.

**Note**

The value of $\Delta$ is not specified in this document. It may be different for different transactions.

**References**

Semi-Numerical Functions

## DC Operating Point Simulation

DC operating point simulation consists of the DC initialization phase, followed by the DC simulation cycle, followed by the DC termination phase.

### The DC Initialization Phase

The DC initialization phase consists of the following steps:

1. Set the value of the current time $T_c$ to 0. Set the value of the current frequency to 0. Set the value of the simulator variable DC_DOMAIN to 1.
2. Compute the driving value and effective value of each event-driven state. Set the current value of each event-driven state to its effective value. Set the current value of each assigned state to its initial value. Set the effective transaction of the driver of the implicit break and halt states to "0 at 0" and set their current value 0.
3. Set the value of each analog system variable to 0.
4. Add the transaction "1 at $T_c$" to the driver of the simulator variable DC_INIT, then execute the DC event cycle. $T_c$ may advance as a result.

5. If the analysis executing the DC algorithm is not a DC analysis, skip the remainder of this step. Otherwise, add the transaction "1 at $T_c$" to the driver of the simulator variable DC_START, then execute the DC event cycle. Tc may advance as a result.
6. Detect any threshold crossings, then execute the DC event cycle. Tc may advance as a result.
7. Set the value of each analog variable that appears in a **start_value** specification to the value of the expression in the **start_value** specification.

**Notes**

1. The initial value of analog local variables is undefined.
2. An implementation may support alternate means to set the current value of each state and the value of each analog system variable.

## The DC Simulation Cycle

The DC simulation cycle consists of the repeated execution of the following steps:

1. Reset $T_c$ to 0. Set the time element of the effective transaction of the driver of the implicit break and halt states to 0.
2. Determine an analog solution point using the DC equations at time $T_c$ and the current values of all states.
3. Detect any threshold crossings, then execute the DC event cycle. $T_c$ may advance as a result.
4. If the break state has not been updated in the preceding step, the DC simulation cycle ends.

**Note**

The halt state is ignored.

## The DC Termination Phase

The DC termination phase consists of the following steps:

1. Add the transaction “1 at $T_c$” to the driver of the simulator variable DC_DONE and set the current time $T_c$ to the earliest time at which any driver becomes active, then execute the state propagation algorithm and set the current value of the simulator variable DC_DONE to 0.
2. Reset $T_c$ to 0. Set the effective transaction of the driver of the implicit break and halt states to “0 at 0” and set their current value to 0.
3. Set the value of the simulator variable DC_DOMAIN to 0.

An *initial point* is a collection of values related to a design, consisting of:

- A time
- A value for each analog system variable in the design
- A value for each state in the design
- A driver for each scalar driving state in the design

The result of the execution of the DC algorithm is an initial point whose time is $T_c$.

## The DC Event Cycle

The DC event cycle consists of the repeated execution of the following steps:

1. Set the value of the next event time $T_e$ to the smaller of:
   - The value of the literal **inf**
   - The earliest time at which any driver becomes active
2. If $T_e$ = **inf** the DC event cycle ends. Otherwise, set the current time $T_c$ to $T_e$.
3. Execute the state propagation algorithm.
4. If DC_INIT = 1, set its current value to 0. If DC_START = 1, set its current value to 0.

## Time Domain Simulation

Time domain simulation consists of the time domain initialization phase, followed by the time domain simulation cycle, followed by the time domain termination phase.

### The Time Domain Initialization Phase

The time domain initialization phase consists of the following steps:

1. Set the value of the current frequency to 0. Set the value of the simulator variable TIME_DOMAIN to 1.
2. Set the value of the current time $T_c$ to the value of the time from the initial point. Set the current value of each event-driven state to its corresponding value from the initial point. Assign to the driver of each event-driven state all transactions in the corresponding driver from the initial point. Set the value of each assigned state to its corresponding value from the initial point. Set the value of each analog system variable to its corresponding value from the initial point.
3. If $T_c$ is equal to the beginning time $T_b$ of the time domain simulation, skip the remainder of this step. Otherwise, adjust the time of each transaction in each driver by adding the value $T_b$ - $T_c$ to the time of the transaction. Then, for each adjustable state in the model add the value $T_b$ - $T_c$ to the value of the state and, if applicable, to the value of each transaction in its driver. Finally, set the value of Tc to the value of Tb.
4. If the initial point has been generated by a time domain analysis, skip the remainder of this step. Otherwise, add the transaction “1 at $T_c$” to the driver of the simulator variable TIME_INIT and set the current time $T_c$ to the earliest time at which any driver becomes active, then execute the state propagation algorithm and set the current value of the simulator variable TIME_INIT to 0.
5. Add the transaction “1 at $T_c$” to the driver of the simulator variable TR_START and set the current time Tc to the earliest time at which any driver becomes active, then execute the state propagation algorithm and set the current value of the simulator variable TR_START to 0.

**Note**

The initial value of analog local variables is undefined.

## The Time Domain Simulation Cycle

The time domain simulation cycle consists of the repeated execution of the following steps:

1. If the driver of the implicit break state or the driver of the implicit halt state becomes active at the current time $T_c$, clear the driver of the simulator variable TIME_STEP_DONE.
2. If the driver of the simulator variable TIME_STEP_DONE contains only the effective transaction, determine the next continuous time Tn and add the transaction “1 at $T_n$” to the driver of the simulator variable TIME_STEP_DONE.
3. Find the earliest threshold crossings in the interval [$T_c$, $T_n$) open to the right.
4. Set the value of the current time $T_c$ to the smaller of:
   - The value of the literal **inf**.
   - The earliest time at which any driver becomes active
5. Determine an analog solution point at $T_c$.
6. Execute the state propagation algorithm.
7. Set the current value of the simulator variable TIME_STEP_DONE to 0.
8. If the value of the halt state is 1, or if the value of $T_c$ equals the value of the literal inf, the time domain simulation cycle ends.

## Time Domain Termination Phase

The time domain termination phase consists of the following steps:

1. Add the transaction “1 at $T_c$” to the driver of the simulator variable TR_DONE and set the current time Tc to the earliest time at which any driver becomes active, then execute the state propagation algorithm and set the current value of the simulator variable TR_DONE to 0.
2. Set the value of the simulator variable TIME_DOMAIN to 0.

# 13

# Lexical Elements

*The description of a design consists of one or more text files. A text file is a sequence of lexical elements, each composed of characters. This chapter describes the composition rules.*

## Character Set

The only characters allowed in the text of an MAST description are the characters of the basic character set. Each character corresponds to a unique code of the ISO 8859-1 character set. Each graphic character is visually represented by a graphical symbol.

```
graphic_character ::=
    lowercase_letter
  | uppercase_letter
  | digit
  | punctuation_character
  | special_character

basic_character ::=
    graphic_character
  | white_space_character
  | format_effector
```

The basic character set is sufficient to write any MAST description. The characters included in each of the categories are defined as follows.

lowercase_letter ::= one of
a b c d e f g h i j k l m n o p q r s t u v w x y z
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

uppercase_letter ::= one of
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ

digit ::= one of
0 1 2 3 4 5 6 7 8 9

punctuation_character ::= one of
" # % & ( ) * + , - . / : ; < = > @ [ ] \ _ { } | ~ m

special_character ::= one of
` ! $ ' ? ^ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ – ® ¯ ° ± ² ³ ´ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿

white_space_character ::= one of
SP NBSP HT

format_effector ::= one of
VT CR LF FF

**Notes**

1. Characters that have no graphical symbol are represented by their ISO 8859-1 abbreviation. Their full ISO 8859-1 names are:

| | | | |
|---|---|---|---|
| SP | space | VT | vertical tabulation |
| NBSP | nonbreaking space | CR | carriage return |
| HT | horizontal tabulation | LF | line feed |
| | | FF | form feed |

2. The line feed character (LF) is sometimes called newline and abbreviated as NL.

**Commentary**

Of the ISO 8859-1 character set only the graphic characters, the white space characters and the format effectors may appear in a text written in the MAST language. Letters, digits, punctuation characters and white space characters

are needed to form valid statements in the language. Special characters may only appear in string literals. White space characters and format effectors may be used to lay out the text of an MAST description in a way that is pleasing to a human reader. Unescaped format effectors also mark the end of a sentence.

## Lexical Elements and Separators

A *lexical element* is either a delimiter, an identifier (which may be a keyword), a numeric literal, a string literal, a special reference designator, or a file name literal.

In some cases adjacent lexical elements must be separated by an explicit *separator*. A separator is either a format effector, or a semicolon (;) or white space character that is not part of a string literal or a comment.

A format effector or a CR character immediately followed by a LF character marks the end of a line.

Any number of separators are allowed between adjacent lexical elements, before the first lexical element in a text file, or after the last lexical element in a text file. At least one separator is required between an identifier, a literal or a special reference designator and an adjacent identifier or literal.

**Example**

```
val nu foo        # space characters separate
                  # adjacent lexical elements
```

**Note**

Each lexical element must fit on one line since the end of a line is a separator.

## Delimiters

A *delimiter* is either one of the following punctuation characters from the basic character set:

```
( ) [ ] { } , . = < > + - * / % & | : ~ \
```

or one of the following compound delimiters composed of two adjacent punctuation characters:

```
== ~= <= >= += -= -> <- ** // ..
```

Each of the single characters listed in this section is a single delimiter except if it is part of a compound delimiter, a numeric literal, a string literal, or a comment. Each of the character combinations listed in this section is a single delimiter except if the characters are part of a string literal or a comment.

**Examples**

```
a=SQRT(b)**3          # a,=,SQRT,(,b,),**,3 are lexical
                      # elements, =,(,),** are delimiters

a = SQRT ( b ) ** 3   # the same statement, but with
                      # separators to improve readability
```

## Identifiers

*Identifiers* are used as names and as reserved words.

identifier ::= basic_identifier | extended_identifier

### Basic Identifiers

A basic identifier consists only of letters, digits, and underlines.

basic_identifier ::=
letter_or_underline *{* letter_or_digit_or_underline *}*

letter_or_underline ::=
lowercase_letter | uppercase_letter | _

letter_or_digit_or_underline ::=
lowercase_letter | uppercase_letter | digit | _

Basic identifiers are case insensitive, and all characters are significant.

### Examples

```
clock_rate     N573UA       _off         _1           A__b
```

The following identifiers refer to the same name:

```
MAST           mast         Mast
```

## Extended Identifiers

Extended identifiers may contain any graphic character or white space character.

extended_identifier ::=
@ " graphic_or_space_character *{* graphic_or_space_character *}* "

graphic_or_space_character ::=
graphic_character | white_space_character

If a quotation mark (") is to be used as one of the graphic characters in an extended identifier, it must be doubled. Extended identifiers are case sensitive, and all characters are significant (a double quotation mark counting as one character). An extended identifier is distinct from any basic identifier.

### Examples

```
@"a and b"     @"i#523"     @"n$3.5"     @"<=>"
```

The following extended identifiers are all different and different from the identifier used in the example in Basic Identifiers:

```
@"MAST"        @"mast"      @"Mast"
```

# Numeric Literals

*Numeric literals* specify integer values and real values.

numeric_literal ::=
integer_literal | real_literal

## Integer Literals

An *integer literal* specifies an integral value in decimal, octal or hexadecimal notation.

integer_literal ::=
decimal_literal | octal_literal | hex_literal

decimal_literal ::=
digit *{* digit *}*

octal_literal ::=
0 octal_mark digit *{* digit *}*

hex_literal ::=
0 hex_mark hex_digit *{* hex_digit *}*

octal_mark ::= o | O

hex_mark ::= x | X

hex_digit ::= digit | a | b | c | d | e | f | A | B | C | D | E | F

In a hexadecimal literal the characters a through f (or A through F) represent the values 10 through 15. It is an error if one of the digits 8 or 9 appears in an octal literal.

**Examples**

```
1            15573        0o777        0x1ab
```

The following integer literals have the same value:

```
59           0o73         0x3b
```

**Note**

The octal mark is the uppercase or lowercase letter O.

### Rationale

Octal literals are prefixed with the sequence 0o, not just a 0 as in the C language, to support the common practice of entering a literal by appending appropriate digits to a default of 0 in schematic capture systems.

## Real Literals

A *real literal* specifies a floating point value.

real_literal ::=
  decimal_literal *[* . decimal_literal *] [* exponent *]*
| . decimal_literal *[* exponent *]*

exponent ::=
  exponent_mark *[* + *]* decimal_literal
| exponent_mark - decimal_literal
| scale_factor

exponent_mark ::= e | d | E | D

The scale factors and their interpretation are as follows. Scale factors are case insensitive.

| | | |
|---|---|---|
| a | atto | $10^{-18}$ |
| f | femto | $10^{-15}$ |
| p | pico | $10^{-12}$ |
| n | nano | $10^{-9}$ |
| u \| mu \| μ | micro | $10^{-6}$ |
| m | milli | $10^{-3}$ |
| k | kilo | $10^{3}$ |
| meg \| me | mega | $10^{6}$ |
| g | giga | $10^{9}$ |

| | | |
|---|---|---|
| t | tera | $10^{12}$ |

### Examples

```
1           1.5          .2          1e5          1.2d-6

1k          4.7p         1.2m        10Meg
```

The following real literals have the same value:

```
3n          .003u        3000p       0.3e-8       3D-9
```

The following forms are not real literals:

```
3 meg       # no space allowed

1.2e5meg    # cannot have both numeric exponent and
            # scale factor
```

### Notes

1. A real literal is syntactically a superset of a decimal literal.
2. m means $10^{-3}$, mu means $10^{-6}$, me or meg mean $10^{6}$.

## String Literals

A *string literal* consists of a (possibly empty) sequence of graphic characters or white space characters bracketed by a pair of quotation marks.

string_literal ::= " *{* graphic_or_space_character *}* "

If a quotation mark (") is to be used as one of the graphic characters in a string literal, it must be doubled. The value of a string literal is the sequence of characters between the bracketing quotation marks, with any pair of adjacent quotation marks counted as a single character. Its length is equal to the number of characters that form its value.

### Examples

```
"string""rise time"

"!$'?^¡¢£¤¥ may appear in a string"

""     # the empty string

""""   # a string containing a single quotation mark
```

### Notes

1. A string literal must fit on one line. Longer strings can be created by the concatenation of string literals.
2. Certain characters and character combinations have a special meaning in a string value used as a format specifier in a message function.

### References

Messages, Additive Operators

### Rationale

A quotation mark (") is entered into a string literal by doubling it. An alternative could be to escape the quotation mark with a backslash (\), but this would require escaping the backslash itself in a string literal. Backward compatibility rules out this alternative because the character combinations \\, \n, \t, and \% have a special meaning as format specifiers in message functions.

The MAST definition is the converse of the corresponding definition in the C language. In C, special characters are escaped with a backslash in the string literal, and in a format specification the percent character (%) must be doubled to print a single percent character.

## Special Reference Designators

A special reference designator consists of letters, digits and underlines.

special_reference_designator ::=
digit *{* letter_or_digit_or_underline *}*

Special reference designators are case insensitive, and all characters are significant.

### Examples

```
1            1a           1d5          2n3055       7_11
```

### Note

Some numeric literals also satisfy the syntax rules for a special reference designator.

## Comments

A number sign (#) that is not part of a string literal marks the beginning of a *comment*. The comment ends immediately before the format effector marking the end of the line. A comment can appear on any line of a MAST description. The absence or presence of comments has no influence on whether a description is legal or not.

## Examples

```
# This is a comment that spans the entire line

a = 5  # this comment follows a MAST statement
       # and is split over two lines
```

## Keywords

A *keyword* is an identifier that has a special meaning in the language. Some keywords are *reserved words*; they must not be used as the name of a declared identifier.

The following identifiers are reserved words.

| | | |
|---|---|---|
| alter | for | state |
| branch | foreign | states |
| component | function | string |
| conflict_resolution | if | struc |

| | | |
|---|---|---|
| const | inf | template |
| control_section | integer | then |
| element | make | undef |
| else | next | union |
| encrypted | number | unit |
| enum | parameter | val |
| equations | parameters | values |
| exit | pin | variable |
| export | return | when |
| external | simvar | while |

The following identifiers are keywords but their use is not reserved.

| | | |
|---|---|---|
| across | input | ref |
| adjust_on_restart | newton_step | sample_points |
| collapse | noise_source | small_signal |
| dc_help | output | ss_partial |
| device_type | pl_set | start_value |
| group | range_for_unit | stress_measure |
| initial_condition | range_for_variable | through |
| inout | range_set | var |

## Sentence Termination

A *sentence* consists of a sequence of lexical elements terminated by an eos (end of sentence) mark.

eos ::= format_effector | ;

A format effector ends a sentence only if the immediately preceding lexical element is not one of the following delimiters.

( [ { , . = < > + - * / % & | : ~ \ == ~= <= >= += -= -> <- ** // ..

Such a delimiter *extends* the sentence such that the sentence includes at least the next lexical element from the text file. A backslash (\) used to extend a sentence is ignored in the syntax of the sentence. It is an error if no lexical element follows a delimiter that extends a sentence.

**Examples**

```
a = 5 ; b = 6      # two sentences on one line, the first
                   # terminated with a semicolon, the
                   # second with a format effector

number c = 5,      # no eos here because of trailing ","
   d = 6, e = 6

number f = 5,      # no eos here because of backslash
              g    # no eos here because line contains no
= 7                # lexical element
```

**Note**

Since a comment ends immediately before the end of a line but is not a lexical element, a backslash used to escape the end of the line must be the lexical element immediately preceding the comment.

## File Inclusion

A line whose first character is the less-than sign (<) specifies file inclusion.

file_inclusion ::=
< file_name_literal

file_name_literal ::=
graphic_character *{* graphic_character *}*

A file name literal cannot contain the number sign (#).

If the file name literal contains neither a solidus (/) nor a backslash (\) character, then it specifies the name of the file to be included. Otherwise, the name of the file is specified by the sequence of characters after the last character that is either a solidus or a backslash, and the sequence of characters preceding the file name specify a hierarchical path whose path components must be separated by either a solidus or a backslash.

The file name literal must specify the location of the file to be included either completely or in conjunction with the data search path of the simulator.

The sequence of lexical elements in the file replaces the line specifying the file inclusion.

### Example

The line

```
<consts.sin       # Get constants from consts.sin
```

specifies that the file consts.sin be found in the simulator's data search path. Its content replaces the line.

### Notes

1. The less-than sign must be the first character on a line and cannot be preceded by white space characters.
2. A file inclusion cannot extend to the following line because it is not an MAST sentence.
3. The graphic characters that may appear in a file name literal may be restricted by the operating system.
4. Whether a file name literal is case sensitive or not depends on the operating system.
5. The definition of the simulator data search path is not part of the MAST language.

# 14

# Predefined Language Environment

*This chapter describes predefined common types, units, pin types, pins, variables, transforms, and functions.*

## Predefined Common Types

```
INTEGER

NUMBER

STRING

struc BREAKPOINT { NUMBER BP, INC ; }

enum STRESS_MEASURES \
   { PEAK, MAX, MIN, AVERAGE, RMS, WINMAX, WINMIN }
```

## Predefined Units

```
unit { "across", "across", "generic_potential" } ACROSS

unit { "through", "through", "generic_flow" } THROUGH
```

## Predefined Pin Types

```
pin NEUTRAL across ACROSS through THROUGH
```

## Predefined Pins

```
NEUTRAL 0
```

## Simulator Variables

Simulator variables provide a means of communication between a model and the simulator kernel. The name of a simulator variable refers to a matching variable in the simulator kernel with semantics that depend on the particular simulator variable.

### Simulator Variables with Function Semantics

Simulator variables with function semantics have a single matching variable in the simulator kernel. Their semantics are those of a function call calling an impure function (except as noted) that returns the value of the matching variable in the simulator kernel.

DC_DOMAIN

| | |
|---|---|
| Description: | Indicates whether the model is being executed by a DC analysis |
| Type: | INTEGER |
| Semantics: | Defined in DC Operating Point Simulation |

TIME_DOMAIN

| | |
|---|---|
| Description: | Indicates whether the model is being executed by a time domain analysis |
| Type: | INTEGER |
| Semantics: | Defined in Time Domain Simulation |

FREQ_DOMAIN

Description: Indicates whether the model is being executed by a frequency domain analysis

Type: INTEGER

Semantics:

TIME

Description: If FREQ_DOMAIN is 1, returns the time at which the small-signal model has been determined. Otherwise, returns the current simulation time $T_c$

Type: NUMBER

Semantics: Defined in DC Operating Point Simulation, Time Domain Simulation

FREQ

Description: Returns the current simulation frequency

Type: NUMBER

Semantics:

FREQ_MAG

Description: Indicates whether the simulator is determining the magnitude value

Type: INTEGER

Semantics:

FREQ_PHASE

Description: Indicates whether the simulator is determining the phase value

Type: INTEGER

Semantics:

STATISTICAL

Description: Indicates whether the simulator is performing a statistical analysis

Type: INTEGER

Semantics:

Note: The semantics of the STATISTICAL simulator variable are those of a pure function.

WORST_CASE

Description: Indicates whether the simulator is performing a worst case statistical analysis

Type: INTEGER

Semantics:

Note: The semantics of the WORST_CASE simulator variable are those of a pure function.

## Simulator Variables with State Semantics

Simulator variables with state semantics have a single matching variable in the simulator kernel. Their driver is owned by the simulator kernel. Their value is the value of the matching variable in the simulator kernel.

DC_INIT

Description: Indicates whether the model is being executed as it enters the DC domain

Type: INTEGER

Semantics: Defined in The DC Initialization Phase and The DC Event Cycle

DC_START

Description: Indicates whether the model is being executed at the beginning of a DC analysis

Type: INTEGER

Semantics: Defined in The DC Initialization Phase and The DC Event Cycle

DC_DONE

Description: Indicates whether the model is being executed at the end of a DC analysis

Type: INTEGER

Semantics: Defined in The DC Termination Phase

TIME_INIT

Description: Indicates whether the model is being executed as it enters the time domain

Type: INTEGER

Semantics: Defined in The Time Domain Initialization Phase

TR_START

Description: Indicates whether the model is being executed at the beginning of a time domain analysis

Type: INTEGER

Semantics: Defined in The Time Domain Initialization Phase

TR_DONE

Description: Indicates whether the model is being executed at the end of a time domain analysis

Type: INTEGER

Semantics: Defined in Time Domain Termination Phase

TIME_STEP_DONE

Description: Indicates whether the model has just completed an analog solution point

Type: INTEGER

Semantics: Defined in The Time Domain Simulation Cycle

## Simulator Variables with Analog Local Variable Semantics

Simulator variables with semantics of an analog local variable have a matching variable in the simulator kernel for each instance of any model in the design. They may be assigned a value in an instance; the corresponding statement must be decorated with the **values** or **equations** attribute. It is an error if the name of a simulator variable with analog local variable semantics appears in an expression.

NEXT_TIME

Description: Specifies the time at which the value of each analog variable in the instance must be determined at the latest

Type: NUMBER

Semantics: Defined in

STEP_SIZE

| | |
|---|---|
| Description: | Specifies the time interval within which the value of each analog variable in the instance must be determined |
| Type: | NUMBER |
| Semantics: | Defined in |

## Transforms

The statements containing the transforms described in this section must be decorated with the **equations** attribute.

D_BY_DT ( waveform )

| | |
|---|---|
| Parameters: | waveform: A simple expression whose scalar subelements are of type NUMBER |
| Result type: | The type of the expression |
| Result: | The derivative of the waveform with respect to time |

DELAY ( waveform, T )

| | |
|---|---|
| Parameters: | waveform: A simple expression whose scalar subelements are of type NUMBER |
| | T: A globally constant expression of a numeric type that evaluates to a nonnegative value |
| Result type: | The type of the expression |
| Result: | The waveform delayed by T |

TRANSFER_FUNCTION ( waveform, numerator, denominator )

| | |
|---|---|
| Parameters: | waveform: A simple expression of type NUMBER |
| | numerator: A globally constant one-dimensional array whose element type is a numeric type |

denominator: A globally constant one-dimensional array whose element type is a numeric type

Result type: NUMBER

Result: The Laplace transfer function of the waveform. In the frequency domain:where num is an array with the same

$$\frac{\sum_{i=0}^{LEN(num)-1} num[LEN(num)-i]\cdot s^{i}}{\sum_{i=0}^{LEN(den)-1} den[LEN(den)-i]\cdot s^{i}}\cdot waveform$$

elements as numerator but with a normalized index range, den is an array with the same elements as denominator but with a normalized index range, and s is the Laplace variable

## Functions

The arguments of the predefined functions are unnamed. Unless noted, the functions are pure functions.

## Nonmathematical Functions

ADDR ( *composite*_name )

| | |
|---|---|
| Parameters: | *composite*_name: A name denoting an object of a composite common type |
| Result type: | INTEGER |
| Result: | A handle for the memory address of the object in the simulator data structures |
| Restrictions: | The semantics of the value returned by the addr() function are not specified in this document<br><br>The layout of the simulator data structures is not specified in this document<br><br>A model is erroneous if it changes any value in the simulator data structures through this interface in a sentence that is not decorated with the **parameters** attribute |

DESIGN_NAME ( )

| | |
|---|---|
| Result type: | STRING |
| Result: | A name given to the root instance by the implementation |

INSTANCE ( )

| | |
|---|---|
| Result type: | STRING |
| Result: | The hierarchical name of the instance containing the function call that calls INSTANCE ( ) |
| Notes: | The result has the following format:<br>/ [ instance_name { / instance_name } ]<br><br>When called in a function F, the INSTANCE function returns the hierarchical name of the instance calling F |

LEN ( *array*_expression *[* , index_position *]* )

Parameters: *array*_expression: A simple expression of an array type

index_position: A simple expression of type INTEGER whose value is between 1 and the dimensionality of the array expression

Result type: INTEGER

Result: If the index position is present, the length of the index range indicated by the index position. Otherwise, the number of elements in the array expression.

LEN ( *string*_expression )

Parameters: *string*_expression: A simple expression of type STRING

Result type: INTEGER

Result: The length of the string expression. **undef** if the value of the string expression equals the value of the literal **undef**.

UNION_TYPE ( *union*_name , alternative )

Parameters: *union*_name: The name of an object of a union type

alternative: A simple name denoting an alternative of the union object

Result type: INTEGER

Result: 1 if the specified alternative is the current alternative of the union object, 0 otherwise

## Functions Supporting Event-Driven Simulation

SCHEDULE_EVENT ( time, *state*_name, value )

| | |
|---|---|
| Parameters: | time: A simple expression of type NUMBER<br><br>*state*_name: A name denoting a state. The state denoted by its longest constant prefix is a *driving state*<br><br>value: A simple expression that is type compatible with the type of the state object |
| Description: | Adds the transaction “V at time” to the driver of each scalar subelement of the state denoted by the name, where V is the corresponding scalar subelement of the value |
| Result type: | INTEGER array of length 2 |
| Result: | The handle associated with the transactions |
| Restrictions: | The statement containing a function call to SCHEDULE_EVENT must be decorated with the states attribute. |
| Note: | The SCHEDULE_EVENT function is an impure function. |

SCHEDULE_NEXT_TIME ( time )

| | |
|---|---|
| Parameters: | time: A simple expression of type NUMBER |
| Description: | Adds the transaction “1 at time” to the driver of the implicit break state |
| Result type: | INTEGER array of length 2 |
| Result: | The handle associated with the transaction |
| Restrictions: | The statement containing a function call to SCHEDULE_NEXT_TIME must be decorated with the states attribute. |

| | |
|---|---|
| Note: | The SCHEDULE_NEXT_TIME function is an impure function. |

HALT_SIMULATION ( time )

| | |
|---|---|
| Parameters: | time: A simple expression of type NUMBER |
| Description: | Adds the transaction “1 at time” to the driver of the implicit halt state |
| Result type: | INTEGER array of length 2 |
| Result: | The handle associated with the transaction |
| Restrictions: | The statement containing a function call to HALT_SIMULATION must be decorated with the states attribute. |
| Note: | The HALT_SIMULATION function is an impure function. |

DESCHEDULE ( handle )

| | |
|---|---|
| Parameters: | handle: The handle of a transaction |
| Description: | Deletes the transactions associated with the handle from the corresponding drivers |
| Result type: | INTEGER |
| Result: | 1 if the handle is invalid, 0 otherwise |
| Restrictions: | The handle must be the result of a function call calling one of the functions SCHEDULE_EVENT, SCHEDULE_NEXT_TIME, or HALT_SIMULATION in the same major declarative region.<br><br>The statement containing a function call to DESCHEDULE must be decorated with the states attribute. |
| Note: | The DESCHEDULE function is an impure function. |

EVENT_ON ( *state*_name *[* , last value *]* )

Parameters: *state*_name: A constant name denoting a state. The state is an *observed state*.

lastvalue: A constant name denoting an assigned state that is type compatible with the type of the object denoted by the state name.

Description: Monitors the state S denoted by the state name until an event occurs on S, then assigns the previous value of S to the state denoted by lastvalue.

Result type: INTEGER

Result: 1 for the duration of Step 2 of the state update algorithm (see The State Propagation Algorithm) if the state object was updated in Step 1, 0 otherwise

Restrictions: The statement containing a function call to EVENT_ON must be decorated with the **states** attribute.

Note: The EVENT_ON function is an impure function.

THRESHOLD ( waveform, reference_waveform *[* , before *[* , after *] ]* )

Parameters: waveform: A simple expression of type NUMBER

reference_waveform: A simple expression of type NUMBER

before: A constant name denoting an assigned state of a numeric type

after: A constant name denoting an assigned state of a numeric type

| | |
|---|---|
| Description: | Monitors the implicit state S associated with the THRESHOLD function until an event occurs on S, then assigns the value S->before to the state denoted by before and the value S->after to the state denoted by after. The assignment occurs immediately after a function call calling the THRESHOLD function returns the value 1. |
| Result type: | INTEGER |
| Result: | 1 for the duration of Step 2 of the state update algorithm (see The State Propagation Algorithm) if the implicit state S associated with the THRESHOLD function was updated in Step 1, 0 otherwise |
| Restrictions: | At least one name denoting an analog variable must appear in the waveform expression or the reference waveform expression. Any such name must be a constant name.<br><br>The statement containing a function call to THRESHOLD must be decorated with the **states** attribute. |
| Note: | The THRESHOLD function is an impure function. |

DRIVEN ( *state*_name )

| | |
|---|---|
| Parameters: | *state*_name: A constant name denoting a driving state |
| Result type: | The type of the state object denoted by the state name |
| Result: | The driving value of the state object |
| Restrictions: | The statement containing a function call to DRIVEN must be decorated with the **states**, **values**, or **equations** attribute. |
| Note: | The DRIVEN function is an impure function. |

LAST_VALUE ( *state*_name )

Parameters: state_name: A constant name denoting an event-driven state

Result type: The type of the state object denoted by the state name

Result: The value of the state object denoted by the state name prior to the last event of the state object. If no event has occurred on a scalar subelement of the state, then the corresponding scalar subelement of the value is **undef**.

Restrictions: The statement containing a function call to LAST_VALUE must be decorated with the **states**, **values**, or **equations** attribute.

Note: The LAST_VALUE function is an impure function.

RAMP ( *state*_name *[* , rise_time *[* , fall_time *] ]* )

Parameters: *state*_name: A constant name denoting an event-driven state whose scalar subelements are of a floating point type

rise_time: A globally constant expression of a numeric type that evaluates to a nonnegative value. Default: 0

fall_time: A globally constant expression of a numeric type that evaluates to a nonnegative value. Default: rise_time

Result type: The type of the state object

Result: A waveform where each scalar subelement W follows the corresponding scalar subelement S of the state object. When an event occurs on S, W ramps linearly from the last value of S to the current value of S.The duration of the ramp is the value of rise_time if S - last_value(S) > 0 and the value of fall_time otherwise.

Restrictions: The statement containing a function call to RAMP must be decorated with the **values** or **equations** attribute.

Note: The RAMP function is an impure function.

SLEW (*state*_name *[* , rising_slope *[* , falling_slope *] ]* )

Parameters: *state*_name: A constant name denoting an event-driven state whose scalar subelements are of a floating point type

rising_slope: A globally constant expression of a numeric type that evaluates to a positive value. Default: **inf**

falling_slope: A globally constant expression of a numeric type that evaluates to a negative value. Default: - rising_slope

Result type: The type of the state object

Result: A waveform where each scalar subelement W follows the corresponding scalar subelement S of the state object. When an event occurs on S, W ramps linearly from the last value of S to the current value of S.The slope of the ramp is the value of rising_slope if S - last_value(S) > 0 and the value of falling_slope otherwise.

Restrictions: The statement containing a function call to SLEW must be decorated with the **values** or **equations** attribute.

Note: The SLEW function is an impure function.

## Messages

Message functions write formatted text to the standard error stream. Each message raises a condition in the simulator that indicates the severity of the message.

MESSAGE ( format *{* , value *}* )

Parameters: format: A simple expression of type STRING

value: A simple expression

Description: Raises a NOTE condition and writes the values to the standard error stream under the control of the format string

Result type: INTEGER

Result: 0

ERROR ( format *{* , value *}* )

Parameters: format: A simple expression of type STRING

value: A simple expression

Description: Raises an ERROR condition and writes the values to the standard error stream under the control of the format string, tagged as a TEMPLATE_ERROR

Result type: INTEGER

Result: 0

WARNING ( format *{* , value *}* )

Parameters: format: A simple expression of type STRING

value: A simple expression

Description: Raises a WARNING condition and writes the values to the standard error stream under the control of the format string, tagged as a TEMPLATE_WARNING

Result type: INTEGER

Result: 0

SABER_MESSAGE ( tag, *{* , value *}* )

Parameters: tag: A simple expression of type STRING

value: A simple expression

| | |
|---|---|
| Description: | Raises the condition indicated by the tag and writes the values to the standard error stream under the control of the message with the specified tag from the Saber simulator message catalog used as the format string |
| Result type: | INTEGER |
| Result: | 0 |

## Format Strings

The message functions use the format string to control writing the values in their argument list. They give the following interpretation to the characters in the format string:

- A percent (%) character consumes the next value from the argument list. The value is converted to a textual representation that is legal for its type, and the textual representation is written literally instead of the percent sign. If no more values are available in the argument list the percent character is written literally. It is an error if the type of the value cannot be determined.
- A backslash (\) character is skipped and the character following the backslash character, if any, is written literally, with two exceptions:
- The character “n” following the backslash character ends the line currently being written and begins a new line of text
- The character “t” following the backslash character is replaced by a HT character that is written literally
- Any other character is written literally.

The end of the format string ends the line currently being written. Any values remaining in the list of values are ignored.

### Example

```
struc { number n=5; string s="abc" ; } p=()
message("p=%; its elements are n=% and s=%",
       p, p->n, p->s)
```

produces the output

```
p=(n=5,s="abc"); its elements are n=5 and s="abc"
```

## Mathematical Functions

Mathematical functions return a result of a numeric type. Their arguments must be simple expressions of a numeric type.

## Trigonometric Functions

ACOS ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The principal value of the arc cosine of x in the range [0, $\pi$]

Restrictions: $-1 \leq x \leq 1$

ASIN ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The principal value of the arc sine of x in the range [$-\pi/2$, $\pi/2$]

Restrictions: $-1 \leq x \leq 1$

ATAN ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The principal value of the arc tangent of x in the range [$-\pi/2$, $\pi/2$]

ATAN2 ( x, y)

Parameters: x: A simple expression of a numeric type

y: A simple expression of a numeric type

Result type: NUMBER

| | |
|---|---|
| Result: | The principal value of the arc tangent of y/x in the range $[-\pi, \pi]$, using the sign of x and y to determine the quadrant of the result |
| Restrictions: | x and y must not be both 0 |

COS ( x )

| | |
|---|---|
| Parameters: | x: A simple expression of a numeric type |
| Result type: | NUMBER |
| Result: | The cosine of x specified in radians |

SIN ( x )

| | |
|---|---|
| Parameters: | x: A simple expression of a numeric type |
| Result type: | NUMBER |
| Result: | The sine of x specified in radians |

TAN ( x )

| | |
|---|---|
| Parameters: | x: A simple expression of a numeric type |
| Result type: | NUMBER |
| Result: | The tangent of x specified in radians |
| Restrictions: | $\|x\| \neq k \cdot (\pi/2)$ where k is any odd integer |

## Hyperbolic and Inverse Hyperbolic Functions

ACOSH ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The inverse hyperbolic cosine of x

Restrictions: $x \geq 1$

ASINH ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The inverse hyperbolic sine of x

ATANH ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The inverse hyperbolic tangent of x

Restrictions: $-1 \leq x \leq 1$

COSH ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The hyperbolic cosine of x

SINH ( x )

Parameters: x: A simple expression of a numeric type

| | |
|---|---|
| Result type: | NUMBER |
| Result: | The hyperbolic sine of x |

TANH ( x )

| | |
|---|---|
| Parameters: | x: A simple expression of a numeric type |
| Result type: | NUMBER |
| Result: | The hyperbolic tangent of x |

## Logarithmic, Exponential and Related Functions

LN ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The natural logarithm of x (base e)

Restrictions: x > 0

LOG ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The decimal logarithm of x (base 10)

Restrictions: x > 0

EXP ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The exponential function of x

LIMEXP ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: For x < 0:1 / limexp(-x)
For 0 £ x £ 80:exp(x)
For 80 < x £ 88:exp(80)Þ(x-79)
For 88 < x £ 88.7:exp(80)Þ(88-79)Þ(1+1e-6Þ(x-88))
For x > 88.7:exp(80)Þ(88-79)Þ(1+1e-6Þ(88.7-88))

SQRT ( x )

Parameters: x: A simple expression of a numeric type

Result type: NUMBER

Result: The nonnegative square root of x

Restrictions: x Š 0

## Semi-Numerical Functions

ABS ( x )

Parameters: x: A simple expression of a numeric type

Result type: The type of x

Result: The absolute value of x

INT ( x )

Parameters: x: A simple expression of a numeric type

Result type: INTEGER

Result: The integral part of x as defined in Type Conversions

SIGN ( x )

Parameters: x: A simple expression of a numeric type

Result type: INTEGER

Result: -1 if x < 0, 0 if x == 0, +1 if x > 0

MAX ( x { , x } )

Parameters: x: An arbitrary number of simple expressions of a numeric type

| | |
|---|---|
| Result type: | The supertype implied by the types of the expressions |
| Result: | The largest value of any expression x |

MIN ( x { , x } )

| | |
|---|---|
| Parameters: | x: An arbitrary number of simple expressions of a numeric type |
| Result type: | The supertype implied by the types of the expressions |
| Result: | The smallest value of any expression x |

RANDOM ( )

| | |
|---|---|
| Result type: | NUMBER |
| Result: | The next value from a pseudo-random sequence |
| Notes: | The RANDOM function is an impure function |
| | The MAST language does not provide a means to set a random number seed |

# 15

# Syntax Summary

*This chapter summarizes the syntax of the MAST language.*

## MAST **Syntax**

across_aspect ::= [Scalar Pin Types]
**across** *physical_unit*_name

actual ::= [Argument Association Lists]
expression

additive_expression ::= [Expressions]
term *{* additive_operator term *}*

additive_operator ::= + | - | // [Additive Operators]

aggregate ::= [Aggregates]
array_aggregate
| structure_aggregate
| union_aggregate

aggregate_association_element ::= [Structure Aggregates]
*[* simple_name = *]* expression

aggregate_association_list ::= [Structure Aggregates]
aggregate_association_element *{* , aggregate_association_element *}*

alter_specification ::= [Alter Specification]
**alter** specifier_list eos

analog_attribute ::= [Analog Variable Declarations]
**export**
| **external**

analog_name ::= [Start Value Specification]
simple_name
| branch_name

analog_variable_declaration ::= [Analog Variable Declarations]
var_declaration
| ref_declaration
| val_declaration
| branch_variable_declaration

argument_association_element ::= [Argument Association Lists]

*[* formal = *]* actual

argument_association_list ::= [Argument Association Lists]
argument_association_element *{* , argument_association_element *}*

argument_list ::= [Argument Lists]
identifier_list

array_aggregate ::= [Array Aggregates]
"[" expression_list "]"

assignment ::= [Loop Statement]
simple_name = expression

assignment_statement ::= [Assignment Statement]
*[* statement_attribute *] [* target = *]* expression eos

basic_character ::= [Character Set]
graphic_character
| white_space_character
| format_effector

basic_identifier ::= [Basic Identifiers]
letter_or_underline { letter_or_digit_or_underline *}*

branch_definition ::= [Branch Variable Declarations]
identifier = branch_name

branch_name ::= [Branch Names]
unit_aspect ( *plus*_pin_aspect
*[* branch_name_separator *minus*_pin_aspect ] )

branch_name_separator ::= , | -> [Branch Names]

branch_variable_declaration ::= [Branch Variable Declarations]
*{* analog_attribute *}* **branch** branch_definition
*{* , branch_definition *}* eos

collapse_specification ::= [Collapse Specification]
*[* **control_section** *]*
**collapse** ( *plus*_pin_aspect, *minus*_pin_aspect ) eos
| *[* **control_section** *]* **collapse** ( simple_name, simple_name ) eos

compilation_unit ::= [Compilation Units]
*{* named_library_unit *} [* root_template *]*

composite_type_definition ::= [Composite Common Types]
| structure_type_definition
| union_type_definition

compound_declarative_item ::= [Compound Statement]
type_declaration
| unit_declaration
| variable_declaration
| simulator_variable_declaration
| group_declaration
| control_section_specification

compound_sentence ::= [Compound Statement]
compound_declarative_item
| statement
| eos

compound_statement ::= [Compound Statement]
*[* statement_attribute *]* "{" *{* compound_sentence *}* "}" eos

condition ::= [Conditional Expressions]
expression

conditional_expression ::= [Conditional Expressions]
**if** condition **then** expression **else** expression

conditional_sentence ::= [Conditional Statement]
statement
| control_section_specification

conditional_statement ::= [Conditional Statement]
*[* statement_attribute *]* **if** ( condition ) *{* format_effector *}*
then_part [ else_part *]*

connection_actual_part ::= extended_name [Instantiation Statement]

connection_association_element ::= [Instantiation Statement]
*[* connection_formal_part : *]* connection_actual_part

connection_association_list ::= [Instantiation Statement]

*{* connection_association_element *}*

connection_definition ::= [Template Connections]
identifier
| decimal_literal

connection_element ::= [Template Connections]
*external*_connection_definition *[* : *internal*_connection_specification *]*

connection_formal_part ::= extended_name [Instantiation Statement]

connection_list ::= [Template Connections]
connection_element *{ [* , *]* connection_element *}*

connection_specification ::= [Template Connections]
identifier
| decimal_literal

context ::= [Contexts]
*{* context_sentence *}*

context_declarative_item ::= [Contexts]
function_declaration
| type_declaration
| unit_declaration
| pin_type_declaration
| parameter_declaration
| state_declaration
| ref_declaration
| pin_declaration
| template_definition

context_sentence ::= [Contexts]
context_declarative_item
| eos

contribution_op ::= += | -= [Contribution Statement]

contribution_statement ::= [Contribution Statement]
*[* statement_attribute *]* name contribution_op expression eos

control_section_specification ::= [Control Section Specifications]
dc_help_specification

| noise_source_specification
| collapse_specification
| start_value_specification
| initial_condition_specification
| restart_specification
| device_type_specification
| nonlinearity_specification
| sample_point_specification
| newton_step_specification
| partial_derivative_specification
| small_signal_specification
| stress_measure_specification
| range_set_specification
| unit_range_specification
| variable_range_specification

dc_help_specification ::= [DC_Help Specification]
*[* **control_section** *]*
**dc_help** ( *plus*_pin_aspect , *minus*_ pin_aspect ) eos

decimal_literal ::= [Integer Literals]
digit *{* digit *}*

decimal_name ::= decimal_literal [Decimal Names]

declaration_statement ::= [Declarations]
function_declaration
| type_declaration
| unit_declaration
| pin_type_declaration
| object_declaration
| group_declaration
| function_definition
| template_definition

declarative_item ::= [Template Bodies]
declaration_statement
| alter_specification
| control_section_specification

declarator ::= [Object Declarations]
identifier *[* "[" index_constraint *{* , index_constraint *}* "]" *]*

[ = expression ]

declarator_list ::= [Object Declarations]
declarator *{* , declarator *}*

default_value ::= literal [Derived Unit Declaration]

derived_unit_declaration ::= [Derived Unit Declaration]
derived_unit_definition identifier *[* = default_value *]*
*[* resolution_indication *]* eos

derived_unit_definition ::= [Derived Units]
**unit** unit_mark

design_unit ::= [Design Units]
library_unit

device_type_specification ::= [Device Type Specification]
*[* **control_section** *]* **device_type** ( *device_class*_expression,
*device_subclass*_expression ) eos

digit ::= one of [Character Set]
0 1 2 3 4 5 6 7 8 9

element_declaration ::= [Structure Types]
variable_declaration

else_part ::= [Conditional Statement]
**else** *{* format_effector *}* statement

enumeration_default_value ::= simple_name [Enumeration Unit Declarations]

enumeration_literal ::= identifier [Enumeration Types]

enumeration_type_definition ::= [Enumeration Types]
**enum** *[* identifier *]* "{" enumeration_literal
*{* , enumeration_literal *}* "}"

enumeration_unit_declaration ::= [Enumeration Unit Declarations]
enumeration_unit_definition identifier = enumeration_default_value
*[* "{" resolution_indication "}" *]*

enumeration_unit_definition ::= [Enumeration Units]

**unit state** "{" enumeration_literal , string_literal ,
string_literal , string_literal
{ , enumeration_literal , string_literal ,
string_literal , string_literal } "}"

eos ::= format_effector | ; [Sentence Termination]

equality_expression ::= [Expressions]
relational_expression *{* equality_operator relational_expression *}*

equality_operator ::= == | ~= [Equality Operators]

equation_statement ::= [Equation Statements]
contribution_statement
| labeled_equation_statement
| make_statement

executable_statement ::= [Executable Statements]
assignment_statement
| loop_statement
| exit_statement
| next_statement
| return_statement
| when_statement
| equation_statement

exit_statement ::= [Exit Statement]
**exit** eos

exponent ::= [Real Literals]
exponent_mark [ + ] decimal_literal
| exponent_mark - decimal_literal
| scale_factor

exponent_mark ::= e | d | E | D [Real Literals]

expression ::= [Expressions]
logical_or_expression

expression_list ::= [Array Aggregates]
expression *{* , expression *}*

extended_identifier ::= [Extended Identifiers]

@ " graphic_or_space_character *{* graphic_or_space_character *}* "

extended_name ::= [Names]
name
| decimal_name

extended_type_mark ::= [Function Declarations]
type_mark [ "[" index_constraint *{* , index_constraint *}* "]" *]*

extended_type_mark_list ::= [Function Declarations]
extended_type_mark *{* , extended_type_mark *} [* , ... *]*
| ...

extended_variable_declaration_list ::= [Function Declarations]
variable_declaration_list *[* , ... *]*
| ...

factor ::= [Expressions]
*[* unary_operator *]* primary *{* ** *[* unary_operator *]* primary }

file_inclusion ::= [File Inclusion]
< file_name_literal

file_name_literal ::= [File Inclusion]
graphic_character *{* graphic_character *}*

formal ::= [Argument Association Lists]
argument_simple_name

format_effector ::= one of [Character Set]
VT CR LF FF

function_argument_list ::= [Function Header]
argument_list
| variable_declaration_list

function_attribute ::= [Function Header]
**encrypted**
| **foreign**

function_body ::= [Function Body]
function_sentence *{* function_sentence *}*

```
function_body_declarative_item ::=                [Function Body]
          function_declaration
        | type_declaration
        | unit_declaration
        | variable_declaration
        | group_declaration
        | function_definition

function_call ::=                                 [Function Calls]
        name ( [ function_argument_association_list ] )

function_declaration ::=                          [Function Declarations]
          foreign identifier_list eos
        |  { function_attribute } type_indication function_declarator_list eos

function_declarator ::=                           [Function Declarations]
        identifier ( [ extended_variable_declaration_list ] )

function_declarator_list ::=                      [Function Declarations]
        function_declarator { , function_declarator }

function_definition ::=                           [Function Definitions]
        function_header "{" function_body "}" eos

function_header ::=                               [Function Header]
        function_header_definition { function_header_sentence }

function_header_declarative_item ::=              [Function Header]
        variable_declaration

function_header_definition ::=                    [Function Header]
        { function_attribute } function result_indication =
                          identifier ( [ function_argument_list ] ) eos

function_header_sentence ::=                      []
          function_header_declarative_item
        | eos

function_sentence ::=                             [Function Body]
          function_body_declarative_item
        | function_statement
        | eos
```

function_statement ::= [Function Body]
assignment_statement
| conditional_statement
| compound_statement
| return_statement
| loop_statement
| exit_statement
| next_statement

generic_statement ::= [Generic Statements]
conditional_statement
| compound_statement

graphic_character ::= [Character Set]
lowercase_letter
| uppercase_letter
| digit
| punctuation_character
| special_character

graphic_or_space_character ::= [Extended Identifiers]
graphic_character | white_space_character

group_constituent ::= name [Group Declarations]

group_constituent_list ::= [Group Declarations]
group_constituent *{* , group_constituent *}*

group_declaration ::= [Group Declarations]
**group** "{" group_constituent_list "}" identifier eos

hex_digit ::= digit | a | b | c | d | e | f | A | B | C | D | E | F[Integer Literals]

hexadecimal_literal ::= [Integer Literals]
0 hex_mark hex_digit { hex_digit }

hex_mark ::= x | X [Integer Literals]

identifier ::= [Identifiers]
basic_identifier
| extended_identifier

identifier_list ::= [Function Declarations]

identifier *{* , identifier *}*

imported_name ::= [Imported Names]
simple_name ( instance_name )

index_constraint ::= [Index Constraints]
*[* lower_bound : *]* upper_bound
| *enumeration*_type_mark

indexed_name ::= [Indexed Names]
prefix "[" expression *{* , expression *}* "]"

initial_condition_specification ::= [Initial Condition Specification]
*[* **control_section** ]
**initial_condition** ( analog_name, expression ) eos

inline_group ::= [Inline Groups]
( group_constituent_list )

instance_name ::= [Instance Names]
prefix . reference_designator

instantiation_statement ::= [Instantiation Statement]
instance_name connection_association_list
*[* = *instance*_argument_association_list *]* eos

integer_literal ::= [Integer Literals]
decimal_literal | octal_literal | hex_literal

iteration_scheme ::= [Loop Statement]
**while** ( condition )
| **for** ( *[* *initial*_assignment *]* ; *[* condition *]* ; *[* update_assignment *]* )

labeled_equation_statement ::= [Labeled Equation Statement]
*[* statement_attribute *]* name : *lhs*_expression = *rhs*_expression eos

letter_or_digit_or_underline ::= [Basic Identifiers]
lowercase_letter | uppercase_letter | digit | _

letter_or_underline ::= [Basic Identifiers]
lowercase_letter | uppercase_letter | _

library_unit ::= [Design Units]

root_template
| template_definition
| function_definition

literal ::= [Literals]
numeric_literal
| string_literal
| enumeration_literal
| **inf**
| **undef**

logical_and_expression ::= [Expressions]
equality_expression *{* & equality_expression *}*

logical_or_expression ::= [Expressions]
logical_and_expression *{* "|" logical_and_expression *}*

loop_statement ::= [Loop Statement]
*[* statement_attribute *]* iteration_scheme *{* format_effector *}* statement

lower_bound ::= [Index Constraints]
expression

lowercase_letter ::= one of [Character Set]
a b c d e f g h i j k l m n o p q r s t u v w x y z
ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø ù ú û ü ý þ ÿ

make_statement ::= [Make Statement]
*[* statement_attribute *]* **make** *lhs*_expression = *rhs*_expression eos

mode ::= **input** | **output** | **inout** [State Declarations]

multiplicative_operator ::= * | / | % [Multiplicative Operators]

name ::= [Names]
simple_name
| instance_name
| imported _name
| selected_name
| branch_name
| indexed_name
| slice_name
| qualified_name

named_library_unit ::= [Compilation Units]
template_definition
| function_definition

newton_step_specification ::= [Newton Step Specification]
*[* **control_section** *]* **newton_step** ( *independent*_set,
*increase*_expression *[* , *decrease*_expression *]* ) eos

next_statement ::= [Next Statement]
**next** eos

noise_source_specification ::= [Noise Source Specification]
*[* **control_section** *]* **noise_source** ( *source*_simple_name,
*plus*_pin_aspect [ , *minus*_pin_aspect ] ) eos
| *[* **control_section** *]* **noise_source** ( *source*_simple_name,
*asv*_simple_name ) eos

nonexecutable_statement ::= [Nonexecutable Statements]
instantiation_statement

nonlinearity_specification ::= [Nonlinearity Specification]
*[* **control_section** *]* **pl_set** ( *dependent*_set, *independent*_set
*[* , *limiting_function*_name *[* , expression *] ]* ) eos

numeric_literal ::= [Numeric Literals]
integer_literal | real_literal

object_declaration ::= [Object Declarations]
parameter_declaration
| variable_declaration
| state_declaration
| analog_variable_declaration
| pin_declaration
| simulator_variable_declaration

octal_literal ::= [Integer Literals]
0 octal_mark digit { digit }

octal_mark ::= o | O [Integer Literals]

parameter_attribute ::= [Parameter Declarations]
**export**
| **external**

| **const**

parameter_declaration ::= [Parameter Declarations]
*{* parameter_attribute *} [* **parameter** *]* type_mark
declarator_list eos
| *{* parameter_attribute } **parameter** *unit*_name
declarator_list eos

partial_derivative_specification ::= [Partial Derivative Specification]
*[* **control_section** *]* **ss_partial** ( identifier, expression, *wrt*_name ) eos

physical_unit_declaration ::= [Physical Unit Declarations]
physical_unit_definition identifier eos

physical_unit_definition ::= [Physical Units]
**unit** "{" *string*_expression , *string*_expression , *string*_expression "}"

pin_aspect ::= [Branch Names]
*pin*_name
| ( *pin*_name )
| 0

pin_attribute ::= [Pin Declarations]
**export**
| **external**

pin_declaration ::= [Pin Declarations]
*{* pin_attribute *}* pin_type_mark declarator_list eos

pin_type_declaration ::= [Pin Type Declarations]
scalar_pin_type_declaration
| structure_pin_type_declaration

pin_type_mark ::= [Pin Declarations]
*pin_type*_name
| structure_pin_type_definition
| *pin*_type_reference

prefix ::= name [Names]

primary ::= [Primaries]
name
| literal

```
        | function_call
        | aggregate
        | structure_overlay
        | conditional_expression
        | ( expression )

punctuation_character ::= one of                [Character Set]
        " # % & ( ) * + , - . / : ; < = > @ [ ] \ _ { } | ~ m

qualified_name ::=                              [Qualified Names]
        prefix .. suffix

range_set_specification ::=                     [Range Set Specification]
        [ control_section ] range_set ( expression ) eos

real_literal ::=                                [Real Literals]
          decimal_literal [ . decimal_literal ] [ exponent ]
        | . decimal_literal [ exponent ]

reference_designator ::=                        [Instance Names]
          identifier
        | special_reference_designator

ref_declaration ::=                             [Ref Declarations]
        { analog_attribute } ref_indication unit_mark declarator_list eos

ref_indication ::=          [ input ] [ ref ]   [Ref Declarations]

relational_expression ::=                       [Expressions]
        additive_expression { relational_operator additive_expression }

relational_operator ::= < | > | <= | >=         [Relational Operators]

resolution_indication ::=                       [Enumeration Unit Declarations]
        conflict_resolution : function_declaration

restart_specification ::=                       [Restart Specification]
        [ control_section ] adjust_on_restart ( simple_name
                                { , simple_name } ) eos

result_indication ::=                           [Function Header]
          untyped_result_indication
        | typed_result_indication
```

return_statement ::= [Return Statement]
    **return** eos

root_template ::= [Template Definitions]
    template_body

sample_point_specification ::= [Sample Point Specification]
    *[* **control_section** *]*
        **sample_points** ( *independent*_set, expression ) eos

scalar_pin_type_declaration ::= [Scalar Pin Type Declarations]
    **pin** identifier scalar_pin_type_definition

scalar_pin_type_definition ::= [Scalar Pin Types]
      across_aspect through_aspect
    | through_aspect across_aspect

scalar_type_definition ::= [Scalar Common Types]
    enumeration_type_definition

scale_factor ::= [Real Literals]
    a | f | p | n | u | mu | μ | m | k | me | meg | g | t

selected_name ::= [Selected Names]
    prefix -> simple_name

sentence ::= [Template Bodies]
      declarative_item
    | statement
    | eos

set ::= analog_name | inline_group [Nonlinearity Specification]

simple_name ::= [Simple Names]
    identifier

simulator_variable_declaration ::= [Simulator Variable Declarations]
    **simvar** identifier_list eos

slice_name ::= prefix "[" index_constraint "]" [Slice Names]

small_signal_specification ::= [Small-Signal Specification]
    *[* **control_section** *]* **small_signal** ( identifier, *category*_identifier,

*report*_expression, expression *[* , *wrt*_name *]* ) eos

special_character ::= one of [Character Set]
` ! $ ' ? ^ ¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ – ® ¯ ° ± ² ³ ´ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿

special_reference_designator ::= [Special Reference Designators]
digit { letter_or_digit_or_underline }

specifier ::= [Alter Specification]
simple_name = expression

specifier_list ::= [Alter Specification]
specifier *{* , specifier *}*

start_value_specification ::= [Start Value Specification]
*[* **control_section** *]* **start_value** ( analog_name, expression ) eos

state_attribute ::= [State Declarations]
**external**
| **foreign**

state_declaration ::= [State Declarations]
*{* state_attribute *} [* mode *]* **state** unit_mark declarator_list eos

statement ::= [Statements]
executable_statement
| nonexecutable_statement
| generic_statement

statement_attribute ::= [Statements]
**parameters**
| **equations**
| **values**
| **states**
| **control_section**

stress_measure_specification ::= [Stress Measure Specification]
*[* **control_section** *]* **stress_measure** (
identifier, *category*_identifier,
*report*_expression, *stress*_expression,
simple_name, *rating*_expression
*[* , *reference_rating*_expression *]* ) eos

string_literal ::= " *{* graphic_or_space_character *}* "[String Literals]

structure_aggregate ::= [Structure Aggregates]
( *[* aggregate_association_list *]* )

structure_overlay ::= [Structure Overlays]
prefix <- ( *[* aggregate_association_list *]* )

structure_pin_type_declaration ::= [Structure Pin Type Declarations]
**pin** structure_pin_type_definition eos

structure_pin_type_definition ::= [Structure Pin Types]
**struc** *[* identifier *]* "{" *{* format_effector *}*
*element*_pin_declaration *{* format_effector *}*
*{ element*_pin_declaration *{* format_effector *} }* "}"

structure_type_definition ::= [Structure Types]
**struc** *[* identifier *]* "{" *{* format_effector *}*
element_declaration *{* format_effector *}*
*{* element_declaration *{* format_effector *} }* "}"

suffix ::= simple_name [Qualified Names]

tag ::= [Derived Units]
simple_name
| qualified_name

target ::= [Assignment Statement]
name
| inline_group

template_attribute ::= [Template Header]
**encrypted**
| **element**
| **component**

template_body ::= [Template Bodies]
*{* template_body_sentence *}*

template_body_sentence ::= sentence [Template Bodies]

template_definition ::= [Template Definitions]
template_header "{" template_body "}" eos

template_header ::= [Template Header]
template_header_definition *{* template_header_sentence *}*

template_header_declarative_item ::= [Template Header Declarations]
type_declaration
| unit_declaration
| pin_type_declaration
| parameter_declaration
| state_declaration
| analog_variable_declaration
| pin_declaration

template_header_definition ::= [Template Header]
*{* template_attribute *}* **template** identifier
*[* connection_list *] [* = *template*_argument_list *]* eos

template_header_sentence ::= [Template Header Declarations]
template_header_declarative_item
| eos

term ::= [Expressions]
factor *{* multiplicative_operator factor *}*

then_part ::= [Conditional Statement]
*[* **then** *{* format_effector *} ]* statement

through_aspect ::= [Scalar Pin Types]
**through** *physical_unit*_name

type_declaration ::= [Type Declarations]
type_definition eos

type_definition ::= [Common Types]
scalar_type_definition
| composite_type_definition

type_indication ::= [Function Declarations]
extended_type_mark
| ( extended_type_mark_list )

type_mark ::= [Derived Units]
**integer**
| **number**

```
        | string
        | type_definition
        | type_reference

type_reference ::=                                  [Derived Units]
          enum enumeration_tag
        | struc structure_tag
        | union union_tag
        | qualified_name

typed_result_indication ::=                         [Function Header]
         variable_declaration
        | ( variable_declaration_list )

unary_operator ::= + | - | ~                        [Unary Operators]

union_aggregate ::=                                 [Union Aggregates]
        ( aggregate_association_element )

union_type_definition ::=                           [Union Types]
        union [ identifier ] "{" { format_effector }
                        element_declaration { format_effector }
                        { element_declaration { format_effector } } "}"

unit_aspect ::= unit_name                           [Branch Names]

unit_declaration ::=                                [Unit Declarations]
          physical_unit_declaration
        | enumeration_unit_declaration
        | derived_unit_declaration

unit_mark ::=                                       [Derived Units]
          unit_name
        | type_mark

unit_range_specification ::=                        [Unit Range Specification]
        [ control_section ] range_for_unit ( unit_name,
                        min_expression, max_expression
                        [ , abs_expression [ , rel_expression ] ] ) eos

untyped_result_indication ::=                       [Function Header]
          identifier
        | inline_group
```

upper_bound ::= [Index Constraints]
expression
| *

uppercase_letter ::= one of [Character Set]
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Þ

val_declaration ::= [Val Declarations]
*{* analog_attribute *}* **val** unit_mark declarator_list eos

variable_attribute ::= [Variable Declarations]
**const**

variable_declaration ::= [Variable Declarations]
*[* variable_attribute *] [* **variable** *]* type_mark declarator_list eos
| *[* variable_attribute *]* **variable** *unit*_name declarator_list eos

variable_declaration_list ::= [Function Header]
variable_declaration *{* , variable_declaration *}*

variable_range_specification ::= [Variable Range Specification]
[ **control_section** *]* **range_for_variable** ( *variable*_set,
*min*_expression, *max*_expression
*[* , *abs*_expression *[* , *rel*_expression *] ]* ) eos

var_declaration ::= [Var Declarations]
*{* analog_attribute *}* var_indication unit_mark declarator_list eos

var_indication ::= *[* **output** *] [* **var** *]* [Var Declarations]

when_statement ::= [When Statement]
*[* statement_attribute *]*
**when** ( expression ) *{* format_effector *}* statement

white_space_character ::= one of [Character Set]
SP NBSP HT

# 16

# Glossary

## MAST Glossary

across branch

A branch name whose unit aspect is the across unit of the plus pin or minus pin of the branch.

across unit

The unit denoted by the across aspect of a scalar pin type definition.

active driver

A driver is active when the current time becomes equal to the time element of the transaction following the effective transaction.

active state

A state that is a component of a net that has an active driver.

adjustable state

A state whose name appears in an adjust_on_restart specification.

alias

An alternate name for an entity.

alternative

An element of a union type.

analog local variable

A branch across variable or an analog variable of kind val.

analog net

A collection of analog system variables or pins that are associated by connection association elements.

analog system variable

A branch through variable or an analog variable of kind **var** or **ref**.

ancestor

The parent instance of an instance, its parent instance, the parent instance of its parent instance, etc., up to the root instance.

apply

A specification applies to an entity if it relates to the entity and has been elaborated.

argument constant expression

A globally constant expression whose primaries are neither the target of an assignment statement decorated with the **parameters** attribute nor the actual argument of a function call calling the ADDR function in a statement decorated with the **parameters** attribute.

argument constant type

A scalar type, or an array type whose element type is an argument constant type and whose index constraints have index ranges with argument constant lower bound, if present, and upper bound, or a structure type or union type whose elements are of an argument constant type. For a relaxed argument constant type any index range may have an assumed upper bound.

argument marshalling

See marshalling.

argument profile

The profile of the objects in an argument list.

assigned state

A state whose value is updated by executing an assignment statement or by an implicit assignment.

assumed

An upper bound of an index range specified by an asterisk (*). Also, an index range whose upper bound is assumed.

base name

A unique name of a type, used to establish type compatibility and pin type compatibility.

base type

The common type of a unit.

binary operator

An operator with two operands.

branch across variable

A branch variable whose branch definition denotes an across branch.

branch through local variable

A branch through variable whose name, or the name of any of its subelements, appears as the target of an assignment statement or the name in a contribution statement, but does not appear as a primary in an expression.

branch through system variable

A branch through variable that is not a branch through local variable.

branch through variable

A branch variable whose branch definition denotes a through branch.

canonical group

A group whose group constituents are objects.

cardinality

The number of types in the profile of an ordered collection of objects or values.

characteristic expression

An expression used by the analog solver to determine the values of the analog variables.

clearing a driver

The deletion of all transactions except the transaction defining the current value of the driver.

common type

A set of values and a set of operations.

composite type

A type that has elements. Composite common types are array types, structure types, and union types. Composite pin types are array pin types and structure pin types.

constant expression

An expression whose value can be determined prior to simulation.

constant name

A name in which each expression that appears in the name is globally constant.

context

A collection of declarations that define an environment for the compilation of a design unit.

current alternative

The element of a union type whose value is the value of the union type.

current value

The value element of the effective transaction of a driver.

declared

An identifier or decimal literal that has been associated with an entity by a declaration is said to be declared.

decorate

The association of an attribute with an entity.

decrease newton step

For the decrease expression of a newton step specification, the value of the subelement INC of the array element that defines the left end of an interval. The decrease newton step limits the change of the independent variable with which it is associated from one iteration to the next if the change is negative.

default value

The initial value of an element of a structure type.

denote

A property of a name or decimal name. The name or decimal name is said to denote an entity where the declaration of the entity is visible.

dependent variable

A scalar subelement of an analog local variable that is a term in an expression, a transform, or an actual expression in a function call in a statement decorated with the **equations** attribute and that depends nonlinearly on independent variables.

design entity

A portion of a hardware design that performs a well-defined function and has a well-defined interaction with the rest of the design. Implemented as an MAST template.

design hierarchy

The complete representation of a design that results from the successive decomposition of a design entity into components, each bound to a design entity.

design library

An implementation-dependent container for compiled design units.

design unit

A portion of code that can be independently compiled and inserted into a design library.

dimensionality

The number of indices of an array type or an array pin type.

directly visible

A declaration is said to be directly visible if the identifier associated with an entity by the declaration is sufficient to denote the entity.

discrete type

An enumeration type or the predefined type INTEGER.

driver

An ordered sequence of one of more transactions. There is a driver for each scalar subelement of each driving state in an instance, for each simulator variable with state semantics, for the implicit break state, for the implicit halt state, and for the implicit state associated with each function call calling the THRESHOLD function.

driving state

A state denoted by the longest constant prefix of the second argument of function SCHEDULE_EVENT. Its value is updated by the state update algorithm.

driving value

For a scalar driving state, the value element of the effective transaction of its driver. For a composite driving state, the composite of the driving values of the scalar subelements of the state.

effective transaction

The one transaction of a driver whose time element is not greater than the current simulation time.

effective value

The value of an event-driven state obtained when evaluating a primary denoting the state in an expression.

elaboration

The process by which a sentence takes effect.

element

A constituent of a composite type.

element pin type

The pin type of an element of an array pin type.

element type

The type of an element of an array type.

entity

An item associated with an identifier or decimal literal by an explicit or implicit declaration.

event

Updating the value of an event-driven state.

event-driven state

A state that is a driving state, or an observed state, or a formal connection of a template, or an actual in a connection association element, or decorated with the **foreign** attribute.

explicit characteristic expression

A characteristic expression defined by equation statements.

explicitly declared

An object declared by a declaration statement is said to be *explicitly declared*.

external name

The name or decimal name of a template connection element defined by the connection definition of the template connection element.

foreign function

A function written in a general purpose programming language, such as C or Fortran. A foreign function must be decorated with the **foreign** attribute in its function declaration.

free

An analog net is free if it is associated with either a branch through variable or an analog system variable of kind **var** whose name is not the label of a labeled equation statement.

fully qualified name

A qualified name where the simple name preceding the first delimiter in the qualified name denotes a design library.

globally constant expression

A constant expression whose value can be determined as part of elaboration.

globally constant type

A scalar type, or an array type whose element type is a globally constant type and whose index constraints have index ranges with globally constant lower bound, if present, and upper bound, or a structure type or union type whose elements are of a globally constant type.

guard

A statement that makes another statement a guarded statement.

guarded statement

A statement whose execution depends on the value of a condition.

hidden

A declaration that is not directly visible.

immediately within

A declaration is said to occur immediately within a declarative region if this region is the innermost declarative region that encloses the declaration.

immediate scope

A portion of a declarative region immediately enclosing a declaration.

implicitly declared

An object declared by a statement other than a declaration statement.

imported object

An object denoted by a name that either is an imported name or has a prefix that contains a portion that is an imported name.

impure function

A function that may return a different result each time it is called, even if different function calls have the same values as actual arguments.

increase newton step

For the increase expression of a newton step specification, the value of the subelement INC of the array element that defines the left end of an interval. The increase newton step limits the change of the independent variable with which it is associated from one iteration to the next if the change is positive.

independent variable

A scalar subelement of an analog system variable, a branch variable, or an analog local variable whose value is defined as an analog system variable or a branch variable, optionally preceded by a unary plus or a unary minus operator, or the difference of two analog system variables, branch through variables, or branch across variables or across branch. names where one pin aspect denotes the reference pin 0.

index range

The range of an index.

inherited

An implicit decoration of a statement obtained from another statement that either guards or encloses the statement.

initial point

A collection of values related to a design, consisting of a time, a value for each analog system variable in the design, a value and a driver for each event-driven state in the design, and a value for each assigned state in the design.

initial value

The value of an initial value expression. It is assigned to the object denoted by the declarator by the elaboration.

initial value expression

The expression that may be part of a declarator.

internal name

The name or decimal name of a template connection element defined by the connection specification of the template connection element.

keyword

An identifier that has a special meaning in the language.

label

The name preceding the colon in a labeled equation statement.

left operand

The operand preceding a binary operator.

length

The number of values in an index range.

locally constant expression

A constant expression whose value can be determined as part of compilation.

locally constant name

A name in which each expression that appears in the name is locally constant.

locally constant type

A scalar type, or an array type whose element type is a locally constant type and whose index constraints have index ranges with locally constant lower bound, if present, and upper bound, or a structure type or union type whose elements are of a locally constant type. For a relaxed locally constant type any index range may have an assumed upper bound.

longest constant prefix

For a constant name, the name itself. Otherwise, the longest prefix of the name that is a constant name.

major declarative region

A declarative region other than a compound statement.

marshalling

An algorithm to copy values of a common type into an array.

member of a set

Either a group constituent if the set is defined as a group, or the object that defines the set.

minor declarative region

A compound statement.

minus pin

The pin denoted by the minus pin aspect of a branch name.

name

An identifier associated with an entity by a declaration. For a type or pin type, a keyword together with an identifier.

named association

An argument association element, connection association element, or aggregate association element containing an explicit specification of the corresponding formal argument, formal connection, or element of the structure type or union type.

name space

see overloading class.

net

A collection of states that are associated by connection association elements.

node

The analog net associated with a pin.

nonparameter group

A group that is not a parameter group.

normalized index range

An index range whose lower bound is 1 and whose upper bound is the length of the index range.

null slice

A slice whose lower bound value exceeds the upper bound value.

numeric type

The predefined types INTEGER and NUMBER.

object

An entity of a given common type or pin type.

observed state

A state containing a subelement whose name appears as the first argument of function EVENT_ON or that is associated as an actual with a formal that is an observed state.

overloaded

An identifier or enumeration literal that has more than one possible meaning.

overloading class

A category of entities within which a name must be unique.

parameter group

A group is a parameter group if the group constituents of the corresponding canonical group are all of class parameter or variable. A group constituent of class variable must be declared either in the major declarative region associated with a function or in a minor declarative region decorated with the **parameters** attribute.

parent instance

The instance containing an instantiation statement is the parent instance of the instance defined by the instantiation statement.

pin flow expression

An expression defining the amount of flow at a pin. The pin flow expression of a pin that defines a node is a characteristic expression of the model.

pin type

A property of a pin through which its compatibility with other pins is established.

plus pin

The pin denoted by the plus pin aspect of a branch name.

positional association

An argument association element, connection association element, or aggregate association element whose corresponding formal argument, formal connection, or element of the structure type is determined by the textual position of the association element in the argument association list, connection association list, or structure aggregate, respectively.

profile

The types of the objects or values of an ordered collection of objects or values, in the order defined by the collection.

pure function

A function that always returns the same result when called with the same values as actual arguments.

range

The set of values representable by a type.

range set

A named collection of ranges.

read

A state is said to be read if it is an observed state or if the name of one of its subelements appears in an expression or as the first argument of one of the functions LAST_VALUE, RAMP, or SLEW.

ref

An analog system variable whose matching equation is defined in another declarative region.

relate

A specification that associates additional information with an entity relates to the entity.

reserved word

A keyword that must not be used as the name of an entity.

resolved unit

A unit whose unit declaration includes a resolution indication.

right operand

The operand following a binary operator.

root instance

The instance at the root of a design hierarchy, representing a complete hardware design.

root template

The template defining the root instance.

scalar type

A type that has no elements. Scalar common types are enumeration types and the predefined types INTEGER, NUMBER, and STRING. Scalar pin types are pin types with scalar across and through aspects.

scheduled state

A state containing a subelement that is either a scheduled state or a driving state or that is associated as an actual with a formal that is a scheduled state.

scope

A portion of a description over which a declaration is active.

sensitive

A when statement is sensitive to a state S if S appears in a specific way in the condition of the when statement.

set

An unordered collection of scalar objects.

simple assignment statement

An assignment statement whose expression is a simple expression.

simple expression

An expression with a profile whose cardinality is one.

simulatable model

The result of elaborating a design hierarchy.

stamp expression

An expression defined by an equation statement or an assignment statement decorated with the **equations** attribute. The scalar subelements of a stamp expression are used to construct the characteristic expressions.

subaggregate

An array aggregate appearing as an expression in an n-dimensional array aggregate.

subelement

A scalar, an element, or an element of an element.

supertype

A type implied by two compatible common types.

tag

An identifier of an enumeration type definition, structure type definition, union type definition, or structure pin type definition that together with a keyword is the name of the corresponding type or pin type.

threshold expression

The difference of the waveform and the reference waveform of a function call calling the THRESHOLD function.

through branch

A branch name whose unit aspect is the through unit of the plus pin or minus pin of the branch.

through unit

The unit denoted by the through aspect of a scalar pin type definition.

tolerance range

A 4-tuple of values, consisting of a minimum, a maximum, an absolute tolerance, and a relative tolerance.

transaction

An element of a driver, consisting of a time and a value and specifying the value that the state associated with the driver may attain at some time. A transaction is written as “value at time”.

type

A characteristic of an object or value through which the compatibility of the object or value with other objects or values can be established. See also common type.

unconstrained array type

An array type with an assumed index range.

unit

A scalar common type decorated with attributes specific to the kind of the unit.

var

An analog system variable whose matching equation is defined in the same declarative region.

visible

When a declaration defines a possible meaning for an identifier.

**Appendix 16: Glossary**
MAST Glossary

# 17

# External Interfaces

*This chapter summarizes MAST external interfaces.*

## Foreign Function Interface

A foreign function is a function written in a standard programming language such as C or Fortran. Foreign functions can be called from templates and MAST functions. Foreign functions can also be specified to be called by the simulator to apply limiting during the determination of an analog solution point.

This section describes the application program interface (API) for foreign functions and the protocols of calling such functions.

### Foreign Function API

The API for foreign functions is a Fortran API of the form:

```
subroutine name(in,nin,ifl,nifl,out,nout,ofl,nofl,undef,ier)
```

where name is the name of the foreign function and the formal arguments have the following types and meaning:

| | |
|---|---|
| `real*8 in(*)` | An array through which all arguments of a function call are passed to the foreign function |
| `integer nin` | The number of elements passed in array in |
| `integer ifl(*)` | An array of flags passed to the foreign function |
| `integer nifl` | The number of elements passed in array ifl |

| | |
|---|---|
| `real*8 out(*)` | An array through which the foreign function returns its results |
| `integer nout(2)` | The number of elements returned in array out. See |
| `integer ofl(*)` | An array of flags returned by the foreign function |
| `integer nofl` | The number of elements returned in array ofl |
| `real*8 undef` | The value of the primary undef, passed into the function |
| `integer ier` | A status returned by the foreign function |

The name of the foreign function must match the name used in the function declaration declaring the foreign function. The use of the arguments is specific to each protocol.

Any programming language that supports functions that are callable from Fortran can be used to write a foreign function. As an example, the corresponding API for the C language is:

```
void C_name(double *in, int *nin, int *ifl, int *nifl,
       double *out, int *nout, int *ofl, int *nofl,
       double *undef, int *ier);
```

where C_name is the C name that corresponds to the Fortran name name, and int has a 32 bit representation.

**Notes**

1. On some operating systems the C_name is the same as the name in all lower case. On other operating systems it is the same as the name in all upper case. On a third class of operating systems the C_name is the same as the name in all lower case, with an underscore (_) appended.
2. The name of the foreign function (name or C_name) must be explicitly exported on some operating systems.

## Foreign Functions Called from a Template or a MAST Function

When a foreign function is called from a template or an MAST function, the values of all actual arguments of the function call are passed to the foreign function by *marshalling* them into the formal argument `in` and setting `nin` to the number or array elements needed to hold the values. Additionally, the value

of the formal argument `undef` is set to the value of the primary **undef**. When the foreign function call has completed, the results of the foreign function call are unmarshalled from the formal argument `out` into one or more values, directed by the context in which the foreign function call appears.

This section describes the basic concepts of calling foreign functions and argument marshalling, that is, the algorithm used to copy values of a common type into an array. The inverse algorithm is used to unmarshal the result returned by the foreign function call.

## Basic Concepts of Calling Foreign Functions

When calling a foreign function, the argument profile and result profile of the function call are determined first. The argument profile of the function call is the profile of the actual arguments of the function call. It is an error if the argument profile of the function call does not match the argument profile of the foreign function and the argument profile of the foreign function has been specified and does not contain an unspecified portion. The result profile of the function call is determined as follows. If the result profile of the foreign function has been specified and does not contain an unspecified portion, then the result profile of the function call is the result profile of the foreign function. Otherwise, the function call is the expression in an assignment statement, and its result profile is the profile of the object or group that is the target of the assignment statement. It is an error if the result profile of the function call does not match the result profile of the foreign function and the result profile of the foreign function has been specified and does not contain an unspecified portion.

To call a foreign function, the values of the actual arguments of the function call are copied in the order in which they appear into consecutive elements of the formal argument `in` of the foreign function, starting with the first element of `in`. Each value is copied according to its type, as described below. The formal argument `nin` is then set to the number of elements of the `in` array occupied by the values. Next, if the result profile of the function call includes a type that has a subelement of an unconstrained array type, then the first element of the formal argument `nout` is set to 0. Otherwise, the first element of the formal argument `nout` is set to the number of elements expected to be returned by the foreign function in formal argument `out`. This number is determined from the result profile of the function call as described below. Next, the second element of the formal argument `nout` is set to the length of the formal argument `out`. This length must be greater than or equal to the value of the first element of the formal argument `nout`. Finally, the formal argument `undef` is set to the value of the primary **undef** for type **number**.

When a foreign function has been called, it must first determine the number of elements it will need in the formal argument `out` to hold all values of the result. Then, it must set the first element of the formal argument `nout` to this number and, if the number is smaller than the second element of the formal argument `nout`, end the function call without copying any values to the `out` array. Otherwise, the foreign routine must copy the result to be returned to the template or MAST function into consecutive elements of the formal argument `out` using the algorithm described in the remainder of this section, starting with the first element of `out`.

If a function call calling a foreign function completes and the value of the first element of the formal argument `nout` is larger than the value of the second element of `nout`, then the size of the formal argument `out` is increased such that the length of `out` matches or exceeds the value of the first element of the formal argument `nout`. The second element of the formal argument `nout` is then set to the new length of the formal argument `out` and the foreign function is called again. It is an error if the value of the first element of the formal argument `nout` is larger than the value of the second element of `nout` when the second call calling the foreign function completes.

When a function call calling a foreign function completes, the elements of the formal argument out are copied to one or more values in the expression in which the function call appears. Copying is driven by the result profile of the function call. It is an error if the number of elements to be copied is larger than the value of the first element of the formal argument `nout`.

## Marshalling a Value of Type INTEGER

Marshalling a value of type INTEGER consists of copying the value to the next available element in the destination array.

## Marshalling a Value of Type REAL

Marshalling a value of type REAL consists of copying the value to the next available element in the destination array.

## Marshalling a Value of Type STRING

Marshalling a value of type STRING consists of copying the value of a string handle to the next available element in the destination array. A string handle is a floating point value associated with the string value. A string handle can be passed to the kernel interface functions defined in Obtaining a Value of Type STRING to obtain the associated string value. Similarly, the kernel interface

functions defined in Defining a Value of Type STRING can be used to obtain a string handle for a string value.

## Marshalling a Value of an Enumeration Type

Marshalling a value of an enumeration type consists of determining the position of the enumeration literal whose enumeration value matches the value in the corresponding enumeration type definition, followed by copying the number indicating the position to the next available element in the destination array. The position of the first enumeration literal in an enumeration type definition is 1. The position of a value of an enumeration type that equals the value of the primary **undef** is the value itself. It is an error if the position of a value of an enumeration type cannot be determined.

## Marshalling a Value of a Structure Type

Marshalling a value of a structure type consists of copying the values of each element of the structure type in the order in which the elements appear in the structure type definition to the next available elements in the destination array.

## Marshalling a Value of a Union Type

Marshalling a value of a union type consists of copying the position of the current alternative in the corresponding union type definition followed by the value of the current alternative to the next available elements in the destination array. The position of the first alternative in a union type definition is 1. The position is the value of the primary **undef** if there is no current alternative. It is an error if the position of the current alternative cannot be determined.

## Marshalling a Value of an Array Type

Marshalling a value of an array type consists of copying the number of elements in the array value followed by each element of the array value to the next available elements in the destination array. For an array type whose dimensionality is greater than 1, the elements are copied by varying the index at the last index position first.

Marshalling an undefined value of an array type consists of copying the value of the primary **undef** to the next available element of the destination array.

**Note**

The order of the elements resulting from marshalling a multi-dimensional array is sometimes called row-major order.

## Limiting Functions

Limiting function are foreign functions called by the simulator kernel during the determination of an analog solution point, to limit the change of the independent variables of a nonlinearity from one iteration to the next. When a limiting function is called, the following values are marshalled into the formal argument `in` of the foreign function:

- The values of the elements of the optional expression following the name of the limiting function in the nonlinearity specification, if present.
- The values of the members of the independent set at the end of the previous iteration, in the order in which the members of the set have been specified in the nonlinearity specification.
- The proposed values of the members of the independent set at the current iteration, in the order in which the members of the set have been specified in the nonlinearity specification.
- The values of the members of the dependent set at the end of the previous iteration, in the order in which the members of the set have been specified in the nonlinearity specification.

Additionally, the value of the formal argument `nin` is set to the number of elements in array `in`, and the values of both the formal argument `nofl` and the first element of the formal argument `nout` are set to the number of members of the independent set.

The limiting function returns in its formal argument out the values to be taken by the simulator kernel for the members of the independent set at the current iteration, in the order in which the members of the set have been specified in the nonlinearity specification. Additionally, if the value of a member of the independent set returned in the formal argument `out` is equal to the proposed value of this member, then the corresponding element in the formal argument `ofl` of the foreign function is set to 0. Otherwise, the corresponding element in the formal argument `ofl` is set to 1.

## Kernel Interface

The simulator kernel provides an interface consisting of a collection of functions that can be called by foreign functions to request kernel services. This section defines these functions.

## Obtaining a Value of Type STRING

The functions in this category return the value of type STRING associated with a string handle. If the string value equals the value of the primary **undef**, the functions return a string value of length zero.

### Fortran API

```
subroutine getstr(handle, string, n)
```

where the formal arguments have the following types and meaning:

| | |
|---|---|
| `real*8 handle` | A string handle passed to the subroutine |
| `character*(*) string` | A character variable to receive the string value associated with the string handle |
| `integer n` | The length of the string value |

If the length of the string value is greater than the size of the formal argument `string`, then the value is truncated to fit the size of `string`.

### C API

```
char *cgetstr(double handle);
```

where the result and the formal argument have the following meaning:

| | |
|---|---|
| `handle` | A string handle passed to the function |
| `cgetstr` | A pointer to the string value associated with the string handle |

## Defining a Value of Type STRING

The functions in this category associate a string handle with a value of type STRING and return the string handle.

### Fortran API

```
subroutine setstr(string, n, handle)
```

where the formal arguments have the following types and meaning:

| | |
|---|---|
| `character*(*) string` | A string value passed to the subroutine |
| `integer n` | The number of characters in the string value |
| `real*8 handle` | The string handle associated with the string value |

### C API

```
double csetstr(const char *string);
```

where the result and the formal argument have the following meaning:

| | |
|---|---|
| `string` | A string value passed to the function |
| `csetstr` | The string handle associated with the string value |

## Obtaining the Name of the Current Design

The functions in this category provide access to the name given to the root instance by the implementation. See also Nonmathematical Functions.

### Fortran API

```
function dsgnnm()
```

where the result has the following type and meaning:

| | |
|---|---|
| `real*8 dsgnnm` | A string handle associated with the design name |

### C API

```
double c_dsgnnm(void);
```

where the result has the following meaning:

| | |
|---|---|
| `c_dsgnnm` | A string handle associated with the design name |

## Calculation of a Limited Exponential

The function in this category implements an exponential function that limits the function value to prevent overflow. The semantics of the function are defined in Logarithmic, Exponential and Related Functions.

### Fortran API

```
function limexp(x)
```

where the result and the formal argument have the following types and meaning:

| | |
|---|---|
| `real*8 x` | The value for which to calculate the limited exponential |
| `real*8 limexp` | The value of the limited exponential function |

### C API

```
double c_limexp(double x);
```

where the result and the formal argument have the following meaning:

| | |
|---|---|
| `x` | The value for which to calculate the limited exponential |
| `c_limexp` | The value of the limited exponential function |

## Obtaining the Value of Simulator Variables

The functions in this category allow a foreign routine to obtain the value of the simulator variables STATISTICAL and WORST_CASE.

### Fortran API

```
function statsv()

function wcsv()
```

where the results have the following types and meaning:

| | |
|---|---|
| `real*8 statsv` | The value of the simulator variable statistical |
| `real*8 wcsv` | The value of the simulator variable worst_case |

### C API

```
double c_statsv(void);

double c_wcsv(void);
```

where the results have the following types and meaning:

| | |
|---|---|
| `c_statsv` | The value of the simulator variable statistical |
| `c_wcsv` | The value of the simulator variable worst_case |

## Obtaining Random Values

The functions in this category provide access to a random number generator provided by the implementation. They can be called interchangeably with the RANDOM function defined in Semi-Numerical Functions.

### Fortran API

```
function envrnd()
```

where the result has the following type and meaning:

| | |
|---|---|
| `real*8 envrnd` | The next value from a pseudo-random sequence |

### C API

```
double c_envrnd(void);
```

where the result has the following meaning:

| | |
|---|---|
| `c_envrnd` | The next value from a pseudo-random sequence |

## Special Attributes

This section describes attributes that may decorate design units but that have no meaning in the MAST language. Such attributes are used for various purposes by tools operating on a description.

### The Encrypted Attribute

The **encrypted** attribute may decorate a design unit; a design unit decorated with this attribute is said to be an encrypted design unit. Encrypted design units may contain portions of text that have been encrypted and that are therefore unreadable for human readers. Implementations may treat instances of encrypted templates differently from instances of templates that are not encrypted templates in matters that are not related to the meaning of a description.

The encryption algorithm is not specified by this document.

### The Component Attribute

The **component** attribute may decorate a design entity; a design entity decorated with this attribute is said to be a component template. The purpose

of the **component** attribute is to indicate that the implementation of the design entity should be hidden from the user. Tools may treat instances of component templates differently from instances of templates that are not component templates in matters that are not related to the meaning of a description.

# Index

## Symbols



## A



## B

## C



## D

## E



## F



## G

## H



## I



## K



## L



## M

## N



## O



## P



## Q



## R

## S

## T



## U

## V



## W