

Details of Language Change

Changes shown in **red font**. Deletions shown in ~~strikethrough red font~~. Comments shown in **green font**.

Twiki has not been kind in regards to font colors and strikethrough. Word and PDF versions of these changes are attached as separate files to this LCS.

LRM 4.2.1 General (Subprogram declarations)

Page 19 near bottom

```
function_specification ::=  
[ pure | impure ] function designator  
subprogram_header  
[ [ parameter ] ( formal_parameter_list ) ] return [identifier :] type_mark
```

Page 20 first paragraph

The specification of a procedure specifies its designator, its generics (if any), and its formal parameters (if any). The specification of a function specifies its designator, its generics (if any), its formal parameters (if any), **the name that the function body may reference to retrieve attributes of the target (if any)**, the subtype of the returned value (the result subtype), and whether or not the function is pure. A function is impure if its specification contains the reserved word **impure**; otherwise, it is said to be pure. A procedure designator is always an identifier. A function designator is either an identifier or an operator symbol. A designator that is an operator symbol is used for the overloading of an operator (see 4.5.2). The sequence of characters represented by an operator symbol shall be an operator belonging to one of the classes of operators defined in 9.2. Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

Page 20 Add after the first paragraph

Add the following new material in entirety after the first paragraph

If the specification of a function includes an identifier prior to the result subtype, that identifier can be used within the function body as if it was a formal input variable of the result subtype of the function. The only allowable action within the function body with this identifier is to retrieve attributes. The attributes returned are the attributes of the target to which the output of the function is assigning. If the named function is used in a context that it does not assign to a target that has deterministic attributes, this form of the named function shall not be called. The attributes that may be referenced are:

- The elements **BASE**, **LEFT**, **RIGHT**, **HIGH**, **LOW**, **ASCENDING**, **SUBTYPE** defined in 16.2.2-Predefined attributes of types and objects.

- All attributes defined in 16.2.3 Predefined attributes of arrays and scalars

It is an error to reference any other attributes.

Page 21 Immediately before section 4.2.2.2

NOTE-- ~~Attributes of an actual are never passed into a subprogram.~~ References to an attribute of a formal parameter are legal only if that formal has such an attribute. Such references retrieve the value of the attribute associated with the actual.

LRM 4.2.2.2 Constant and variable parameters

Page 21, near middle

For parameters of class constant or variable, ~~only~~ the values ~~and attributes~~ of the actual or formal are transferred into or out of the subprogram call. The manner of such transfers, and the accompanying access privileges that are granted for constant and variable parameters, are described in this subclause.

~~For a nonforeign subprogram having a parameter of class constant or variable, if the actual is an identifier, the attributes of the parameter shall be assigned from the corresponding attributes of the actual. If the actual is not an identifier, the attributes of the parameter shall be assigned as:~~

- ~~- BASE, SUBTYPE: Same as BASE, SUBTYPE attributes as defined in section 16.2.2 Predefined attributes of types and objects~~
- ~~- LEFT, RIGHT, HIGH, LOW: Set to the value of the actual.~~
- ~~- ASCENDING: Set to TRUE~~

LRM 4.2.2.3 Signal parameters

Page 22 near middle

For a signal parameter of mode in or inout, the actual signal is associated with the corresponding formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, a reference to the formal signal parameter within an expression is equivalent to a reference to the actual signal. ~~A reference to an attribute of the formal signal parameter is equivalent to a reference to the attribute of the actual signal.~~

LRM 12.2 Scope of Declarations

Page 186 middle of page

- h) A declaration that occurs immediately within a protected type declaration
- i) An architecture body
- ~~j) A return identifier after the formal parameter declaration in a subprogram declaration or specification~~

Annex C - Syntax Summary

LRM 16.2.3 Predefined attributes of arrays

Page 243 top of the page

Renumber this section as 16.2.3.1 and rename 16.2.3 as "Predefined attributes of arrays and scalars"

Page 244 near the top of the page

Add new section 16.2.3.2 Predefined attributes of scalars with text as shown below. Note: This content of this new section is based solely on the text from 16.2.3.1, but removing all references to arrays. As such, the red font and strikethrough of text is meant only to convey the differences between 16.2.3.1 and 16.2.3.2 to aid in the review process. Strikethrough text is not to be part of the LRM revision.

16.2.3.2 Predefined attributes of scalars

Throughout this section one will see SA in red font with a strikethrough on the A only which may be difficult to discern.

SA'LEFT ~~[(N)]~~ Kind: Value.

Prefix: Any prefix SA that is appropriate for a ~~an array scalar~~ object, or an alias thereof, or that denotes a ~~an array-scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: Type of the left bound of the ~~Nth index~~ range of SA.

Result: Left bound of the ~~Nth index~~-range of SA. (If SA is an alias for a ~~an array scalar~~ object, then the result is the left bound of the ~~Nth index~~-range from the declaration of SA, not that of the object.)

SA'RIGHT ~~[(N)]~~ Kind: Value.

Prefix: Any prefix SA that is appropriate for a ~~an array scalar~~ object, or an alias thereof, or that denotes a ~~an array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: Type of the ~~Nth index~~-range of SA.

Result: Right bound of the ~~Nth index~~-range of SA. (If SA is an alias for a ~~an array scalar~~ object, then the result is the right bound of the ~~Nth index~~-range from the declaration of SA, not that of the object.)

SA'HIGH ~~[(N)]~~ Kind: Value.

Prefix: Any prefix SA that is appropriate for a ~~an array scalar~~ object, or an alias thereof, or that denotes a ~~an array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: Type of the ~~Nth index~~-range of SA.

Result: Upper bound of the ~~Nth index~~-range of *SA*. (If *SA* is an alias for an ~~array scalar~~ object, then the result is the upper bound of the ~~Nth index~~-range from the declaration of *SA*, not that of the object.)

SA'LOW ~~{(N)}~~ Kind: Value.

Prefix: Any prefix *SA* that is appropriate for an ~~array scalar~~ object, or an alias thereof, or that denotes an ~~array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: Type of the ~~Nth index~~-range of *SA*.

Result: Lower bound of the ~~Nth index~~-range of *SA*. (If *SA* is an alias for an ~~array scalar~~ object, then the result is the lower bound of the ~~Nth index~~-range from the declaration of *SA*, not that of the object.)

SA'RANGE ~~{(N)}~~ Kind: Range.

Prefix: Any prefix *SA* that is appropriate for an ~~array scalar~~ object, or an alias thereof, or that denotes an ~~array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: The type of the ~~Nth index~~-range of *SA*.

Result: The range *SA*'LEFT(~~N~~) to *SA*'RIGHT(~~N~~) if the ~~Nth index~~-range of *SA* is

ascending, or the range *SA*'LEFT(~~N~~) **downto** *SA*'RIGHT(~~N~~) if the ~~Nth~~-range of *SA* is descending. (If *SA* is an alias for ~~scalar~~ object, then the result is determined by the ~~Nth index~~-range from the declaration of *SA*, not that of the object.)

SA'REVERSE_RANGE ~~{(N)}~~ Kind: Range.

Prefix: Any prefix *SA* that is appropriate for a ~~an array scalar~~ object, or an alias thereof, or that denotes a ~~an array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: The type of the ~~Nth index~~-range of *SA*.

Result: The range *SA*'RIGHT(~~N~~) **downto** *SA*'LEFT(~~N~~) if the ~~Nth index~~-range of *SA* is ascending, or the range *SA*'RIGHT(~~N~~) to *SA*'LEFT(~~N~~) if the ~~Nth~~-range of *SA* is descending. (If *SA* is an alias for an ~~array scalar~~ object, then the result is determined by the ~~Nth index~~-range from the declaration of *SA*, not that of the object.)

SA'LENGTH ~~{(N)}~~ Kind: Value.

Prefix: Any prefix *SA* that is appropriate for a ~~an array scalar~~ object, or an alias thereof, or that denotes an ~~array scalar~~ subtype whose ~~index range is ranges-are~~ defined by a constraint. *SA* cannot be of type real or a physical type.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: *universal_integer* .

Result: Number of values in the ~~Nth-index~~-range; i.e., if the ~~Nth-index~~-range of SA is a null range, then the result is 0. Otherwise, the result is the value of T'POS(SA'HIGH(~~N~~)) T'POS(SA'LOW(~~N~~)) + 1, where T is the subtype of ~~the Nth-index of SA~~.

SA'ASCENDING [~~(N)~~] Kind: Value.

Prefix: Any prefix SA that is appropriate for an array scalar object, or an alias thereof, or that denotes an array scalar subtype whose ~~index range is ranges-are~~ defined by a constraint.

~~Parameter: A locally static expression of type universal_integer, the value of which shall not exceed the dimensionality of A. If omitted, it defaults to 1.~~

Result type: Type BOOLEAN.

Result: TRUE if the ~~Nth-index~~-range of SA is defined with an ascending range; FALSE otherwise.

SA'ELEMENT Kind: Subtype.

Prefix: Any prefix SA that is appropriate for an array scalar object, or an alias thereof, or that denotes an array scalar subtype.

Result: If SA is an array scalar subtype, the result is the element subtype of SA. If SA is a an array scalar object, the result is the fully constrained element subtype that is the element subtype of SA., ~~together with constraints defining any index ranges that are determined by application of the rules of 5.3.2.2.~~

(If SA is an alias for a an-array scalar object, then the result is determined by the declaration of SA, not that of the object.)

LRM Annex C Syntax Summary Changes

Page 489 near middle of the page

Changes are shown in red font.

```
interface_function_specification ::= [§ 6.5.4]  
[ pure | impure ] function designator  
[ [ parameter ] ( formal_parameter_list ) ] return [identifier :] type_mark
```