



IEEE Standard VHDL Language Reference Manual

Amendment 1: Procedural Language Application Interface

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

1076cTM

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
5 September 2007

IEEE Std 1076cTM-2007
(Amendment to
IEEE Std 1076TM-2002)

IEEE Standard VHDL Language Reference Manual

Amendment 1: Procedural Language Application Interface

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 8 March 2007

IEEE SA-Standards Board

Abstract: This amendment adds a simulation runtime application interface (VHDL Programming Interface or VHPI) to the existing base standard IEEE Std 1076-2002.

Keywords: computer languages, electronic systems, hardware, hardware design, VHDL

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2007 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 5 September 2007. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

MagicDraw and No Magic, Inc., are registered trademarks of No Magic, Inc. in the United States and other countries.

Print: ISBN 0-7381-5523-3 SH95627
PDF: ISBN 0-7381-5524-1 SS95627

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied **“AS IS.”**

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

This introduction is not part of IEEE Std 1076c-2007, IEEE Standard VHDL Language Reference Manual—Amendment 1: Procedural Language Application Interface.

The VHSIC Hardware Description Language (VHDL) is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware.

This document is an amendment to IEEE Std 1076-2002. The main change is the addition of the VHDL Procedural Interface (VHPI), which allows external programs to access information about a VHDL description processed by a VHDL tool. The amendment also includes a minor change to clarify the rules for scope of architecture bodies.

The maintenance of the VHDL language standard is an ongoing process. The chair of the VHDL Analysis and Standardization Group (VASG), otherwise known as the 1076 Working Group, extends his gratitude to all who have participated in this amendment. In particular, the VHDL community owes a debt of gratitude to the members of the VHPI Task Force, who have devoted many hours of effort over an extended period to prepare this specification.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

The following individuals participated as members of the VHPI Task Force in the development of this amendment:

Françoise Martinolle, *Chair*
Peter J. Ashenden, *Technical Editor*

Stephen A. Bailey
Mikhail Brodsky
Patrick Bryant
Debi Dalio
Steven J. Dovich

Tristan Gingold
Frank Guerino
Amit Kohli
Jim W. Lewis
Paul Menchini
Vishwanath Raman

John J. Shields
David Townsend
John C. Willis
Debra Wimpey
Alex Zamfirescu

The following individuals participated as members of the P1076 Working Group:

Jim W. Lewis, *Chair*
Charles Swart, *Vice Chair*
Peter J. Ashenden, *Secretary*

John Aynsley
Stephen A. Bailey
Victor Berman
Françoise Martinolle

Egbert Molenkamp
Robert J. Myers
John P. Ries
Sukrit Shankar

John J. Shields
Ajayharsh Varikat
John C. Willis
Mark Zwolinski

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Peter J. Ashenden
Stephen A. Bailey
Victor Berman
Dennis B. Brophy
Juan C. Carreon
James S. Case
Danila Chernetsov
Keith Chow
Ernst Christen
S. M. Claassen
Tommy P. Cooper
Thomas J. Dineen
Steven J. Dovich
Sourav K. Dutta
George E. Economakos
John W. Fendrich

Randall C. Groves
M. K. Hashmi
Werner Hoelzl
Jim W. Lewis
G. L. Luri
Françoise Martinolle
Gary L. Michel
Michael D. Mirmak
Egbert Molenkamp
Robert J. Myers
Michael S. Newman
William R. Paulsen
Gregory D. Peterson
Ulrich Pohl
Iulian E. Profir

Vikram Punj
Vishwanath Raman
John P. Ries
Stephen C. Schwarm
John J. Shields
David W. Smith
Charles Swart
Lance G. Thompson
Mark-Rene Uchida
Joe T. Uchiyama
Alain Vachoux
Srinivasa R. Vemuru
John C. Willis
Oren Yuen
Janusz Zalewski
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 8 March 2007, it had the following membership:

Steve M. Mills, *Chair*
Robert M. Gown, *Vice Chair*
Don Wright, *Past Chair*
Judith Gorman, *Secretary*

Richard DeBlasio
Alex Gelman
William R. Goldbach
Arnold M. Greenspan
Joanna N. Guenin
Julian Forster*
Kenneth S. Hanus
William B. Hopf

Richard H. Hulett
Hermann Koch
Joseph L. Koepfinger*
John Kulick
David J. Law
Glenn Parsons
Ronald C. Petersen
Tom A. Prevost

Narayanan Ramachandran
Greg Ratta
Robby Robson
Anne-Marie Sahazizian
Virginia C. Sulzberger*
Malcolm V. Thaden
Richard L. Townsend
Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Alan H. Cookson, *NIST Representative*
Virginia C. Sulzberger, *Member/TAB Representative*

Don Messina
IEEE Standards Program Manager, Document Development

Michael D. Kipness
IEEE Standards Program Manager, Technical Program Development

Contents

0.	Overview of this standard	1
0.2	Structure and terminology of this standard	1
1.	Design entities and configurations	3
1.2	Architecture bodies	3
2.	Subprograms and packages	5
2.1	Subprogram declarations.....	5
5.	Specifications	7
5.1	Attribute specification.....	7
6.	Names.....	9
6.3	Selected names	9
8.	Sequential statements	11
8.5	Variable assignment statement.....	11
9.	Concurrent statements	13
9.6	Component instantiation statements	13
10.	Scope and visibility	17
10.1	Declarative region	17
10.2	Scope of declarations	17
10.3	Visibility.....	18
12.	Elaboration and execution	19
12.1	Elaboration of a design hierarchy	19
12.3	Elaboration of a declarative part	19
12.4	Elaboration of a statement part	19
12.5	Dynamic elaboration	20
12.6	Execution of a model	20
15.	VHDL Procedural Interface overview	31
15.1	Organization of the interface.....	31
15.2	Capability sets	32
15.3	Handles	34

16.	VHPI access functions	37
16.1	Information access functions	37
16.2	Property access functions	39
16.3	Access by name functions	40
17.	VHPI information model	41
17.1	Formal notation	41
17.2	Class inheritance hierarchy	42
17.3	Name properties	42
17.4	The stdUninstantiated package	56
17.5	The stdHierarchy package	59
17.6	The stdTypes package	68
17.7	The stdExpr package	70
17.8	The stdSpec package	74
17.9	The stdSubprograms package	76
17.10	The stdStmts package	78
17.11	The stdConnectivity package	84
17.12	The stdCallbacks package	88
17.13	The stdEngine package	89
17.14	The stdForeign package	89
17.15	The stdMeta package	90
17.16	The stdTool package	92
17.17	Application contexts	93
18.	VHPI tool execution	95
18.1	Registration phase	95
18.2	Analysis phase	101
18.3	Elaboration phase	101
18.4	Initialization phase	103
18.5	Simulation phase	103
18.6	Save phase	103
18.7	Restart phase	104
18.8	Reset phase	104
18.9	Termination phase	105
19.	VHPI callbacks	107
19.1	Callback functions	107
19.2	Callback reasons	108
20.	VHPI value access and update	121
20.1	Value structures and types	121
20.2	Reading object values	125
20.3	Formatting values	126
20.4	Updating object values	127
20.5	Scheduling transactions on drivers	131

21.	VHPI function reference	135
21.1	vhpi_assert	135
21.2	vhpi_check_error	136
21.3	vhpi_compare_handles	138
21.4	vhpi_control	139
21.5	vhpi_create	141
21.6	vhpi_disable_cb	143
21.7	vhpi_enable_cb	143
21.8	vhpi_format_value	144
21.9	vhpi_get	146
21.10	vhpi_get_cb_info	146
21.11	vhpi_get_data	147
21.12	vhpi_get_foreignf_info	149
21.13	vhpi_get_next_time	150
21.14	vhpi_get_phys	151
21.15	vhpi_get_real	152
21.16	vhpi_get_str	152
21.17	vhpi_get_time	153
21.18	vhpi_get_value	154
21.19	vhpi_handle	155
21.20	vhpi_handle_by_index	156
21.21	vhpi_handle_by_name	158
21.22	vhpi_is_printable	160
21.23	vhpi_iterator	161
21.24	vhpi_printf	162
21.25	vhpi_protected_call	163
21.26	vhpi_put_data	165
21.27	vhpi_put_value	167
21.28	vhpi_register_cb	168
21.29	vhpi_register_foreignf	169
21.30	vhpi_release_handle	171
21.31	vhpi_remove_cb	172
21.32	vhpi_scan	172
21.33	vhpi_schedule_transaction	173
21.34	vhpi_vprintf	176
	Annex B (informative) Glossary	179
	Annex F (informative) Bibliography	185
	Annex G (normative) VHPI header file	187
	Annex H (informative) Description of accompanying files	213

IEEE Standard VHDL Language Reference Manual

Amendment 1: Procedural Language Application Interface

NOTE—The editing instructions contained in this amendment define how to merge the material contained therein into the existing base standard and its amendments to form the comprehensive standard.

The editing instructions are shown in ***bold italic***. Four editing instructions are used: change, delete, insert, and replace. ***Change*** is used to make small corrections in existing text or tables. The editing instruction specifies the location of the change and describes what is being changed by using ~~striketrough~~ (to remove old material) or underscore (to add new material). ***Delete*** removes existing material. ***Insert*** adds new material without disturbing the existing material. Insertions may require renumbering. If so, renumbering instructions are given in the editing instruction. ***Replace*** is used to make changes in figures or equations by removing the existing figure or equation and replacing it with a new one. Editorial notes will not be carried over into future editions because the changes will be incorporated into the base standard.

0. Overview of this standard

0.2 Structure and terminology of this standard

0.2.3 Front matter, examples, notes, references, and annexes

Change the first paragraph in 0.2.3 as shown:

Prior to this subclause are several pieces of introductory material; following Clause-14 21 are some annexes and an index. The front matter, annexes (except Annex G), and index serve to orient and otherwise aid the user of this standard, but are not part of the definition of VHDL; Annex G, however, is normative.

1. Design entities and configurations

1.2 Architecture bodies

Change the second paragraph in 1.2 as shown:

The identifier defines the simple name of the architecture body; this simple name distinguishes architecture bodies associated with the same entity declaration. ~~For the purpose of interpreting the scope and visibility of the identifier (see 10.2 and 10.3), the declaration of the identifier is considered to occur after the final declarative item of the entity declarative part of the corresponding entity declaration.~~

2. Subprograms and packages

2.1 Subprogram declarations

2.1.1 Formal parameters

2.1.1.1 Constant and variable parameters

Change the second and third paragraphs in 2.1.1.1 as shown:

For a nonforeign subprogram having a parameter of a scalar type or an access type, or for a subprogram decorated with the 'FOREIGN attribute defined in package STANDARD for which the attribute value is of the form described in 18.1.3, the parameter is passed by copy. At the start of each call, if the mode is **in** or **inout**, the value of the actual parameter is copied into the associated formal parameter; it is an error if, after applying any conversion function or type conversion present in the actual part of the applicable association element (see 4.3.2.2), the value of the actual parameter does not belong to the subtype denoted by the subtype indication of the formal. After completion of the subprogram body, if the mode is **inout** or **out** and the associated actual parameter is not forced, the value of the formal parameter is copied back into the associated actual parameter; it is similarly an error if, after applying any conversion function or type conversion present in the formal part of the applicable association element, the value of the formal parameter does not belong to the subtype denoted by the subtype indication of the actual.

For a nonforeign subprogram having a parameter whose type is an array or record, an implementation may pass parameter values by copy, as for scalar types. In that case, after completion of the subprogram body, if the mode is **inout** or **out**, the value of each subelement of the formal parameter is only copied back to the corresponding subelement of the associated actual parameter if the subelement of the associated actual parameter is not forced. If a parameter of mode **out** is passed by copy, then the range of each index position of the actual parameter is copied in, and likewise for its subelements or slices. Alternatively, an implementation may achieve these effects by reference; that is, by arranging that every use of the formal parameter (to read or update its value) be treated as a use of the associated actual parameter throughout the execution of the subprogram call. The language does not define which of these two mechanisms is to be adopted for parameter passing, nor whether different calls to the same subprogram are to use the same mechanism. The execution of a subprogram is erroneous if its effect depends on which mechanism is selected by the implementation.

5. Specifications

5.1 Attribute specification

Change the seventh paragraph in 5.1 as shown:

The expression specifies the value of this attribute for each of the named entities inheriting the attribute as a result of this attribute specification. The type of the expression in the attribute specification must be the same as (or implicitly convertible to) the type mark in the corresponding attribute declaration. If the entity name list denotes an entity declaration, architecture body, or configuration declaration, then the expression is required to be locally static (see 7.4). Similarly, if the entity name list denotes a subprogram and the attribute designator denotes the 'FOREIGN' attribute defined in package STANDARD, then the expression is required to be locally static.

6. Names

6.3 Selected names

Change the sixth paragraph in 6.3 as shown:

An expanded name denotes a primary unit contained in a design library if the prefix denotes the library and the suffix is the simple name of a primary unit whose declaration is contained in that library. An expanded name denotes all primary units contained in a library if the prefix denotes the library and the suffix is the reserved word **all**. An expanded name is not allowed for a secondary unit, particularly for an architecture body.

Change the eighth paragraph in 6.3 as shown:

An expanded name denotes a named entity declared immediately within a named construct if the prefix denotes a construct that is an entity declaration, an architecture body, a subprogram declaration, a subprogram body, a block statement, a process statement, a generate statement, or a loop statement, and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within that construct. This form of expanded name is only allowed within the construct itself, or if the prefix denotes an entity declaration and the expanded name occurs within an architecture body corresponding to the entity declaration.

An expanded name denotes a named entity declared immediately within an architecture body if the prefix denotes the entity declaration corresponding to the architecture body and the suffix is the simple name, character literal, or operator symbol of a named entity whose declaration occurs immediately within the architecture body. This form of expanded name is only allowed within the architecture body.

Change the tenth paragraph in 6.3 as shown:

If, according to the visibility rules, there is at least one possible interpretation of the prefix of a selected name as the name of an enclosing entity declaration, architecture, subprogram, block statement, process statement, generate statement, or loop statement, or if there is at least one possible interpretation of the prefix of a selected name as the name of an object of a protected type, then the only interpretations considered are those of the immediately preceding ~~two~~three paragraphs. In this case, the selected name is always interpreted as an expanded name. In particular, no interpretations of the prefix as a function call are considered.

8. Sequential statements

8.5 Variable assignment statement

Change the fifth and sixth paragraphs in 8.5 as shown:

For the execution of a variable assignment whose target is a variable name, the variable name and the expression are first evaluated. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is an array (in which case the assignment involves a subtype conversion). Finally, the value of the expression becomes the new value of the variable. each subelement of the variable that is not forced is updated with the corresponding subelement of the expression. A design is erroneous if it depends on the order of evaluation of the target and source expressions of an assignment statement.

The execution of a variable assignment whose target is in the form of an aggregate proceeds in a similar fashion, except that each of the names in the aggregate is evaluated, and a subtype check is performed for each subelement or slice of the right-hand side value that corresponds to one of the names in the aggregate. The value of the subelement or slice of the right-hand side value then becomes the new value of the variable denoted by the corresponding name. For each variable denoted by a name corresponding to a subelement or slice of the right-hand side value, each subelement of the variable that is not forced is updated with the corresponding subelement of the subelement or slice of the right-hand side value.

Insert the following note after Note 3 in 8.5:

4—The value of a composite variable or of any element or slice of a composite variable is considered to have changed if any of the subelements of the variable, element or slice changes value.

9. Concurrent statements

9.6 Component instantiation statements

9.6.1 Instantiation of a component

Change the paragraphs in 9.6.1 as shown:

A component instantiation statement whose instantiated unit contains a name denoting a component is equivalent to a ~~triple pair~~ of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component declaration; the ~~intermediate-inner~~ block represents the design entity declaration to which the component is bound; and the ~~inner block represents the corresponding architecture body~~. Each is defined by a block statement.

The header of the block statement corresponding to the component declaration consists of the generic and port clauses (if present) that appear in the component declaration, followed by the generic map and port map aspects (if present) that appear in the corresponding component instantiation statement. The meaning of any identifier appearing in the header of this block statement is associated with the corresponding occurrence of the identifier in the generic clause, port clause, generic map aspect, or port map aspect, respectively. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity declaration.

The header of the block statement corresponding to the design entity declaration consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the binding indication that binds the component instance to that design entity declaration. The declarative part of the block statement corresponding to the design entity declaration consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the design entity declaration consists of the concurrent statements from the entity statement part, followed by a nested block statement corresponding to the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this ~~intermediate~~ block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body, respectively.

The header of the block statement corresponding to the architecture body is empty. The declarative part of the block statement corresponding to the architecture body consists of the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the architecture body consists of the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the architecture body.

Change the example code in 9.6.1 as shown:

```
C: block                                -- Component block.
    port (A,B : inout BIT);              -- Local ports.
    port map (A => S1, B => S2);          -- Actual/local binding.
begin
    X: block                            -- Design entity block.
        port (P1, P2 : inout BIT);      -- Formal ports.
        port map (P1 => A, P2 => B);      -- Local/formal binding.
        constant Delay: Time := 1 ms;   -- Entity declarative item.
```

~~_____~~	~~signal P3: BIT;~~	~~_____~~	~~-- Architecture declarative item.~~
	begin		
	CheckTiming (P1, P2, 2*Delay);		-- Entity statement.
~~_____~~	~~Y: block~~		
~~_____~~	~~signal P3: Bit;~~	~~_____~~	~~-- Architecture declarative item.~~
~~_____~~	~~begin~~		
	— P3 <= P1 after Delay;		-- Architecture statements.
	— P2 <= P3 after Delay;		
	— B: block -- Internal block hierarchy.		
	— .		
	— .		
	— .		
	— begin		
	— .		
	— .		
	— .		
	— end block;		
~~_____~~	~~end block Y;~~		
	end block X ;		
	end block C;		

9.6.2 Instantiation of a design entity

Change the paragraphs in 9.6.2 as shown:

A component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration is equivalent to a ~~triple-pair~~ of nested block statements that couple the block hierarchy in the containing design unit to a unique copy of the block hierarchy contained in another design unit (i.e., the subcomponent). The outer block represents the component instantiation statement; the ~~intermediate-inner~~ block represents the design entity declaration to which the instance is bound; and the ~~inner block~~ represents the corresponding architecture body. Each is defined by a block statement.

The header of the block statement corresponding to the component instantiation statement is empty, as is the declarative part of this block statement. The statement part of the block statement corresponding to the component declaration consists of a nested block statement corresponding to the design entity declaration.

The header of the block statement corresponding to the design entity declaration consists of the generic and port clauses (if present) that appear in the entity declaration that defines the interface to the design entity, followed by the generic map and port map aspects (if present) that appear in the component instantiation statement that binds the component instance to a copy of that design entity. The declarative part of the block statement corresponding to the design entity declaration consists of the declarative items from the entity declarative part, followed by the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the design entity declaration consists of the concurrent statements from the entity statement part, followed by a nested block statement corresponding to the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the entity declaration or architecture body, respectively.

The header of the block statement corresponding to the architecture body is empty. The declarative part of the block statement corresponding to the architecture body consists of the declarative items from the declarative part of the corresponding architecture body. The statement part of the block statement corresponding to the architecture body consists of the concurrent statements from the statement part of the corresponding architecture body. The meaning of any identifier appearing anywhere in this block statement is that associated with the corresponding occurrence of the identifier in the architecture body.

Change the first occurrence of the following example code in 9.6.2 as shown:

```

C: block                                -- Instance block.
  begin
    X: block                            -- Design entity block.
      port (P1, P2: inout BIT);        -- Entity declaration ports.
      port map (P1 => S1, P2 => S2);    -- Instantiation statement port map.
      constant Delay: DELAY_LENGTH := 1 ms; -- Entity declarative items.
      use WORK.TimingChecks.all;
      signal P3: BIT;                -- Architecture declarative item.
    begin
      CheckTiming (P1, P2, 2*Delay);    -- Entity statement.
    Y: block
    signal P3: BIT;                -- Architecture declarative item.
    begin
    P3 <= P1 after Delay;        -- Architecture statements.
    P2 <= P3 after Delay;
    B: block
      .
      .
      .
    begin
      .
      .
      .
    end block B;
    end block Y;
  end block X;
end block C;

```

Change the second occurrence of the following example code in 9.6.2 as shown:

```

C: block                                -- Instance block.
  begin
    X: block                            -- Design entity block.
      port (P1, P2: inout BIT);        -- Entity declaration ports.
      port map (P1 => S1, P2 => S2);    -- Instantiation statement port map.
      constant Delay: DELAY_LENGTH := 1 ms; -- Entity declarative items.
      use WORK.TimingChecks.all;
      signal P3: BIT;                -- Architecture declarative item.
    begin
      CheckTiming (P1, P2, 2*Delay);    -- Entity statement.
    Y: block
    signal P3: BIT;                -- Architecture declarative item.
    begin
    P3 <= P1 after Delay;        -- Architecture statements.
    P2 <= P3 after Delay;
    B: block
      .
      .
      .
    begin
      .
      .

```

```
—      •  
—      end block B;  
—      end block Y;  
      end block X;  
end block C;
```

10. Scope and visibility

10.1 Declarative region

Change list item a) in 10.1 as shown:

- a) An entity declaration, together with a corresponding architecture body

Delete list item b) in 10.1 as shown and renumber the list:

- b) ~~An architecture body~~

Delete the paragraph before the note in 10.1 as shown:

~~The declarative region associated with an architecture body is considered to occur immediately within the declarative region associated with the entity declaration corresponding to the given architecture body.~~

Replace the note in 10.1 as shown:

~~NOTE—The fact that an architecture body has an associated root declarative region does not mean that the declarative region associated with the architecture is directly within the associated root declarative region. Instead, the declarative region associated with the corresponding entity declaration surrounds the declarative region associated with the architecture.~~

~~NOTE—An architecture body, though a declaration, does not occur immediately within any declarative region.~~

10.2 Scope of declarations

Change the second paragraph in 10.2 as shown:

The scope of a declaration, except for an architecture body, extends from the beginning of the declaration to the end of the immediately closing declarative region; the scope of an architecture body extends from the beginning to the end of the architecture body. In either case, this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations in the following list, the scope of the declaration extends beyond the immediate scope:

Change list item h) in 10.2 as shown:

- h) A declaration that occurs immediately within a protected type declaration.

Insert the following list item after list item h) in 10.2:

- i) An architecture body

Change the third paragraph in 10.2 as shown:

In the absence of a separate subprogram declaration, the subprogram specification given in the subprogram body acts as the declaration, and rule c) applies also in such a case. In each of these cases except i), the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration.

Change the note in 10.2 as shown and add Note 2:

NOTES

1—These scope rules apply to all forms of declaration. In particular, they apply also to implicit declarations and to named ~~design~~ primary units.

2—The scope of an entity declaration includes an associated architecture body, if any. Thus, the entity name may be used within the architecture body as the prefix of an expanded name denoting a declaration that occurs immediately within the entity declaration or the architecture body. The scope of an architecture body does not include the corresponding entity declaration. Thus, the entity cannot use an expanded name to refer to the architecture body nor to any declaration within the architecture body.

10.3 Visibility

Insert list item e) after list item d) in 10.3 as shown and renumber the list:

- e) For an architecture body associated with a given entity declaration: at the place of an architecture identifier (between the parentheses) in the second form of an instantiated unit in a component instantiation statement.

Change the paragraph following the lettered list in 10.3 as shown:

Finally, within the declarative region associated with a construct other than a record type declaration or a protected type, any declaration that occurs immediately within the region and that also occurs textually within the construct is visible by selection at the place of the suffix of an expanded name whose prefix denotes the construct. Similarly, within an architecture body, any declaration that occurs immediately within the architecture body or the corresponding entity declaration is visible by selection at the place of the suffix of an expanded name whose prefix denotes the entity declaration.

12. Elaboration and execution

12.1 Elaboration of a design hierarchy

Insert the following paragraph after the first paragraph of 12.1:

At the beginning of the elaboration of a design hierarchy, every registered and enabled `vhpicbStartOfElaboration` callback is executed. Once the elaboration of a given design hierarchy is complete, every registered and enabled `vhpicbEndOfElaboration` callback is executed.

12.3 Elaboration of a declarative part

Change list items a) and b) in 12.3 as shown:

- a) The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN' attribute defined in package STANDARD (see 5.1 and 14.2) and for which the value of the attribute is not of the form described in 18.1.3.
- b) The architecture declarative part of a design entity whose architecture is decorated with the 'FOREIGN' attribute defined in package STANDARD and for which the value of the attribute is not of the form described in 18.1.3.

Insert the following note after the note in 12.3, renumber the first note as 1—, and add the word NOTES above Note 1:

2—A function called in an expression evaluated during elaboration of a declarative item may be a foreign function.

12.4 Elaboration of a statement part

Change the second paragraph in 12.4 as shown:

~~For this case, the statements are not elaborated; instead, the design entity is subject to implementation dependent elaboration.~~

For this case, there are two subcases:

- If the value of the attribute is of the form described in 18.1.3, the statements are not elaborated; instead, the elaboration function of the foreign model is invoked, as described in 18.3, at the point where elaboration of the statements of the block statement corresponding to the architecture body would otherwise occur.
- Otherwise, the statements are not elaborated; instead, the design entity is subject to implementation-dependent elaboration.

12.4.3 Component instantiation statements

Change 12.4.3 as shown:

Elaboration of a component instantiation statement that instantiates a component declaration has no effect unless the component instance is either fully bound to a design entity defined by an entity declaration and architecture body or bound to a configuration of such a design entity. If a component instance is so bound, then elaboration of the corresponding component instantiation statement consists of the elaboration of the implied block statement representing the component instance and (within that block) the implied block

statements representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.1.

Elaboration of a component instantiation statement whose instantiated unit denotes either a design entity or a configuration declaration consists of the elaboration of the implied block statement representing the component instantiation statement and (within that block) the implied block statements representing the design entity to which the component instance is bound. The implied block statements are defined in 9.6.2.

12.5 Dynamic elaboration

Change list item b) in 12.5 as shown:

- b) Execution of a subprogram call involves the elaboration of the parameter interface list of the corresponding subprogram declaration; this involves the elaboration of each interface declaration to create the corresponding formal parameters. Actual parameters are then associated with formal parameters. Next, if the subprogram is a method of a protected type (see 3.5.1) or an implicitly declared file operation (see 3.4.1), the elaboration blocks (suspends execution while retaining all state), if necessary, until exclusive access to the object denoted by the prefix of the method or to the file object denoted by the file parameter of the file operation is secured. Finally, if the designator of the subprogram is not decorated with the 'FOREIGN' attribute defined in package STANDARD, the declarative part of the corresponding subprogram body is elaborated and the sequence of statements in the subprogram body is executed. If the designator of the subprogram is decorated with the 'FOREIGN' attribute defined in package STANDARD, ~~then the subprogram body is subject to implementation-dependent elaboration and execution. there are two cases:~~
 - If the value of the attribute is of the form described in 18.1.3, the declarative part of the corresponding subprogram body is not elaborated nor is the sequence of statements in the subprogram body executed; instead, the execution function of the foreign model is invoked, as described in 18.3.1.
 - Otherwise, the subprogram body is subject to implementation-dependent elaboration and execution.

12.6 Execution of a model

Change the third paragraph in 12.6 as shown:

For any given signal that is explicitly declared within a model, the kernel process contains ~~a variable~~ variables representing the driving value and current value of that signal. Any evaluation of a name denoting a given signal retrieves the current value of the corresponding variable in the kernel process. During simulation, the kernel process updates these variables from time to time, based upon the current values of sources of the corresponding signal.

12.6.1 Drivers

Change the third and fourth paragraphs in 12.6.1 as shown:

A driver always contains at least one transaction. The initial contents of a driver associated with a given signal are defined by the default value associated with the signal (see 4.3.1.2). The kernel process contains a variable representing the current value of the driver. The initial value of the variable is the value component of the initial transaction of the driver.

~~For any driver, there is exactly one transaction whose time component is not greater than the current simulation time. The current value of the driver is the value component of this transaction. If, as the result of the advance of time, the current time becomes equal to the time component of the next transaction, then the~~

first transaction is deleted from the projected output waveform and the next becomes the current value of the driver.

For any driver, if, as the result of the advance of time, the current time becomes equal to the time component of the second transaction of the driver, the first transaction is deleted from the projected output waveform, and what was the second transaction becomes the first transaction. Then, or if a force or deposit is scheduled for the driver, the variable containing the current value of the driver is updated as follows:

- If a force is scheduled for the driver, the driver becomes forced and the variable containing the current value of the driver is updated with the force value for the driver.
- If the driver is forced and no force is scheduled for the driver, the variable containing the current value of the driver is unchanged from its previous value.
- If a deposit is scheduled for the driver and the driver is not forced, the variable containing the current value of the driver is updated with the deposit value for the driver.
- Otherwise, the variable containing the current value of the driver is updated with the value component of the first transaction of the driver.

When this action occurs on a driver, any registered and enabled `vhpiCbTransaction` callbacks associated with the given driver are executed. Moreover, if the current value of the driver changes as a result of this action, any registered and enabled `vhpiCbValueChange` callbacks associated with the given driver are executed.

12.6.2 Propagation of signal values

Change the first paragraph and the following list in 12.6.2 as shown:

As simulation time advances, the transactions in the projected output waveform of a given driver (see 12.6.1) will each, in succession, become the value of the driver. When a driver acquires a new value in this way or as a result of a force or deposit scheduled for the driver, regardless of whether the new value is different from the previous value, that driver is said to be *active* during that simulation cycle. For the purposes of defining driver activity, a driver acquiring a value from a null transaction is assumed to have acquired a new value. A signal is said to be *active* during a given simulation cycle if

- One of its sources is active
- One of its subelements is active
- The signal is named in the formal part of an association element in a port association list and the corresponding actual is active
- The signal is a subelement of a resolved signal and the resolved signal is active.
- A force or a deposit is scheduled for the signal
- The signal is a subelement of another signal for which a force or a deposit is scheduled

Change the fifth paragraph in 12.6.2 as shown:

The kernel process determines two values for certain signals during ~~any given simulation cycle~~ certain simulation cycles. The *driving value* of a given signal is the value that signal provides as a source of other signals. The *effective value* of a given signal is the value obtainable by evaluating a reference to the signal within an expression. The driving value and the effective value of a signal are not always the same, especially when resolution functions and conversion functions or type conversions are involved in the propagation of signal values.

Change the eighth, ninth, and tenth paragraphs in 12.6.2 as shown:

The driving value of any basic signal *S* is determined as follows:

- If *S* has no source, then the driving value of *S* is given by the default value associated with *S* (see 4.3.1.2).
- If *S* has one source that is a driver and *S* is not a resolved signal (see 4.3.1.2), then the driving value of *S* is the current value of that driver.
- If *S* has one source that is a port and *S* is not a resolved signal, then the driving value of *S* is the driving value of the formal part of the association element that associates *S* with that port (see 4.3.2.2). The driving value of a formal part is obtained by evaluating the formal part as follows: If no conversion function or type conversion is present in the formal part, then the driving value of the formal part is the driving value of the signal denoted by the formal designator. Otherwise, the driving value of the formal part is the value obtained by applying either the conversion function or type conversion (whichever is contained in the formal part) to the driving value of the signal denoted by the formal designator.
- If *S* is a resolved signal and has one or more sources, then the driving values of the sources of *S* are examined. It is an error if any of these driving values is a composite where one or more subelement values are determined by the null transaction (see 8.4.1) and one or more subelement values are not determined by the null transaction. If *S* is of signal kind **register** and all the sources of *S* have values determined by the null transaction, then the driving value of *S* is unchanged from its previous value. Otherwise, the driving value of *S* is obtained by executing the resolution function associated with *S*, where that function is called with an input parameter consisting of the concatenation of the driving values of the sources of *S*, with the exception of the value of any source of *S* whose current value is determined by the null transaction.

The driving value of any signal *S* that is not a basic signal is determined as follows:

- If *S* is a subelement of a resolved signal *R*, the driving value of *S* is the corresponding subelement value of the driving value of *R*.
- Otherwise (*S* is a nonresolved, composite signal), the driving value of *S* is equal to the aggregate of the driving values of each of the basic signals that are the subelements of *S*.

The driving value of any signal *S* is determined as follows:

- a) If a driving-value force is scheduled for *S* or for a signal of which *S* is a subelement, *S* becomes driving-value forced and the driving value of *S* is the driving force value for *S* or the element of the driving force value for the signal of which *S* is a subelement, as appropriate.
- b) If *S* is driving-value forced and no driving-value force is scheduled for *S* or for a signal of which *S* is a subelement, the driving value of *S* is unchanged from its previous value.
- c) If a driving-value deposit is scheduled for *S* or for a signal of which *S* is a subelement and *S* is not driving-value forced, the driving value of *S* is the driving deposit value for *S* or the element of the driving deposit value for the signal of which *S* is a subelement, as appropriate.
- d) Otherwise, if *S* is a basic signal:
 - If *S* has no source, then the driving value of *S* is given by the default value associated with *S* (see 4.3.1.2).
 - If *S* has one source that is a driver and *S* is not a resolved signal (see 4.3.1.2), then the driving value of *S* is the current value of that driver.
 - If *S* has one source that is a port and *S* is not a resolved signal, then the driving value of *S* is the driving value of the formal part of the association element that associates *S* with that port (see 4.3.2.2). The driving value of a formal part is obtained by evaluating the formal part as follows: If no conversion function or type conversion is present in the formal part, then the driving value of the formal part is the driving value of the signal denoted by the formal designator. Otherwise, the driving value of the formal part is the value obtained by applying either the conversion function or type conversion (whichever is contained in the formal part) to the driving value of the signal denoted by the formal designator.

- If S is a resolved signal and has one or more sources, then the driving values of the sources of S are examined. It is an error if any of these driving values is a composite where one or more subelement values are determined by the null transaction (see 8.4.1) and one or more subelement values are not determined by the null transaction. If S is of signal kind **register** and all the sources of S have values determined by the null transaction, then the driving value of S is unchanged from its previous value. Otherwise, the driving value of S is obtained by executing the resolution function associated with S, where that function is called with an input parameter consisting of the concatenation of the driving values of the sources of S, with the exception of the value of any source of S whose current value is determined by the null transaction.
- e) Alternatively, if S is not a basic signal:
 - If S is a subelement of a resolved signal R, the driving value of S is the corresponding subelement value of the driving value of R.
 - Otherwise (S is a nonresolved, composite signal), the driving value of S is equal to the aggregate of the driving values of each of the basic signals that are the subelements of S.

For a scalar signal S, the *effective value* of S is determined in the following manner:

- ~~If S is a signal declared by a signal declaration, a port of mode **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.~~
- ~~If S is a connected port of mode **in** or **inout**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S (see 4.3.2.2). The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.~~
- ~~If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S (see 4.3.1.2).~~
- a) If an effective-value force is scheduled for S or for a signal of which S is a subelement, S becomes effective-value forced and the effective value of S is the effective force value for S or the element of the effective force value for the signal of which S is a subelement, as appropriate.
- b) If S is effective-value forced and no effective-value force is scheduled for S or for a signal of which S is a subelement, the effective value of S is unchanged from its previous value.
- c) If an effective-value deposit is scheduled for S or for a signal of which S is a subelement and S is not effective-value forced, the effective value of S is the effective deposit value for S or the element of the effective deposit value for the signal of which S is a subelement, as appropriate.
- d) Otherwise:
 - If S is a signal declared by a signal declaration, a port of mode **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.
 - If S is a connected port of mode **in** or **inout**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S (see 4.3.2.2). The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.
 - If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S (see 4.3.1.2).

Change the thirteenth paragraph in 12.6.2 as shown:

In order to update a signal during a given simulation cycle, the kernel process first determines the driving and effective values of that signal. The kernel process then updates the variable containing the driving value with the newly determined driving value. The kernel also updates the variable containing the current value of the signal with the newly determined effective value, as follows:

Change the fourteenth paragraph and insert the new paragraph in 12.6.2 as follows:

Updating a signal *S* of type *T* is said to *change* the current value of *S* if and only if the expression “*S* = *S*'Delayed” evaluates to False, where the “=” operator in the expression is the predefined “=” on type *T*. If updating a signal causes the current value of that signal to change, then an *event* is said to have occurred on the signal, unless the update occurs by application of the *vhpi_put_value* function with an update mode of *vhpiDeposit* or *vhpiForce* to an object that represents the signal. This definition applies to any updating of a signal, whether such updating occurs according to the above rules or according to the rules for updating implicit signals given in 12.6.3. The occurrence of an event will cause the resumption and subsequent execution of certain processes during the simulation cycle in which the event occurs, if and only if those processes are currently sensitive to the signal on which the event has occurred.

Each time a signal *S* is updated, any registered and enabled *vhpiCbTransaction* callbacks associated with *S* are executed. Each time there is an event on a signal *S*, any registered and enabled *vhpiCbValueChange* callbacks associated with *S* are executed.

Delete the fifteenth paragraph in 12.6.2. as shown:

~~For any signal other than one declared with the signal kind **register**, the driving and effective values of the signal are determined and the current value of that signal is updated as described above in every simulation cycle. A signal declared with the signal kind **register** is updated in the same fashion during every simulation cycle except those in which all of its sources have current values that are determined by null transactions.~~

Insert the new paragraph before the last paragraph in 12.6.2 as follows:

For any signal that is part of a given net, the driving and effective values of the signal are determined and the variables containing the driving value and current value of that signal are updated as described above in those simulation cycles in which any driver or signal on the net is active.

Insert the following note at the end of 12.6.2:

9—If a net includes an implicitly declared GUARD signal, the drivers of signals referred to in the corresponding guard expression determine the value of the GUARD signal. Hence, those drivers are part of the net, and when any of the drivers are active, the signals that are part of the net are updated.

12.6.3 Updating implicit signals

Change the second paragraph in 12.6.3 as shown:

For any implicit signal GUARD, the current value of the signal is modified if and only if the corresponding guard expression contains a reference to a signal *S* and if *S* is active during the current simulation cycle. In such a case, the implicit signal GUARD is updated by evaluating the corresponding guard expression and assigning the result of that evaluation to the variable representing the current value of the signal. Whenever an implicit signal GUARD is updated, any registered and enabled *vhpiCbTransaction* callbacks associated with the given signal are executed.

Change the fourth paragraph in 12.6.3 as shown:

If an event has occurred on signal *S*, then *S*'Stable(*T*) is updated by assigning the value FALSE to the variable representing the current value of *S*'Stable(*T*), and the driver of *S*'Stable(*T*) is assigned the waveform TRUE after *T*. Otherwise, if the driver of *S*'Stable(*T*) is active, then *S*'Stable(*T*) is updated by assigning the current value of the driver to the variable representing the current value of *S*'Stable(*T*). Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form *S*'Stable(*T*) is updated, any registered and enabled *vhpiCbTransaction* callbacks associated with the given signal are executed.

Change the sixth, seventh, and eighth paragraphs in 12.6.3 as shown:

If signal S is active, then $S'Quiet(T)$ is updated by assigning the value FALSE to the variable representing the current value of $S'Quiet(T)$, and the driver of $S'Quiet(T)$ is assigned the waveform TRUE **after** T . Otherwise, if the driver of $S'Quiet(T)$ is active, then $S'Quiet(T)$ is updated by assigning the current value of the driver to the variable representing the current value of $S'Quiet(T)$. Otherwise, neither the variable nor the driver is modified. Whenever a signal of the form $S'Quiet(T)$ is updated, any registered and enabled `vhpicbTransaction` callbacks associated with the given signal are executed.

Finally, for any implicit signal $S'Transaction$, the current value of the signal is modified if and only if S is active. If signal S is active, then $S'Transaction$ is updated by assigning the value of the expression (**not** $S'Transaction$) to the variable representing the current value of $S'Transaction$. At most one such assignment will occur during any given simulation cycle. Whenever a signal of the form $S'Transaction$ is updated, any registered and enabled `vhpicbTransaction` callbacks associated with the given signal are executed.

For any implicit signal $S'Delayed(T)$, the signal is not updated by the kernel process. Instead, it is updated by constructing an equivalent process (see 14.1) and executing that process.

Each time there is an event on a signal S , where S is any one of

- an implicit signal GUARD
- $P'Stable(T)$, for any prefix P and any time T
- $P'Quiet(T)$, for any prefix P and any time T
- $P'Transaction$, for any prefix P

any registered and enabled `vhpicbValueChange` callbacks associated with S are executed.

Change the title of 12.6.4 as shown:

12.6.4 The simulation cycle

Delete the paragraphs after the first paragraph in 12.6.4 and insert 12.6.4.1 and 12.6.4.2 as shown:

At the beginning of initialization, the current time, T_e , is assumed to be 0 ns.

The initialization phase consists of the following steps:

- The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.
- The value of each implicit signal of the form $S'Stable(T)$ or $S'Quiet(T)$ is set to True. The value of each implicit signal of the form $S'Delayed(T)$ is set to the initial value of its prefix, S .
- The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.
- Each nonpostponed process in the model is executed until it suspends.
- Each postponed process in the model is executed until it suspends.
- The time of the next simulation cycle (which in this case is the first simulation cycle), T_R , is calculated according to the rules of step f) of the simulation cycle, below.

A simulation cycle consists of the following steps:

- a) The current time, T_c , is set equal to T_n . Simulation is complete when $T_n = \text{TIME'HIGH}$ and there are no active drivers or process resumptions at T_n .
- b) Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
- c) Each implicit signal in the model is updated. (Events may occur on signals as a result.)
- d) For each process, P , if P is currently sensitive to a signal, S , and if an event has occurred on S in this simulation cycle, then P resumes.
- e) Each nonpostponed process that has resumed in the current simulation cycle is executed until it suspends.
- f) If the break flag is set, the time of the next simulation cycle, T_n , is determined by setting it to the earliest of
 - 1) TIME'HIGH ,
 - 2) The next time at which a driver becomes active, or
 - 3) The next time at which a process resumes.
 If $T_n = T_c$, then the next simulation cycle (if any) will be a *delta cycle*.
- g) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise, each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends. Then T_n is recalculated according to the rules of step f). It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

NOTES

- 1—The initial value of any implicit signal of the form S' Transaction is not defined.
- 2—Updating of explicit signals is described in 12.6.2; updating of implicit signals is described in 12.6.3.
- 3—When a process resumes, it is added to one of two sets of processes to be executed (the set of postponed processes and the set of nonpostponed processes). However, no process actually begins to execute until all signals have been updated and all executable processes for this simulation cycle have been identified. Nonpostponed processes are always executed during step c) of every simulation cycle, while postponed processes are executed during step g) of every simulation cycle that does not immediately precede a delta cycle.
- 4—The second and third steps of the initialization phase and steps b) and c) of the simulation cycle may occur in interleaved fashion. This interleaving may occur because the implicit signal GUARD may be used as the prefix of another implicit signal; moreover, implicit signals may be associated as actuals with explicit signals, making the value of an explicit signal a function of an implicit signal.

12.6.4.1 Initialization

At the beginning of initialization, the current time, T_c , is assumed to be 0 ns.

The initialization phase consists of the following steps:

- a) Each registered and enabled `vhpiCbStartofInitialization` callback is executed.
- b) Each registered and enabled `vhpiCbStartOfNextCycle` and `vhpiCbRepStartOfNextCycle` callback is executed.
- c) The driving value and the effective value of each explicitly declared signal are computed, and the variables representing the driving value and current value of the signal are set to the driving value and effective value, respectively. The current value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation. If a force or deposit was scheduled for any driver or signal, the force or deposit is no longer scheduled for the driver or signal.

- d) The value of each implicit signal of the form S'Stable(T) or S'Quiet(T) is set to True. The value of each implicit signal of the form S'Delayed(T) is set to the initial value of its prefix, S.
- e) The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.
- f) Each registered and enabled vhpCbStartOfProcesses and vhpCbRepStartOfProcesses callback is executed.
- g) Each nonpostponed process in the model is executed until it suspends. Each registered and enabled vhpCbSuspend callback associated with a nonpostponed process is executed.
- h) For each elaborated instance of a registered foreign architecture, the corresponding execution function is invoked.
- i) Each registered and enabled vhpCbEndOfProcesses and vhpCbRepEndOfProcesses callback is executed.
- j) Each registered and enabled vhpCbStartOfPostponed and cbRepStartOfPostponed callback is executed.
- k) Each postponed process in the model is executed until it suspends.
- l) Each registered and enabled vhpCbSuspend callback associated with a postponed process is executed.
- m) The time of the next simulation cycle (which in this case is the first simulation cycle), T_n , is calculated according to the rules of step f) of the simulation cycle in 12.6.4.2.
- n) If the VHDL tool executing the initialization phase has requested a model save that has not yet been satisfied, the model is saved as described in 18.6.
- o) Each registered and enabled vhpCbEndOfInitialization callback is executed.

NOTES

1—The initial value of any implicit signal of the form S'Transaction is not defined.

2—Updating of explicit signals is described in 12.6.2; updating of implicit signals is described in 12.6.3.

3—The callbacks vhpCbResume and vhpCbRepResume are not executed during initialization as processes do not resume during initialization.

12.6.4.2 The simulation cycle

A simulation cycle consists of the following steps:

- a) If $T_n = \text{TIME'HIGH}$ and there are no active drivers, process resumptions, or registered and enabled vhpCbAfterDelay, vhpCbRepAfterDelay, vhpCbTimeout, or vhpCbRepTimeout callbacks to occur at T_n , the following actions occur in the indicated order. Otherwise, the remainder of this step is skipped.
 - 1) Each registered and enabled vhpCbQuiescence callback is executed.
 - 2) T_c is set equal to T_n .
 - 3) If $T_n = \text{TIME'HIGH}$ and there are no active drivers, process resumptions, or registered and enabled vhpCbAfterDelay, vhpCbRepAfterDelay, vhpCbTimeout, or vhpCbRepTimeout callbacks to occur at T_n , simulation is complete.
- b) The following actions occur in the indicated order:
 - 1) If the current simulation cycle is not a delta cycle, each registered and enabled vhpCbNextTimeStep and vhpCbRepNextTimeStep callback is executed.
 - 2) Each registered and enabled vhpCbStartOfNextCycle and vhpCbRepStartOfNextCycle callback is executed.

- 3) Each registered and enabled `vhpiCbAfterDelay` and `vhpiCbRepAfterDelay` callback is executed.
- c) Each active driver in the model is updated, then each active explicit signal in the model is updated. (Events may occur on signals as a result.) If a force or deposit was scheduled for any driver or signal, the force or deposit is no longer scheduled for driver or signal.
- d) Each implicit signal in the model is updated. (Events may occur on signals as a result.)
- e) The following actions occur in the indicated order:
 - 1) Each registered and enabled `vhpiCbStartOfProcesses` and `vhpiCbRepStartOfProcesses` callback is executed. If an event has occurred on a signal *S* in this simulation cycle, then each registered and enabled `vhpiCbSensitivity` callback associated with *S* is executed.
 - 2) For each process, *P*, if *P* is currently sensitive to a signal, *S*, and if an event has occurred on *S* in this simulation cycle, then *P* resumes.
 - 3) For each nonpostponed process *P* that has resumed in the current simulation cycle, each registered and enabled `vhpiCbResume` callback associated with *P* is executed.
 - 4) Each nonpostponed process that has resumed in the current simulation cycle is executed until it suspends, and each registered and enabled `vhpiCbTimeOut` and `vhpiCbRepTimeOut` callback whose triggering condition is met is executed.
 - 5) For each nonpostponed process *P* that has resumed in the current simulation cycle, each registered and enabled `vhpiCbSuspend` callback associated with *P* is executed.
 - 6) Each registered and enabled `vhpiCbEndOfProcesses` and `vhpiCbRepEndOfProcesses` callback is executed.
- f) The time of the next simulation cycle, T_n , is determined by setting it to the earliest of
 - 1) `TIME'HIGH`,
 - 2) The next time at which a driver becomes active,
 - 3) The next time at which a process resumes, or
 - 4) The next time at which a registered and enabled `vhpiCbAfterDelay`, `vhpiCbRepAfterDelay`, `vhpiCbTimeOut`, or `vhpiCbRepTimeOut` callback is to occur.

If $T_n = T_c$, then the next simulation cycle (if any) will be a *delta cycle*.
- g) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise, the following actions occur in the indicated order:
 - 1) Each registered and enabled `vhpiCbLastKnownDeltaCycle` and `vhpiCbRepLastKnownDeltaCycle` callback is executed. T_n is recalculated according to the rules of step f).
 - 2) If the next simulation cycle will be a delta cycle, the remainder of this step is skipped.
 - 3) Each registered and enabled `vhpiCbStartOfPostponed` and `vhpiCbRepStartOfPostponed` callback is executed.
 - 4) For each postponed process *P*, if *P* has resumed but has not been executed since its last resumption, then each registered and enabled `vhpiCbResume` callback associated with *P* is executed.
 - 5) Each postponed process that has resumed but has not been executed since its last resumption is executed until it suspends.
 - 6) For each postponed process *P*, if *P* has executed in the previous step, then each registered and enabled `vhpiCbSuspend` callback associated with *P* is executed.
 - 7) T_n is recalculated according to the rules of step f).
 - 8) Each registered and enabled `vhpiCbEndOfTimeStep` and `vhpiCbRepEndOfTimeStep` callback is executed. It is an error if the execution of any postponed process or any callback executed in step g) causes a delta cycle to occur immediately after the current simulation cycle.

- h) If the VHDL tool executing the simulation cycle has requested a model save that has not yet been satisfied, the model is saved as described in 18.6.

Immediately prior to the execution of the first simulation cycle, each registered and enabled `vhpicbStartOfSimulation` callback is executed. Immediately subsequent to the execution of the final simulation cycle (i.e., when simulation is complete), each registered and enabled `vhpicbEndOfSimulation` callback is executed.

NOTES

1—Updating of explicit signals is described in 12.6.2; updating of implicit signals is described in 12.6.3.

2—When a process resumes, it is added to one of two sets of processes to be executed (the set of postponed processes and the set of nonpostponed processes). However, no process actually begins to execute until all signals have been updated and all executable processes for this simulation cycle have been identified. Nonpostponed processes are always executed during step e) of every simulation cycle, while postponed processes are executed during step g) of every simulation cycle that does not immediately precede a delta cycle.

3—Steps c), d), and e) of the initialization phase and steps c) and d) of the simulation cycle may occur in interleaved fashion. This interleaving may occur because the implicit signal `GUARD` may be used as the prefix of another implicit signal; moreover, implicit signals may be associated as actuals with explicit signals, making the value of an explicit signal a function of an implicit signal.

4—The `vhpicbQuiescence` callback cannot schedule any events that would cause a delta cycle. The `vhpicbEndOfTimeStep` and `vhpicbRepEndOfTimeStep` callbacks cannot schedule any events or `vhpicbAfterDelay` callbacks.

Insert the following clauses after Clause 14:

15. VHDL Procedural Interface overview

The VHDL Procedural Interface (VHPI) is an application-programming interface to VHDL tools that allows programmatic access to a VHDL model during its analysis, elaboration, and execution. The VHPI is described in this clause, subsequent clauses through to Clause 21, and Annex G.

15.1 Organization of the interface

The VHPI consists of two aspects:

- An information model that represents the topology and state of a VHDL model
- A number of functions that operate on the information model to access or affect the state of the VHDL model and that interact with tools during analysis, elaboration, or execution of the VHDL model

A *tool* is a program that maintains a representation of a VHDL model and provides the VHPI functions. A *VHPI program* is a program that calls the VHPI functions.

The information model is expressed in an object-oriented manner as a set of *classes* that bear *relationships* to one another. The classes are data types that have data *properties* and subprogram *operations*. A *subclass* may be derived from one or more *superclasses*, in which case it *inherits* the properties and operations of its superclasses. An *object* is an instance of a class and of any superclasses of that class. The *most-specialized* class of an object is the class of which the object is a member and that has no subclass of which the object is also a member. An *abstract* class cannot be the most-specialized class of any object; however, it may be a superclass of a non-abstract class that is the most-specialized class of an object.

Some objects are *static*; that is, once created, they remain in existence until termination of the tool. Other objects are *dynamic*; that is, once created, they may cease to exist at a later time during execution of the tool.

The properties of a class represent data that is characteristic of an object of the class. The VHPI provides functions that allow a VHPI program to access and modify the values of properties of a given object. By using such functions, a VHPI program can access and modify values of VHDL objects in a VHDL model.

In addition to the inheritance relationship, a class may bear an association relationship with one or more other classes. A *one-to-one* association means that an object of a class is associated with at most one object of the second class. A *one-to-many* association means that an object of the class is associated with possibly more than one object of the second class. The VHPI provides functions that allow a VHPI program to traverse associations; that is, to locate objects that are associated with a given object.

The information model contains two sub-models. The first, referred to as the *library information model*, represents the design units that comprise a VHDL model after analysis and prior to elaboration. The second, referred to as the *design hierarchy information model*, represents the elaborated VHDL model. It contains instances, created through elaboration, of objects from the library information model. The design hierarchy information model may be used by a tool that simulates the VHDL model to gain access to the state of the VHDL model during execution. The design hierarchy information model includes associations with objects in the library information model, allowing navigation between the information models. The information model also contains objects representing the tool and its environment.

A VHPI program can interact with a tool by providing *callback* functions. Such functions are identified to the tool using VHPI registration functions. The tool then calls the functions in response to events specified during registration. Such events include phases of tool execution, phases of model simulation, and changes of value of VHDL objects.

The VHPI also provides *utility* functions for such purposes as printing, error checking, and tool control.

In this standard, the VHPI information model is described using the UML notation. UML is described in OMG Unified Modeling Language Specification [B15]. The VHPI is ISO C-compliant. The VHPI functions are expressed as C functions, and the data and arguments used by the functions are expressed as C data types. ISO C is described in ISO/IEC 9899:1999 [B12], as corrected by ISO/IEC 9899:1999/Cor 1:2001 [B13] and ISO/IEC 9899:1999/Cor 2:2004 [B14].

15.1.1 VHPI naming conventions

Named items in the VHPI conform to the following conventions:

- The names of functions consist of the letters `vhpi` followed by an underline character and one or more words, each of which consists of lowercase letters, with a single underline character between words.
- The names of items other than functions consist of the letters `vhpi` followed by one or more words, each of which consists of an uppercase letter followed by zero or more lowercase letters, with no character between words.
- The names of types end in an uppercase letter T.
- The names of enumeration constants that correspond to classes end in an uppercase letter K.
- The names of enumeration constants that correspond to properties end in an uppercase letter P.
- The names of enumeration constants that correspond to one-to-many associations end in a lowercase letter s, indicating plurality.
- Some words are abbreviated, for example, `decl` for declaration, `stmt` for statement, `conc` for concurrent, `seq` for sequential, and `subp` for subprogram.

In this standard, C identifiers are formatted in a monospaced font to enhance readability of the text.

15.2 Capability sets

The VHPI is divided into a number of capability sets, each of which provides a subset of the VHPI operations, properties, and functions. Corresponding to each capability set, there is an enumeration constant of type `vhpiCapabilitiesT` defined in the VHPI header file (see Annex G).

The VHPI capability set names and corresponding enumeration constants are:

- Hierarchy set: `vhpiProvidesHierarchy`.
A tool that implements this hierarchy set shall provide access to objects in the design hierarchy information model that represent statically elaborated regions and declarations and shall provide access to the values of declared objects.
- Static access set: `vhpiProvidesStaticAccess`. This set requires the hierarchy set.
A tool that implements the static access set shall additionally provide access to objects in the design hierarchy information model that represent statically elaborated statements and the expressions within them.
- Connectivity set: `vhpiProvidesConnectivity`. This set requires the hierarchy set.

- A tool that implements the connectivity set shall additionally provide access to objects in the design hierarchy information model that represent drivers, contributors, loads, and port associations.
- Post-analysis set: `vhpiProvidesPostAnalysis`.
A tool that implements the post-analysis set shall provide access to objects in the library information model and shall provide access to the values of declared objects that are initialized with locally static expressions.
 - Basic foreign model set: `vhpiProvidesForeignModel`. This set requires the hierarchy set.
A tool that implements the basic foreign model set shall additionally support creation of foreign models and foreign model callbacks and shall provide access to objects in the design hierarchy information model that represent foreign models.
 - Advanced foreign model set: `vhpiProvidesAdvancedForeignModel`. This set requires the basic foreign model set.
A tool that implements the advanced foreign model set shall additionally support creation of foreign drivers and processes and scheduling of transactions on foreign drivers.
 - Save/restart set: `vhpiProvidesSaveRestart`.
A tool that implements the save/restart set shall support save and restart of foreign models, use of the `vhpi_put_data` and `vhpi_get_data` functions, save, and restart callbacks and shall provide access to the `Id` and `SaveRestartLocation` properties.
 - Reset set: `vhpiProvidesReset`.
A tool that implements the reset set shall support reset of foreign models and reset callbacks.
 - Basic debug and runtime simulation set: `vhpiProvidesDebugRuntime`. This set requires the static access set and the connectivity set.
A tool that implements the debug and runtime simulation set shall support use of the `vhpi_control`, `vhpi_get_time`, and `vhpi_get_next_time` functions; object value change callbacks for signals, ports, and drivers; time and action callbacks; and updating of signals, ports, and drivers.
 - Advanced debug and runtime simulation set: `vhpiProvidesAdvancedDebugRuntime`. This set requires the basic debug and runtime simulation set.
A tool that implements the advanced debug and runtime simulation set shall additionally support object value change callbacks for variables, updating of variables, and the `LineOffset` property.
 - Dynamically elaboration set: `vhpiProvidesDynamicElab`. This set requires the debug and runtime simulation set.
A tool that implements the dynamic elaboration set shall additionally provide access to objects that represent dynamically elaborated regions, declarations, and constructs.

If a tool specifies that it implements a given capability set, it shall provide all of the operations, properties, and functions specified for the capability set. If the capability set requires one or more other capability sets, the tool shall also implement the required capability sets. A tool shall provide a value for the `vhpiCapabilitiesP` property of the `tool` class that specifies the capability sets that the tool implements.

If a VHPI program calls an operation or function that is not provided in the capability sets provided by a tool, the function shall raise a VHPI error condition. Similarly, if a VHPI program accesses a property that is not provided in the capability sets provided by a tool, the access function shall raise a VHPI error condition. In both cases, the error message returned by a subsequent call to `vhpi_check_error` shall indicate that the operation is not implemented.

15.3 Handles

A *handle* is an opaque reference to an object in the VHPI information model. It is represented as a value of the data type `vhpiHandleT` (see Annex G); however, the interpretation of the representation is implementation defined. A handle allows a VHPI program to refer to an object without assuming details of the representation of the object. The VHPI provides functions that operate on objects referred to by handles. The particular operations that are legal for an object referred to by a handle depend on the class of the object. The class is identified by the `Kind` property of the object.

In this standard, if an object is described as being of a given class, the object may be of the given class, provided the class is not an abstract class or any non-abstract subclass of the given class.

NOTE—The `Kind` property of an object identifies the most specific class of the object, that is, the class for which no subclass is also a class of the object.

15.3.1 Handle creation

A handle is created by a tool as the result of one of the following functions called by a VHPI program:

- `vhpi_handle_by_name`, which returns a handle that refers to an object identified by a name
- `vhpi_handle_by_index`, which returns a handle that refers to an object in an ordered one-to-many association
- `vhpi_handle`, which returns a handle that refers to the object in a one-to-one association
- `vhpi_create`, which creates or modifies an object, such as a driver, a process statement, or a collection, and returns a handle that refers to the object
- `vhpi_register_cb`, which returns a handle that refers to the callback object
- `vhpi_register_foreignf`, which returns a handle that refers to the callback object
- `vhpi_iterator`, which returns a handle that refers to an iterator
- `vhpi_scan`, which returns a handle that refers to an object referenced by an iterator

A tool shall support multiple VHPI programs, each of which acquires handles. The way in which a tool implements handles shall allow a VHPI program to function correctly independently of other VHPI programs executing concurrently. A tool may share between VHPI programs resources associated with the implementation of handles and the objects to which they refer. However, the occurrence of such sharing shall not alter the effect of the VHPI programs.

If a tool creates two handles that refer to the same object, the tool may create two distinct handles or may provide the same handle in both cases. Two distinct handles that refer to the same object are equivalent.

NOTE—The number of handles that an implementation can create may be constrained by the capacity of the host system.

15.3.2 Handle release

The function `vhpi_release_handle` called by a VHPI program causes a tool to release a handle. If a tool shares resources associated with handles and one VHPI program releases a handle, other VHPI programs shall be able to continue to refer to objects using handles that they have not released.

The tool may reclaim resources associated with the representation of a released handle.

NOTES

- 1—It is recommended that a VHPI program release handles when they are no longer needed.

2—A tool may reclaim resources associated with a handle when the handle is released by a VHPI program, provided the requirements of 15.3 are met. As a consequence, resources may not be reclaimed immediately upon release of a handle by a VHPI program, as the resources may be associated with handles in use by other VHPI programs.

15.3.3 Handle comparison

The function `vhpi_compare_handles` compares handles. It returns the value `vhpiTrue` if the handles are equivalent (that is, they refer to the same object); otherwise it returns the value `vhpiFalse`.

15.3.4 Validity of handles

The *lifetime* of an object is the duration of existence of the object in the VHPI information model. A static object is created at some time during the execution of a tool and exists until termination of the tool. A dynamic object is created at some time during the execution of a tool and may cease to exist at a later time during the execution of the tool, either as a consequence of execution of the VHDL model or of removal by a VHPI program.

A tool can create a handle that refers to an object only during the lifetime of the object. A handle is said to be *valid* from the time of its creation until the time at which it is released, or until the object that it refers to ceases to exist, or until termination of the tool; at other times it is *invalid*. A VHPI program that attempts to refer to an object using an invalid handle is erroneous.

NOTE—A VHPI program that attempts to release an invalid handle is also erroneous.

16. VHPI access functions

This clause describes the VHPI functions that are used by VHPI programs to access the information model of a VHDL model.

16.1 Information access functions

The VHPI information access functions allow a VHPI program to navigate an association between objects.

The VHPI header file defines enumeration types that contain enumeration constants corresponding to association roles specified implicitly or explicitly in the information model. The name of each enumeration constant is the name of the corresponding role prefixed with the letters `vhpi`.

16.1.1 One-to-one association traversal

The VHPI header file defines the enumeration type `vhpiOneToOneT` that contains enumeration constants corresponding to one-to-one association roles.

If the information model includes a one-to-one association that is navigable from a reference class to a target class, the function `vhpi_handle` navigates from an object of the reference class to an object of the target class (see 21.19).

Examples:

Given the information model described by the UML class diagram shown in Figure 1

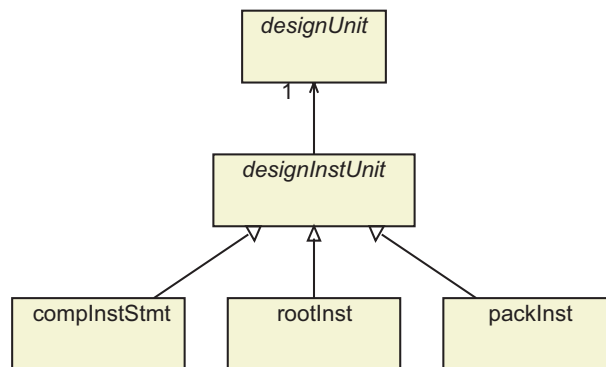


Figure 1—UML class diagram

the following VHPI program navigates from an object of the `compInstStmt` class to an object of the `designUnit` class using the enumeration constant `vhpiDesignUnit`.

```

void get_binding_info(vhpiHandleT instHdl) {
    char duName[MAXSTR];
    char libName[MAXSTR];
    vhpiHandleT duHdl;
    switch (vhpi_get(vhpiKindP, instHdl)) {
    case vhpiCompInstStmtK:
    case vhpiRootInstK:

```

```

case vhpiPackInstK:
    duHdl = vhpi_handle(vhpiDesignUnit, instHdl);
    sprintf (duName, "%s", vhpi_get_str(vhpiUnitNameP, duHdl));
    sprintf(libName, "%s", vhpi_get_str(vhpiLibLogicalNameP, duHdl));
    vhpi_printf("design unit name %s in library %s\n", duName, libName);
    break;
default:
    break;
}/* end switch */
}/* get_binding_info() */

```

Given the information model described by the UML class diagram shown in Figure 2

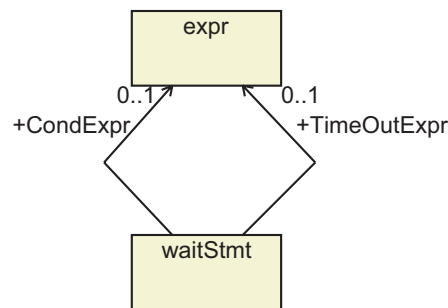


Figure 2—UML class diagram

the following VHPI program navigates from an object of the `waitStmt` class to one object of the `expr` class using the enumeration constant `vhpiCondExpr` and to a second object of the `expr` class using the enumeration constant `vhpiTimeOutExpr`.

```

vhpiHandleT stmtHdl, condHdl, timeHdl;
if (vhpi_get(vhpiKindP, stmtHdl) == vhpiWaitStmtK) {
    condHdl = vhpi_handle(vhpiCondExpr, stmtHdl);
    timeHdl = vhpi_handle(vhpiTimeOutExpr, stmtHdl);
}

```

16.1.2 One-to-many association traversal

The VHPI header file defines the enumeration type `vhpiOneToManyT` that contains enumeration constants corresponding to one-to-many association roles.

If the information model includes a one-to-many association that is navigable from a reference class to a target class, the function `vhpi_iterator` navigates from an object of the reference class to a set of objects of the target class (see 21.23).

If the information model includes an ordered one-to-many association that is navigable from a reference class to a target class, the function `vhpi_handle_by_index` navigates from an object of the reference class to an object of the target class (see 21.20).

NOTES

1—A VHPI program can use the `vhpi_scan` function to access the objects referred to by an iterator.