INTERNATIONAL STANDARD

IEC 61691-1-1

First edition
2004-10

IEEE 1076™

Behavioural languages –

Part 1-1:
VHDL language reference manual

Reference number
IEC 61691-1-1(E):2004
IEEE Std. 1076(E):2002

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

Publication numbering

As from 1 January 1997 all IEC publications are issued with a designation in the 60000 series. For example, IEC 34-1 is now referred to as IEC 60034-1.

Consolidated editions

The IEC is now publishing consolidated versions of its publications. For example, edition numbers 1.0, 1.1 and 1.2 refer, respectively, to the base publication, the base publication incorporating amendment 1 and the base publication incorporating amendments 1 and 2.

Further information on IEC publications

The technical content of IEC publications is kept under constant review by the IEC, thus ensuring that the content reflects current technology. Information relating to this publication, including its validity, is available in the IEC Catalogue of publications (see below) in addition to new editions, amendments and corrigenda. Information on the subjects under consideration and work in progress undertaken by the technical committee which has prepared this publication, as well as the list of publications issued, is also available from the following:

- **IEC Web Site (www.iec.ch)**

- **Catalogue of IEC publications**

  The on-line catalogue on the IEC web site (www.iec.ch/searchpub) enables you to search by a variety of criteria including text searches, technical committees and date of publication. On-line information is also available on recently issued publications, withdrawn and replaced publications, as well as corrigenda.

- **IEC Just Published**

  This summary of recently issued publications (www.iec.ch/online_news/ justpub) is also available by email. Please contact the Customer Service Centre (see below) for further information.

- **Customer Service Centre**

  If you have any questions regarding this publication or need further assistance, please contact the Customer Service Centre:

  Email: custserv@iec.ch
  Tel: +41 22 919 02 11
  Fax: +41 22 919 03 00

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

INTERNATIONAL STANDARD

IEC 61691-1-1

First edition
2004-10

IEEE 1076™

Behavioural languages –

Part 1-1:
VHDL language reference manual

Copyright © IEEE 2004 — All rights reserved

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

No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland
Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc, 3 Park Avenue, New York, NY 10016-5997, USA
Telephone: +1 732 562 3800 Telefax: +1 732 562 1571 E-mail: stds-info@ieee.org Web: www.standards.ieee.org

Commission Electrotechnique Internationale
International Electrotechnical Commission
Международная Электротехническая Комиссия

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

CONTENTS

FOREWORD 5

IEEE Introduction 9

0. Overview of this standard 10

0.1 Intent and scope of this standard 10
0.2 Structure and terminology of this standard 10

1. Design entities and configurations 13

1.1 Entity declarations 13
1.2 Architecture bodies 17
1.3 Configuration declarations 20

2. Subprograms and packages 26

2.1 Subprogram declarations 26
2.2 Subprogram bodies 29
2.3 Subprogram overloading 32
2.4 Resolution functions 34
2.5 Package declarations 35
2.6 Package bodies 36
2.7 Conformance rules 38

3. Types 39

3.1 Scalar types 40
3.2 Composite types 46
3.3 Access types 51
3.4 File types 54
3.5 Protected types 56

4. Declarations 60

4.1 Type declarations 60
4.2 Subtype declarations 61
4.3 Objects 62
4.4 Attribute declarations 76
4.5 Component declarations 77
4.6 Group template declarations 77
4.7 Group declarations 78

5. Specifications 80

5.1 Attribute specification 80
5.2 Configuration specification 82
5.3 Disconnection specification 90

6. Names 93

6.1 Names 93
6.2 Simple names 94
6.3 Selected names 95
6.4 Indexed names 97

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 61691-1-1-2004

6.5 Slice names ... 98
6.6 Attribute names ... 98

7. Expressions ... 100

7.1 Expressions ... 100
7.2 Operators ... 101
7.3 Operands ... 109
7.4 Static expressions ... 116
7.5 Universal expressions ... 118

8. Sequential statements ... 120

8.1 Wait statement ... 122
8.2 Assertion statement ... 119
8.3 Report statement ... 123
8.4 Signal assignment statement ... 123
8.5 Variable assignment statement ... 128
8.6 Procedure call statement ... 129
8.7 If statement ... 130
8.8 Case statement ... 130
8.9 Loop statement ... 131
8.10 Next statement ... 132
8.11 Exit statement ... 133
8.12 Return statement ... 133
8.13 Null statement ... 133

9. Concurrent statements ... 135

9.1 Block statement ... 135
9.2 Process statement ... 136
9.3 Concurrent procedure call statements ... 137
9.4 Concurrent assertion statements ... 138
9.5 Concurrent signal assignment statements ... 139
9.6 Component instantiation statements ... 144
9.7 Generate statements ... 150

10. Scope and visibility ... 151

10.1 Declarative region ... 151
10.2 Scope of declarations ... 152
10.3 Visibility ... 153
10.4 Use clauses ... 156
10.5 The context of overload resolution ... 157

11. Design units and their analysis ... 158

11.1 Design units ... 158
11.2 Design libraries ... 158
11.3 Context clauses ... 159
11.4 Order of analysis ... 160

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

12. Elaboration and execution ... 161

12.1 Elaboration of a design hierarchy ... 161
12.2 Elaboration of a block header ... 163
12.3 Elaboration of a declarative part ... 164
12.4 Elaboration of a statement part ... 168
12.5 Dynamic elaboration ... 171
12.6 Execution of a model ... 171

13. Lexical elements ... 178

13.1 Character set ... 178
13.2 Lexical elements, separators, and delimiters ... 181
13.3 Identifiers ... 182
13.4 Abstract literals ... 182
13.5 Character literals ... 184
13.6 String literals ... 184
13.7 Bit string literals ... 185
13.8 Comments ... 186
13.9 Reserved words ... 187
13.10 Allowable replacements of characters ... 190

14. Predefined language environment ... 189

14.1 Predefined attributes ... 189
14.2 Package STANDARD ... 203
14.3 Package TEXTIO ... 210

Annex A (informative) Syntax summary ... 215

Annex B (informative) Glossary ... 234

Annex C (informative) Potentially nonportable constructs ... 253

Annex D (informative) Changes from IEEE Std 1076, 2000 Edition ... 254

Annex E (informative) Features under consideration for removal ... 255

Annex F (informative) Bibliography ... 256

Annex G (informative) List of Participants ... 257

Index ... 258

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

INTERNATIONAL ELECTROTECHNICAL COMMISSION

BEHAVIOURAL LANGUAGES –

Part 1-1: VHDL language reference manual

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 61691-1-1 has been processed through IEC technical committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1076 (2002)	93/193/FDIS	93/198/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives.

The committee has decided that the contents of this publication will remain unchanged until 2007.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

IEC 61691 consists of the following parts, under the general title *Behavioural languages:*

IEC/IEEE 61691-1-1, Part 1-1: *VHDL language reference manual*

IEC 61691-2, Part 2: *VHDL multilogic system for model interoperability*

IEC 61691-3-1, Part 3-1: *Analog description in VHDL* (under consideration)

IEC 61691-3-2, Part 3-2: *Mathematical operation in VHDL*

IEC 61691-3-3, Part 3-3: *Synthesis in VHDL*

IEC 61691-3-4, Part 3-4: *Timing expressions in VHDL* (under consideration)

IEC 61691-3-5, Part 3-5: *Library utilities in VHDL* (under consideration)

IEC/IEEE 61691-4, Part 4: *Verilog® hardware description language*

IEC/IEEE 61691-5, Part 5: *VITAL ASIC (application specific integrated circuit) modeling specification*

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

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 IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim 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 IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim 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. IEC/IEEE Dual Logo International Standards documents are supplied “AS IS”.

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International 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 IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or 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.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or 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, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

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.

NOTE – 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 for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

IEEE Standard VHDL

Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 26 July 2002
American National Standards Institute

Approved 21 March 2002
IEEE-SA Standards Board

Abstract: VHSIC Hardware Description Language (VHDL) is defined. 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. Its primary audiences are the implementors of tools supporting the language and the advanced users of the language.
Keywords: computer languages, electronic systems, hardware, hardware design, VHDL

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

IEEE Introduction

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 specifies IEEE Std 1076-2002™, which is a revision of IEEE Std 1076, 2000 Edition™. This revision incorporates the addition of protected types and enhancements to the specification of shared variables which were completed in IEEE Std 1076, 2000 Edition™. As VHDL is now in wide use throughout the world, the 1076 Working Group endeavored to maintain a high level of stability with this revision. Although this revision does not provide significant changes to VHDL, it does enhance and clarify the language specification in several areas. Most notable is the improvement in the specification of default binding rules, buffer ports, scope and visibility, allowance of multi-byte characters in comments and other areas which will increase the portability of descriptions.

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 revision and encourages the participation of all interested parties in future language revisions. If interested in participating, please contact the VASG at stds-vasg@ieee.org or visit the following website: http://www.eda.org/pub/vasg.

IEEE-SA Trademark Usage/Compliance Statement

Proper usage of the trademark **IEEE Std 1076-2002™** is mandatory and is to be followed in all references of the Standard. The mark **IEEE®** is the registered trademark of the Institute of Electrical and Electronics Engineers, Inc., and must be used in bold type. It is to appear with the registered trademark symbol "®" the first time "**IEEE**" appears in the text. The use of "IEEE Std 1076-2002" should include the trademark symbol "TM" (e.g., **IEEE Std 1076-2002™**) at least the first time it is used in text, unless the number of the standard is also trademark registered (e.g., **802®**), then the symbol "®" must be used.

It is not permissible to use the standard number alone or with "**IEEE**" to indicate conformance or compliance with the associated standard. The user of the Standard should contact the Manager, Standards Licensing and Contracts for information concerning issues regarding indicating product compliance with an **IEEE** standard. To represent that a product has been designed to meet an **IEEE** standard, it is permissible to state that "the product has been engineered, designed or manufactured with the intent to meet the requirements of **IEEE Std 1076-2002™**." However, it is ***not*** permissible to state or refer to a product as "1076 compliant," "1076 certified," "**IEEE 1076** conformant," "**IEEE 1076** certified," or the like, unless the user has obtained a Certification License from the IEEE.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

BEHAVIOURAL LANGUAGES –

Part 1-1: VHDL language reference manual

0. Overview of this standard

This clause describes the purpose and organization of this standard, the IEEE Standard VHDL Language Reference Manual.

0.1 Intent and scope of this standard

The intent of this standard is to define VHSIC Hardware Description Language (VHDL) accurately. Its primary audiences are the implementor of tools supporting the language and the advanced user of the language. Other users are encouraged to use commercially available books, tutorials, and classes to learn the language in some detail prior to reading this standard. These resources generally focus on how to use the language, rather than how a VHDL-compliant tool is required to behave.

At the time of its publication, this document was the authoritative definition of VHDL. From time to time, it may become necessary to correct and/or clarify portions of this standard. Such corrections and clarifications may be published in separate documents. Such documents modify this standard at the time of their publication and remain in effect until superseded by subsequent documents or until the standard is officially revised.

0.2 Structure and terminology of this standard

This standard is organized into clauses, each of which focuses on some particular area of the language. Within each clause, individual constructs or concepts are discussed in each subclause.

Each subclause describing a specific construct begins with an introductory paragraph. Next, the syntax of the construct is described using one or more grammatical *productions*.

A set of paragraphs describing the meaning and restrictions of the construct in narrative form then follow. Unlike many other IEEE standards, which use the verb *shall* to indicate mandatory requirements of the standard and *may* to indicate optional features, the verb *is* is used uniformly throughout this document. In all cases, *is* is to be interpreted as having mandatory weight.

Additionally, the word *must* is used to indicate mandatory weight. This word is preferred over the more common *shall*, as *must* denotes a different meaning to different readers of this standard.

a) To the developer of tools that process VHDL, *must* denotes a requirement that the standard imposes. The resulting implementation is required to enforce the requirement and to issue an error if the requirement is not met by some VHDL source text.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

b) To the VHDL model developer, *must* denotes that the characteristics of VHDL are natural consequences of the language definition. The model developer is required to adhere to the constraint implied by the characteristic.

c) To the VHDL model user, *must* denotes that the characteristics of the models are natural consequences of the language definition. The model user can depend on the characteristics of the model implied by its VHDL source text.

Finally, each clause may end with examples, notes, and references to other pertinent clauses.

0.2.1 Syntactic description

The form of a VHDL description is described by means of context-free syntax using a simple variant of the backus naur form; in particular:

a) Lowercase words in roman font, some containing embedded underlines, are used to denote syntactic categories, for example:

formal_port_list

Whenever the name of a syntactic category is used, apart from the syntax rules themselves, spaces take the place of underlines (thus, “formal port list” would appear in the narrative description when referring to the above syntactic category).

b) Boldface words are used to denote reserved words, for example:

array

Reserved words must be used only in those places indicated by the syntax.

c) A *production* consists of a *left-hand side*, the symbol “::=” (which is read as “can be replaced by”), and a *right-hand side*. The left-hand side of a production is always a syntactic category; the right-hand side is a replacement rule. The meaning of a production is a textual-replacement rule: any occurrence of the left-hand side may be replaced by an instance of the right-hand side.

d) A vertical bar (|) separates alternative items on the right-hand side of a production unless it occurs immediately after an opening brace, in which case it stands for itself, as follows:

letter_or_digit ::= letter | digit
choices ::= choice { | choice }

In the first instance, an occurrence of “letter_or_digit” can be replaced by either “letter” or “digit.” In the second case, “choices” can be replaced by a list of “choice,” separated by vertical bars [see item f) for the meaning of braces].

e) Square brackets [ ] enclose optional items on the right-hand side of a production; thus, the following two productions are equivalent:

return_statement ::= **return** [expression] ;
return_statement ::= **return** ; | **return** expression ;

Note, however, that the initial and terminal square brackets in the right-hand side of the production for signatures (see 2.3.2) are part of the syntax of signatures and do not indicate that the entire right-hand side is optional.

f) Braces { } enclose a repeated item or items on the right-hand side of a production. The items may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two productions are equivalent:

term ::= factor { multiplying_operator factor }
term ::= factor | term multiplying_operator factor

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

g) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *type*_name and *subtype*_name are both syntactically equivalent to name alone.

h) The term simple_name is used for any occurrence of an identifier that already denotes some declared entity.

0.2.2 Semantic description

The meaning and restrictions of a particular construct are described with a set of narrative rules immediately following the syntactic productions. In these rules, an italicized term indicates the definition of that term and identifiers appearing entirely in uppercase letters refer to definitions in package STANDARD (see 14.2).

The following terms are used in these semantic descriptions with the following meanings:

erroneous: The condition described represents an ill-formed description; however, implementations are not required to detect and report this condition. Conditions are deemed erroneous only when it is impossible in general to detect the condition during the processing of the language.

error: The condition described represents an ill-formed description; implementations are required to detect the condition and report an error to the user of the tool.

illegal: A synonym for “error.”

legal: The condition described represents a well-formed description.

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

Prior to this subclause are several pieces of introductory material; following Clause 14 are some annexes and an index. The front matter, annexes, and index serve to orient and otherwise aid the user of this standard, but are not part of the definition of VHDL.

Some clauses of this standard contain examples, notes, and cross-references to other clauses of the standard; these parts always appear at the end of a clause. Examples are meant to illustrate the possible forms of the construct described. Illegal examples are italicized. Notes are meant to emphasize consequences of the rules described in the clause or elsewhere. In order to distinguish notes from the other narrative portions of this standard, notes are set as enumerated paragraphs in a font smaller than the rest of the text. Cross-references are meant to guide the user to other relevant clauses of the standard. Examples, notes, and cross-references are not part of the definition of the language.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

1. Design entities and configurations

The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a subsystem, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A *configuration* can be used to describe how design entities are put together to form a complete design.

A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design. The top-level block in such a hierarchy is the design entity itself; such a block is an *external* block that resides in a library and may be used as a component of other designs. Nested blocks in the hierarchy are *internal* blocks, defined by block statements (see 9.1).

A design entity may also be described in terms of interconnected components. Each component of a design entity may be bound to a lower-level design entity in order to define the structure or behavior of that component. Successive decomposition of a design entity into components, and binding those components to other design entities that may be decomposed in like manner, results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy.* The bindings necessary to identify a design hierarchy can be specified in a configuration of the top-level entity in the hierarchy.

This clause describes the way in which design entities and configurations are defined. A design entity is defined by an *entity declaration* together with a corresponding *architecture body.* A configuration is defined by a *configuration declaration.*

1.1 Entity declarations

An entity declaration defines the interface between a given design entity and the environment in which it is used. It may also specify declarations and statements that are part of the design entity. A given entity declaration may be shared by many design entities, each of which has a different architecture. Thus, an entity declaration can potentially represent a class of design entities, each with the same interface.

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;
```

The entity header and entity declarative part consist of declarative items that pertain to each design entity whose interface is defined by the entity declaration. The entity statement part, if present, consists of concurrent statements that are present in each such design entity.

If a simple name appears at the end of an entity declaration, it must repeat the identifier of the entity declaration.

1.1.1 Entity header

The entity header declares objects used for communication between a design entity and its environment.

```
entity_header ::=
    [ formal_generic_clause ]
    [ formal_port_clause ]
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

generic_clause ::=
generic (generic_list) ;

port_clause ::=
port (port_list) ;

The generic list in the formal generic clause defines generic constants whose values may be determined by the environment. The port list in the formal port clause defines the input and output ports of the design entity.

In certain circumstances, the names of generic constants and ports declared in the entity header become visible outside of the design entity (see 10.2 and 10.3).

Examples:

— An entity declaration with port declarations only:

```
entity Full_Adder is
    port (X, Y, Cin: in Bit; Cout, Sum: out Bit) ;
end Full_Adder ;
```

— An entity declaration with generic declarations also:

```
entity AndGate is
    generic
        (N: Natural := 2);
    port
        (Inputs:   in Bit_Vector (1 to N);
         Result:   out Bit) ;
end entity AndGate ;
```

— An entity declaration with neither:

```
entity TestBench is
end TestBench ;
```

1.1.1.1 Generics

Generics provide a channel for static information to be communicated to a block from its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements.

generic_list ::= *generic*_interface_list

The generics of a block are defined by a generic interface list; interface lists are described in 4.3.2.1. Each interface element in such a generic interface list declares a formal generic.

The value of a generic constant may be specified by the corresponding actual in a generic association list. If no such actual is specified for a given formal generic (either because the formal generic is unassociated or because the actual is **open**), and if a default expression is specified for that generic, the value of this expression is the value of the generic. It is an error if no actual is specified for a given formal generic and no default expression is present in the corresponding interface element. It is an error if some of the subelements of a composite formal generic are connected and others are either unconnected or unassociated.

NOTE—Generics may be used to control structural, dataflow, or behavioral characteristics of a block, or may simply be used as documentation. In particular, generics may be used to specify the size of ports; the number of subcomponents within a block; the timing characteristics of a block; or even the physical characteristics of a design such as temperature, capacitance, or location.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

1.1.1.2 Ports

Ports provide channels for dynamic communication between a block and its environment. The following applies to both external blocks defined by design entities and to internal blocks defined by block statements, including those equivalent to component instantiation statements and generate statements (see 9.7).

port_list ::= *port*_interface_list

The ports of a block are defined by a port interface list; interface lists are described in 4.3.2.1. Each interface element in the port interface list declares a formal port.

To communicate with other blocks, the ports of a block can be associated with signals in the environment in which the block is used. Moreover, the ports of a block may be associated with an expression in order to provide these ports with constant driving values; such ports must be of mode **in**. A port is itself a signal (see 4.3.1.2); thus, a formal port of a block may be associated as an actual with a formal port of an inner block. The port, signal, or expression associated with a given formal port is called the actual corresponding to the formal port (see 4.3.2.2). The actual, if a port or signal, must be denoted by a static name (see 6.1). The actual, if an expression, must be a globally static expression (see 7.4).

After a given description is completely elaborated (see Clause 12), if a formal port is associated with an actual that is itself a port, then the following restrictions apply depending upon the mode (see 4.3.2) of the formal port:

a) For a formal port of mode **in**, the associated actual must be a port of mode **in**, **inout**, or **buffer**.
b) For a formal port of mode **out**, the associated actual must be a port of mode **out**, **inout**, or **buffer**.
c) For a formal port of mode **inout**, the associated actual must be a port of mode **inout**, or **buffer**.
d) For a formal port of mode **buffer**, the associated actual must be a port of mode **out**, **inout**, or **buffer**.
e) For a formal port of mode **linkage**, the associated actual may be a port of any mode.

If a formal port is associated with an actual port, signal, or expression, then the formal port is said to be *connected*. If a formal port is instead associated with the reserved word **open**, then the formal is said to be *unconnected*. It is an error if a port of mode **in** is unconnected or unassociated (see 4.3.2.2) unless its declaration includes a default expression (see 4.3.2). It is an error if a port of any mode other than **in** is unconnected or unassociated and its type is an unconstrained array type. It is an error if some of the subelements of a composite formal port are connected and others are either unconnected or unassociated.

NOTE—Ports of mode linkage may be removed from a future version of the language (see Annex F).

1.1.2 Entity declarative part

The entity declarative part of a given entity declaration declares items that are common to all design entities whose interfaces are defined by the given entity declaration.

entity_declarative_part ::=
    { entity_declarative_item }

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

```
entity_declarative_item ::=
        subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | signal_declaration
      | shared_variable_declaration
      | file_declaration
      | alias_declaration
      | attribute_declaration
      | attribute_specification
      | disconnection_specification
      | use_clause
      | group_template_declaration
      | group_declaration
```

Names declared by declarative items in the entity declarative part of a given entity declaration are visible within the bodies of corresponding design entities, as well as within certain portions of a corresponding configuration declaration.

The various kinds of declaration are described in Clause 4, and the various kinds of specification are described in Clause 5. The use clause, which makes externally defined names visible within the block, is described in Clause 10.

Example:

— An entity declaration with entity declarative items:

```
entity ROM is
    port (      Addr:   in      Word;
                Data:   out     Word;
                Sel:    in      Bit);
    type        Instruction is array (1 to 5) of Natural;
    type        Program is array (Natural range <>) of Instruction;
    use         Work.OpCodes.all, Work.RegisterNames.all;
    constant ROM_Code: Program :=
        (
                (STM,   R14,    R12,    12,     R13) ,
                (LD,    R7,     32,     0,      R1  ) ,
                (BAL,   R14,    0,      0,      R7  ) ,
                        •
                        •       -- etc.
                        •
        ) ;
end ROM;
```

NOTE—The entity declarative part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.3.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

1.1.3 Entity statement part

The entity statement part contains concurrent statements that are common to each design entity with this interface.

```
entity_statement_part ::=
        { entity_statement }

entity_statement ::=
         concurrent_assertion_statement
        | passive_concurrent_procedure_call
        | passive_process_statement
```

It is an error if any statements other than concurrent assertion statements, concurrent procedure call statements, or process statements appear in the entity statement part. All entity statements must be passive (see 9.2). Such statements may be used to monitor the operating conditions or characteristics of a design entity.

Example:

— An entity declaration with statements:

```
entity Latch is
    port (  Din:    in    Word;
            Dout:   out   Word;
            Load:   in    Bit;
            Clk:    in    Bit );
    constant Setup: Time := 12 ns;
    constant PulseWidth: Time := 50 ns;
    use Work.TimingMonitors.all;
begin
    assert Clk='1' or Clk'Delayed'Stable (PulseWidth);
    CheckTiming (Setup, Din, Load, Clk);
end ;
```

NOTE—The entity statement part of a design entity whose corresponding architecture is decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.4.

1.2 Architecture bodies

An architecture body defines the body of a design entity. It specifies the relationships between the inputs and outputs of a design entity and may be expressed in terms of structure, dataflow, or behavior. Such specifications may be partial or complete.

```
architecture_body ::=
        architecture identifier of entity_name is
            architecture_declarative_part
        begin
            architecture_statement_part
        end [ architecture ] [ architecture_simple_name ] ;
```

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.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

The entity name identifies the name of the entity declaration that defines the interface of this design entity. For a given design entity, both the entity declaration and the associated architecture body must reside in the same library.

If a simple name appears at the end of an architecture body, it must repeat the identifier of the architecture body.

More than one architecture body may exist corresponding to a given entity declaration. Each declares a different body with the same interface; thus, each together with the entity declaration represents a different design entity with the same interface.

NOTE—Two architecture bodies that are associated with different entity declarations may have the same simple name, even if both architecture bodies (and the corresponding entity declarations) reside in the same library.

1.2.1 Architecture declarative part

The architecture declarative part contains declarations of items that are available for use within the block defined by the design entity.

```
architecture_declarative_part ::=
        { block_declarative_item }

block_declarative_item ::=
          subprogram_declaration
        | subprogram_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | signal_declaration
        | shared_variable_declaration
        | file_declaration
        | alias_declaration
        | component_declaration
        | attribute_declaration
        | attribute_specification
        | configuration_specification
        | disconnection_specification
        | use_clause
        | group_template_declaration
        | group_declaration
```

The various kinds of declaration are described in Clause 4, and the various kinds of specification are described in Clause 5. The use clause, which makes externally defined names visible within the block, is described in Clause 10.

NOTE—The declarative part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. (See 12.3).

1.2.2 Architecture statement part

The architecture statement part contains statements that describe the internal organization and/or operation of the block defined by the design entity.

```
architecture_statement_part ::=
        { concurrent_statement }
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

All of the statements in the architecture statement part are concurrent statements, which execute asynchronously with respect to one another. The various kinds of concurrent statements are described in Clause 9.

Examples:

— A body of entity Full_Adder:

```
architecture DataFlow of Full_Adder is
    signal A,B: Bit;
begin
    A <= X xor Y;
    B <= A and Cin;
    Sum <= A xor Cin;
    Cout <= B or (X and Y);
end architecture DataFlow ;
```

— A body of entity TestBench:

```
library Test;
use Test.Components.all;
architecture Structure of TestBench is
    component Full_Adder port (X, Y, Cin: Bit; Cout, Sum: out Bit);
    end component;
    signal A,B,C,D,E,F,G: Bit;
    signal OK: Boolean;
begin
    UUT:        Full_Adder    port map (A,B,C,D,E);
    Generator:  AdderTest     port map (A,B,C,F,G);
    Comparator: AdderCheck    port map (D,E,F,G,OK);
end Structure;
```

— A body of entity AndGate:

```
architecture Behavior of AndGate is
begin
    process (Inputs)
        variable Temp: Bit;
    begin
        Temp := '1';
        for i in Inputs'Range loop
            if Inputs(i) = '0' then
                Temp := '0';
                exit;
            end if;
        end loop;
        Result <= Temp after 10 ns;
    end process;
end Behavior;
```

NOTE—The statement part of an architecture decorated with the 'FOREIGN attribute is subject to special elaboration rules. See 12.4.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

1.3 Configuration declarations

The binding of component instances to design entities is performed by configuration specifications (see 5.2); such specifications appear in the declarative part of the block in which the corresponding component instances are created. In certain cases, however, it may be appropriate to leave unspecified the binding of component instances in a given block and to defer such specification until later. A configuration declaration provides the mechanism for specifying such deferred bindings.

configuration_declaration ::=
    configuration identifier **of** *entity*_name **is**
        configuration_declarative_part
        block_configuration
    end [**configuration**] [*configuration*_simple_name] ;

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_declarative_item ::=
    use_clause
    | attribute_specification
    | group_declaration

The entity name identifies the name of the entity declaration that defines the design entity at the apex of the design hierarchy. For a configuration of a given design entity, both the configuration declaration and the corresponding entity declaration must reside in the same library.

If a simple name appears at the end of a configuration declaration, it must repeat the identifier of the configuration declaration.

NOTES

1—A configuration declaration achieves its effect entirely through elaboration (see Clause 12). There are no behavioral semantics associated with a configuration declaration.

2—A given configuration may be used in the definition of another, more complex configuration.

Examples:

— An architecture of a microprocessor:

```
architecture Structure_View of Processor is
    component ALU port ( ••• );  end component;
    component MUX port ( ••• );  end component;
    component Latch port ( ••• );  end component;
begin
    A1: ALU port map ( ••• ) ;
    M1: MUX port map ( ••• ) ;
    M2: MUX port map ( ••• ) ;
    M3: MUX port map ( ••• ) ;
    L1: Latch port map ( ••• ) ;
    L2: Latch port map ( ••• ) ;
end Structure_View ;
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

— A configuration of the microprocessor:

```
library TTL, Work ;
configuration V4_27_87 of Processor is
    use Work.all ;
    for Structure_View
        for A1: ALU
                use configuration TTL.SN74LS181 ;
        end for ;
        for M1,M2,M3: MUX
                use entity Multiplex4 (Behavior) ;
        end for ;
        for all: Latch
                — use defaults
        end for ;
    end for ;
end configuration V4_27_87 ;
```

1.3.1 Block configuration

A block configuration defines the configuration of a block. Such a block is either an internal block defined by a block statement or an external block defined by a design entity. If the block is an internal block, the defining block statement is either an explicit block statement or an implicit block statement that is itself defined by a generate statement.

```
block_configuration ::=
        for block_specification
            { use_clause }
            { configuration_item }
        end for ;

block_specification ::=
          architecture_name
        | block_statement_label
        | generate_statement_label [ ( index_specification ) ]

index_specification ::=
          discrete_range
        | static_expression

configuration_item ::=
          block_configuration
        | component_configuration
```

The block specification identifies the internal or external block to which this block configuration applies.

If a block configuration appears immediately within a configuration declaration, then the block specification of that block configuration must be an architecture name, and that architecture name must denote a design entity body whose interface is defined by the entity declaration denoted by the entity name of the enclosing configuration declaration.

If a block configuration appears immediately within a component configuration, then the corresponding components must be fully bound (see 5.2.1.1), the block specification of that block configuration must be an architecture name, and that architecture name must denote the same architecture body as that to which the corresponding components are bound.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

If a block configuration appears immediately within another block configuration, then the block specification of the contained block configuration must be a block statement or generate statement label, and the label must denote a block statement or generate statement that is contained immediately within the block denoted by the block specification of the containing block configuration.

If the scope of a declaration (see 10.2) includes the end of the declarative part of a block corresponding to a given block configuration, then the scope of that declaration extends to each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. Similarly, if a declaration is visible (either directly or by selection) at the end of the declarative part of a block corresponding to a given block configuration, then the declaration is visible in each configuration item contained in that block configuration, with the exception of block configurations that configure external blocks. Additionally, if a given declaration is a homograph of a declaration that a use clause in the block configuration makes potentially directly visible, then the given declaration is not directly visible in the block configuration or any of its configuration items. See 10.3.

For any name that is the label of a block statement appearing immediately within a given block, a corresponding block configuration may appear as a configuration item immediately within a block configuration corresponding to the given block. For any collection of names that are labels of instances of the same component appearing immediately within a given block, a corresponding component configuration may appear as a configuration item immediately within a block configuration corresponding to the given block.

For any name that is the label of a generate statement immediately within a given block, one or more corresponding block configurations may appear as configuration items immediately within a block configuration corresponding to the given block. Such block configurations apply to implicit blocks generated by that generate statement. If such a block configuration contains an index specification that is a discrete range, then the block configuration applies to those implicit block statements that are generated for the specified range of values of the corresponding generate parameter; the discrete range has no significance other than to define the set of generate statement parameter values implied by the discrete range. If such a block configuration contains an index specification that is a static expression, then the block configuration applies only to the implicit block statement generated for the specified value of the corresponding generate parameter. If no index specification appears in such a block configuration, then it applies to exactly one of the following sets of blocks:

— All implicit blocks (if any) generated by the corresponding generate statement, if and only if the corresponding generate statement has a generation scheme including the reserved word **for**

— The implicit block generated by the corresponding generate statement, if and only if the corresponding generate statement has a generation scheme including the reserved word **if** and if the condition in the generate scheme evaluates to TRUE

— No implicit or explicit blocks, if and only if the corresponding generate statement has a generation scheme including the reserved word **if** and the condition in the generate scheme evaluates to FALSE.

If the block specification of a block configuration contains a generate statement label, and if this label contains an index specification, then it is an error if the generate statement denoted by the label does not have a generation scheme including the reserved word **for**.

Within a given block configuration, whether implicit or explicit, an implicit block configuration is assumed to appear for any block statement that appears within the block corresponding to the given block configuration, if no explicit block configuration appears for that block statement. Similarly, an implicit component configuration is assumed to appear for each component instance that appears within the block corresponding to the given block configuration, if no explicit component configuration appears for that instance. Such implicit configuration items are assumed to appear following all explicit configuration items in the block configuration.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

It is an error if, in a given block configuration, more than one configuration item is defined for the same block or component instance.

NOTES

1—As a result of the rules described in the preceding paragraphs and in Clause 10, a simple name that is visible by selection at the end of the declarative part of a given block is also visible by selection within any configuration item contained in a corresponding block configuration. If such a name is directly visible at the end of the given block declarative part, it will likewise be directly visible in the corresponding configuration items, unless a use clause for a different declaration with the same simple name appears in the corresponding configuration declaration, and the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the name will be directly visible within the corresponding configuration items except at those places that fall within the scope of the additional use clause (at which places neither name will be directly visible).

2—If an implicit configuration item is assumed to appear within a block configuration, that implicit configuration item will never contain explicit configuration items.

3—If the block specification in a block configuration specifies a generate statement label, and if this label contains an index specification that is a discrete range, then the direction specified or implied by the discrete range has no significance other than to define, together with the bounds of the range, the set of generate statement parameter values denoted by the range. Thus, the following two block configurations are equivalent:

```
for Adders(31 downto 0) ••• end for;
for Adders(0 to 31) ••• end for;
```

4—A block configuration is allowed to appear immediately within a configuration declaration only if the entity declaration denoted by the entity name of the enclosing configuration declaration has associated architectures. Furthermore, the block specification of the block configuration must denote one of these architectures.

Examples:

— A block configuration for a design entity:

```
for ShiftRegStruct                    -- An architecture name.
    -- Configuration items
    -- for blocks and components
    -- within ShiftRegStruct.
end for ;
```

— A block configuration for a block statement:

```
for B1                                -- A block label.
    -- Configuration items
    -- for blocks and components
    -- within block B1.
end for ;
```

1.3.2 Component configuration

A component configuration defines the configuration of one or more component instances in a corresponding block.

```
component_configuration ::=
    for component_specification
        [ binding_indication ; ]
        [ block_configuration ]
    end for ;
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

The component specification (see 5.2) identifies the component instances to which this component configuration applies. A component configuration that appears immediately within a given block configuration applies to component instances that appear immediately within the corresponding block.

It is an error if two component configurations apply to the same component instance.

If the component configuration contains a binding indication (see 5.2.1), then the component configuration implies a configuration specification for the component instances to which it applies. This implicit configuration specification has the same component specification and binding indication as that of the component configuration.

If a given component instance is unbound in the corresponding block, then any explicit component configuration for that instance that does not contain an explicit binding indication will contain an implicit, default binding indication (see 5.2.2). Similarly, if a given component instance is unbound in the corresponding block, then any implicit component configuration for that instance will contain an implicit, default binding indication.

It is an error if a component configuration contains an explicit block configuration and the component configuration does not bind all identified component instances to the same design entity.

Within a given component configuration, whether implicit or explicit, an implicit block configuration is assumed for the design entity to which the corresponding component instance is bound, if no explicit block configuration appears and if the corresponding component instance is fully bound.

Examples:

— A component configuration with binding indication:

```
for all: IOPort
    use entity StdCells.PadTriState4 (DataFlow)
        port map (Pout=>A, Pin=>B, IO=>Dir, Vdd=>Pwr, Gnd=>Gnd) ;
end for ;
```

— A component configuration containing block configurations:

```
for D1: DSP
    for DSP_STRUCTURE
        -- Binding specified in design entity or else defaults.
        for Filterer
                -- Configuration items for filtering components.
        end for ;
        for Processor
                -- Configuration items for processing components.
        end for ;
end for ;
end for ;
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

NOTE—The requirement that all component instances corresponding to a block configuration be bound to the same design entity makes the following configuration illegal:

```
architecture A of E is
    component C is end component C;
    for L1: C use entity E1(X);
    for L2: C use entity E2(X);
begin
    L1: C;
    L2: C;
end architecture A;

configuration Illegal of Work.E is
  for A
    for all: C
                for X      -- Does not apply to the same design entity in all instances of C.
                  •••
                end for;   -- X
    end for; -- C
  end for; -- A
end configuration Illegal ;
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

2. Subprograms and packages

Subprograms define algorithms for computing values or exhibiting behavior. They may be used as computational resources to convert between values of different types, to define the resolution of output values driving a common signal, or to define portions of a process. Packages provide a means of defining these and other resources in a way that allows different design units to share the same declarations.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. Certain functions, designated *pure* functions, return the same value each time they are called with the same values as actual parameters; the remainder, *impure* functions, may return a different value each time they are called, even when multiple calls have the same actual parameter values. In addition, impure functions can update objects outside of their scope and can access a broader class of values than can pure functions. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.

Packages may also be defined in two parts. A package declaration defines the visible contents of a package; a package body provides hidden details. In particular, a package body contains the bodies of any subprograms declared in the package declaration.

2.1 Subprogram declarations

A subprogram declaration declares a procedure or a function, as indicated by the appropriate reserved word.

```
subprogram_declaration ::=
      subprogram_specification ;

subprogram_specification ::=
         procedure designator [ ( formal_parameter_list ) ]
        | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
                              return type_mark

designator ::=  identifier  |  operator_symbol

operator_symbol ::=  string_literal
```

The specification of a procedure specifies its designator and its formal parameters (if any). The specification of a function specifies its designator, its formal parameters (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 2.3.1). The sequence of characters represented by an operator symbol must be an operator belonging to one of the classes of operators defined in 7.2. Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

NOTES

1—All subprograms can be called recursively.

2—The restrictions on pure functions are enforced even when the function appears within a protected type. That is, pure functions whose body appears in the protected type body must not directly reference variables declared immediately within the declarative region associated with the protected type. However, impure functions and procedures whose bodies appear in the protected type body may make such references. Such references are made only when the referencing subprogram has exclusive access to the declarative region associated with the protected type.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

2.1.1 Formal parameters

The formal parameter list in a subprogram specification defines the formal parameters of the subprogram.

formal_parameter_list ::= *parameter*_interface_list

Formal parameters of subprograms may be constants, variables, signals, or files. In the first three cases, the mode of a parameter determines how a given formal parameter is accessed within the subprogram. The mode of a formal parameter, together with its class, also determines how such access is implemented. In the fourth case, that of files, the parameters have no mode.

For those parameters with modes, the only modes that are allowed for formal parameters of a procedure are **in**, **inout**, and **out**. If the mode is **in** and no object class is explicitly specified, **constant** is assumed. If the mode is **inout** or **out**, and no object class is explicitly specified, **variable** is assumed.

For those parameters with modes, the only mode that is allowed for formal parameters of a function is the mode **in** (whether this mode is specified explicitly or implicitly). The object class must be **constant**, **signal**, or **file**. If no object class is explicitly given, **constant** is assumed.

In a subprogram call, the actual designator (see 4.3.2.2) associated with a formal parameter of class **signal** must be a name denoting a signal. The actual designator associated with a formal of class **variable** must be a name denoting a variable. The actual designator associated with a formal of class **constant** must be an expression. The actual designator associated with a formal of class **file** must be a name denoting a file.

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 formal.

2.1.1.1 Constant and variable parameters

For parameters of class **constant** or **variable**, only the values 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 a scalar type or an access type, 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**, 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. 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.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

For a subprogram having a parameter whose type is a protected type, the parameter is passed by reference. It is an error if the mode of the parameter is other than **inout**.

For a formal parameter of a constrained array subtype of mode **in** or **inout**, it is an error if the value of the associated actual parameter (after application of any conversion function or type conversion present in the actual part) does not contain a matching element for each element of the formal. For a formal parameter whose declaration contains a subtype indication denoting an unconstrained array type, the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram. It is also an error if, in either case, the value of each element of the actual array (after applying any conversion function or type conversion present in the actual part) does not belong to the element subtype of the formal. If the formal parameter is of mode **out** or **inout**, it is also an error if, at the end of the subprogram call, the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the element subtype of the actual.

NOTE—For parameters of array and record types, the parameter passing rules imply that if no actual parameter of such a type is accessible by more than one path, then the effect of a subprogram call is the same whether or not the implementation uses copying for parameter passing. If, however, there are multiple access paths to such a parameter (for example, if another formal parameter is associated with the same actual parameter), then the value of the formal is undefined after updating the actual other than by updating the formal. A description using such an undefined value is erroneous.

2.1.1.2 Signal parameter

For a formal parameter of class **signal**, references to the signal, the driver of the signal, or both, are passed into the subprogram call.

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.

It is an error if signal-valued attributes 'STABLE, 'QUIET, 'TRANSACTION, and 'DELAYED of formal signal parameters of any mode are read within a subprogram.

A process statement contains a driver for each actual signal associated with a formal signal parameter of mode **out** or **inout** in a subprogram call. Similarly, a subprogram contains a driver for each formal signal parameter of mode **out** or **inout** declared in its subprogram specification.

For a signal parameter of mode **inout** or **out**, the driver of an actual signal is associated with the corresponding driver of the formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, an assignment to the driver of a formal signal parameter is equivalent to an assignment to the driver of the actual signal.

If an actual signal is associated with a signal parameter of any mode, the actual must be denoted by a static signal name. It is an error if a conversion function or type conversion appears in either the formal part or the actual part of an association element that associates an actual signal with a formal signal parameter.

If an actual signal is associated with a signal parameter of any mode, and if the type of the formal is a scalar type, then it is an error if the bounds and direction of the subtype denoted by the subtype indication of the formal are not identical to the bounds and direction of the subtype denoted by the subtype indication of the actual.

If an actual signal is associated with a formal signal parameter, and if the formal is of a constrained array subtype, then it is an error if the actual does not contain a matching element for each element of the formal. If an actual signal is associated with a formal signal parameter, and if the subtype denoted by the subtype indication of the declaration of the formal is an unconstrained array type, then the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

It is also an error if the mode of the formal is **in** or **inout** and if the value of each element of the actual array does not belong to the element subtype of the formal.

A formal signal parameter is a guarded signal if and only if it is associated with an actual signal that is a guarded signal. It is an error if the declaration of a formal signal parameter includes the reserved word **bus** (see 4.3.2).

NOTE—It is a consequence of the preceding rules that a procedure with an **out** or **inout** signal parameter called by a process does not have to complete in order for any assignments to that signal parameter within the procedure to take effect. Assignments to the driver of a formal signal parameter are equivalent to assignments directly to the actual driver contained in the process calling the procedure.

2.1.1.3 File parameters

For parameters of class **file**, references to the actual file are passed into the subprogram. No particular parameter-passing mechanism is defined by the language, but a reference to the formal parameter must be equivalent to a reference to the actual parameter. It is an error if an association element associates an actual with a formal parameter of a file type and that association element contains a conversion function or type conversion. It is also an error if a formal of a file type is associated with an actual that is not of a file type.

At the beginning of a given subprogram call, a file parameter is open (see 3.4.1) if and only if the actual file object associated with the given parameter in a given subprogram call is also open. Similarly, at the beginning of a given subprogram call, both the access mode of and external file associated with (see 3.4.1) an open file parameter are the same as, respectively, the access mode of and the external file associated with the actual file object associated with the given parameter in the subprogram call.

At the completion of the execution of a given subprogram call, the actual file object associated with a given file parameter is open if and only if the formal parameter is also open. Similarly, at the completion of the execution of a given subprogram call, the access mode of and the external file associated with an open actual file object associated with a given file parameter are the same as, respectively, the access mode of and the external file associated with the associated formal parameter.

2.2 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
        subprogram_specification is
            subprogram_declarative_part
        begin
            subprogram_statement_part
        end [ subprogram_kind ] [ designator ] ;

subprogram_declarative_part ::=
        { subprogram_declarative_item }
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

subprogram_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration

subprogram_statement_part ::=
{ sequential_statement }

subprogram_kind ::= **procedure** | **function**

The declaration of a subprogram is optional. In the absence of such a declaration, the subprogram specification of the subprogram body acts as the declaration. For each subprogram declaration, there shall be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body shall conform (see 2.7) to the subprogram specification of the declaration. Furthermore, both the declaration and the body must occur immediately within the same declarative region (see 10.1).

If a subprogram kind appears at the end of a subprogram body, it must repeat the reserved word given in the subprogram specification. If a designator appears at the end of a subprogram body, it must repeat the designator of the subprogram.

It is an error if a variable declaration in a subprogram declarative part declares a shared variable. (See 4.3.1.3 and 8.1.4.)

A *foreign subprogram* is one that is decorated with the attribute 'FOREIGN, defined in package STANDARD (see 14.2). The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram; such restrictions may include restrictions on the number and allowable order of the parameters.

Excepting foreign subprograms, the algorithm performed by a subprogram is defined by the sequence of statements that appears in the subprogram statement part. For a foreign subprogram, the algorithm performed is implementation defined.

The execution of a subprogram body is invoked by a subprogram call. For this execution, after establishing the association between the formal and actual parameters, the sequence of statements of the body is executed if the subprogram is not a foreign subprogram; otherwise, an implementation-defined action occurs. Upon completion of the body or implementation-dependent action, if exclusive access to an object of a protected type was granted during elaboration of the declaration of the subprogram (see 12.5), the exclusive access is rescinded. Then, return is made to the caller (and any necessary copying back of formal to actual parameters occurs).

A process or a subprogram is said to be a *parent* of a given subprogram S if that process or subprogram contains a procedure call or function call for S or for a parent of S.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

An *explicit* signal is a signal other than an implicit signal GUARD or other than one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION. The *explicit ancestor* of an implicit signal is found as follows. The implicit signal GUARD has no explicit ancestor. An explicit ancestor of an implicit signal defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION is the signal found by recursively examining the prefix of the attribute. If the prefix denotes an explicit signal, a slice, or a member (see Clause 3) of an explicit signal, then that is the explicit ancestor of the implicit signal. Otherwise, if the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION, this rule is recursively applied. If the prefix is an implicit signal GUARD, then the signal has no explicit ancestor.

If a pure function subprogram is a parent of a given procedure and if that procedure contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object must be declared within the declarative region formed by the function (see 10.1) or within the declarative region formed by the procedure; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. If a pure function is the parent of a given procedure, then that procedure must not contain a reference to an explicitly declared file object (see 4.3.1.4) or to a shared variable (see 4.3.1.3).

Similarly, if a pure function subprogram contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object must be declared within the declarative region formed by the function; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. A pure function must not contain a reference to an explicitly declared file object.

A pure function must not be the parent of an impure function.

The rules of the preceding three paragraphs apply to all pure function subprograms. For pure functions that are not foreign subprograms, violations of any of these rules are errors. However, since implementations cannot in general check that such rules hold for pure function subprograms that are foreign subprograms, a description calling pure foreign function subprograms not adhering to these rules is erroneous.

Example:

— The declaration of a foreign function subprogram:

```
package P is
    function F return INTEGER;
    attribute FOREIGN of F: function is "implementation-dependent information";
end package P;
```

NOTES

1—It follows from the visibility rules that a subprogram declaration must be given if a call of the subprogram occurs textually before the subprogram body, and that such a declaration must occur before the call itself.

2—The preceding rules concerning pure function subprograms, together with the fact that function parameters must be of mode **in**, imply that a pure function has no effect other than the computation of the returned value. Thus, a pure function invoked explicitly as part of the elaboration of a declaration, or one invoked implicitly as part of the simulation cycle, is guaranteed to have no effect on other objects in the description.

3—VHDL does not define the parameter-passing mechanisms for foreign subprograms.

4—The declarative parts and statement parts of subprograms decorated with the 'FOREIGN attribute are subject to special elaboration rules. See 12.3 and 12.4.5.

5—A pure function subprogram must not reference a shared variable. This prohibition exists because a shared variable cannot be declared in a subprogram declarative part and a pure function cannot reference any variable declared outside of its declarative region.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

6—A subprogram containing a wait statement must not have an ancestor that is a subprogram declared within either a protected type declaration or a protected type body.

2.3 Subprogram overloading

Two formal parameter lists are said to have the same *parameter type profile* if and only if they have the same number of parameters, and if at each parameter position the corresponding parameters have the same base type. Two subprograms are said to have the same *parameter and result type profile* if and only if both have the same parameter type profile, and if either both are functions with the same result base type or neither of the two is a function.

A given subprogram designator can be used to designate multiple subprograms. The subprogram designator is then said to be overloaded; the designated subprograms are also said to be overloaded and to overload each other. If two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile.

A call to an overloaded subprogram is ambiguous (and therefore is an error) if the name of the subprogram, the number of parameter associations, the types and order of the actual parameters, the names of the formal parameters (if named associations are used), and the result type (for functions) are not sufficient to identify exactly one (overloaded) subprogram.

Similarly, a reference to an overloaded resolution function name in a subtype indication is ambiguous (and is therefore an error) if the name of the function, the number of formal parameters, the result type, and the relationships between the result type and the types of the formal parameters (as defined in 2.4) are not sufficient to identify exactly one (overloaded) subprogram specification.

Examples:

— Declarations of overloaded subprograms:

```
procedure Dump(F: inout Text; Value: Integer);
procedure Dump(F: inout Text; Value: String);

procedure Check (Setup: Time; signal D: Data; signal C: Clock);
procedure Check (Hold: Time; signal C: Clock; signal D: Data);
```

— Calls to overloaded subprograms:

```
Dump (Sys_Output, 12);
Dump (Sys_Error, "Actual output does not match expected output");

Check (Setup=>10 ns, D=>DataBus, C=>Clk1);
Check (Hold=>5 ns, D=>DataBus, C=>Clk2);
Check (15 ns, DataBus, Clk) ;
    -- Ambiguous if the base type of DataBus is the same type as the base type of Clk.
```

NOTES

1—The notion of parameter and result type profile does not include parameter names, parameter classes, parameter modes, parameter subtypes, or default expressions or their presence or absence.

2—Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be solved in two ways: qualified expressions can be used for some or all actual parameters and for the result, if any; or the name of the subprogram can be expressed more explicitly as an expanded name (see 6.3).

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

2.3.1 Operator overloading

The declaration of a function whose designator is an operator symbol is used to overload an operator. The sequence of characters of the operator symbol must be one of the operators in the operator classes defined in 7.2.

The subprogram specification of a unary operator must have a single parameter, unless the subprogram specification is a method (see 3.5.1) of a protected type. In this latter case, the subprogram specification must have no parameters. The subprogram specification of a binary operator must have two parameters; unless the subprogram specification is a method of a protected type, in which case, the subprogram specification must have a single parameter. If the subprogram specification of a binary operator has two parameters, for each use of this operator, the first parameter is associated with the left operand, and the second parameter is associated with the right operand.

For each of the operators “+” and “–”, overloading is allowed both as a unary operator and as a binary operator.

NOTES

1—Overloading of the equality operator does not affect the selection of choices in a case statement in a selected signal assignment statement, nor does it affect the propagation of signal values.

2—A user-defined operator that has the same designator as a short-circuit operator (i.e., a user-defined operator that overloads the short-circuit operator) is not invoked in a short-circuit manner. Specifically, calls to the user-defined operator always evaluate both arguments prior to the execution of the function.

3—Functions that overload operator symbols may also be called using function call notation rather than operator notation. This statement is also true of the predefined operators themselves.

Examples:

```
type MVL is ('0', '1', 'Z', 'X') ;
function "and" (Left, Right: MVL) return MVL ;
function "or" (Left, Right: MVL) return MVL ;
function "not" (Value: MVL) return MVL ;

signal Q,R,S: MVL ;

Q <= 'X' or '1';
R <= "or" ('0','Z');
S <= (Q and R) or not S;
```

2.3.2 Signatures

A signature distinguishes between overloaded subprograms and overloaded enumeration literals based on their parameter and result type profiles. A signature can be used in an attribute name, entity designator, or alias declaration.

signature ::= [[type_mark { , type_mark }] [**return** type_mark]]

(Note that the initial and terminal brackets are part of the syntax of signatures and do not indicate that the entire right-hand side of the production is optional.) A signature is said to match the parameter and the result type profile of a given subprogram if, and only if, all of the following conditions hold:

— The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

— At each parameter position, the base type denoted by the type mark of the signature is the same as the base type of the corresponding formal parameter of the subprogram.

— If the reserved word **return** is present, the subprogram is a function and the base type of the type mark following the reserved word in the signature is the same as the base type of the return type of the function, or the reserved word **return** is absent and the subprogram is a procedure.

Similarly, a signature is said to match the parameter and result type profile of a given enumeration literal if the signature matches the parameter and result type profile of the subprogram equivalent to the enumeration literal defined in 3.1.1.

Example:

```
attribute BuiltIn of "or" [MVL, MVL return MVL]: function is TRUE;
        -- Because of the presence of the signature, this attribute specification
        -- decorates only the "or" function defined in 2.3.1.

attribute Mapping of JMP [return OpCode] : literal is "001";
```

2.4 Resolution functions

A resolution function is a function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal. Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of the signal or in the declaration of the subtype of the signal. A signal with an associated resolution function is called a resolved signal (see 4.3.1.2).

A resolution function must be a pure function (see 2.1); moreover, it must have a single input parameter of class **constant** that is a one-dimensional, unconstrained array whose element type is that of the resolved signal. The type of the return value of the function must also be that of the signal. Errors occur at the place of the subtype indication containing the name of the resolution function if any of these checks fail (see 4.2).

The resolution function associated with a resolved signal determines the *resolved value* of the signal as a function of the collection of inputs from its multiple sources. If a resolved signal is of a composite type, and if subelements of that type also have associated resolution functions, such resolution functions have no effect on the process of determining the resolved value of the signal. It is an error if a resolved signal has more connected sources than the number of elements in the index type of the unconstrained array type used to define the parameter of the corresponding resolution function.

Resolution functions are implicitly invoked during each simulation cycle in which corresponding resolved signals are active (see 12.6.1). Each time a resolution function is invoked, it is passed an array value, each element of which is determined by a corresponding source of the resolved signal, but excluding those sources that are drivers whose values are determined by null transactions (see 8.4.1). Such drivers are said to be *off*. For certain invocations (specifically, those involving the resolution of sources of a signal declared with the signal kind **bus**), a resolution function may thus be invoked with an input parameter that is a null array; this occurs when all sources of the bus are drivers, and they are all off. In such a case, the resolution function returns a value representing the value of the bus when no source is driving it.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

Example:

```
function WIRED_OR (Inputs: BIT_VECTOR) return BIT is
        constant FloatValue: BIT := '0';
begin
        if Inputs'Length = 0 then
            --  This is a bus whose drivers are all off.
            return FloatValue;
        else
            for I in Inputs'Range loop
                if Inputs(I) = '1' then
                    return '1';
                end if;
            end loop;
            return '0';
        end if;
end function WIRED_OR;
```

2.5 Package declarations

A package declaration defines the interface to a package. The scope of a declaration within a package can be extended to other design units.

```
package_declaration ::=
        package identifier is
            package_declarative_part
        end [ package ] [ package_simple_name ] ;

package_declarative_part ::=
        { package_declarative_item }

package_declarative_item ::=
          subprogram_declaration
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | signal_declaration
        | shared_variable_declaration
        | file_declaration
        | alias_declaration
        | component_declaration
        | attribute_declaration
        | attribute_specification
        | disconnection_specification
        | use_clause
        | group_template_declaration
        | group_declaration
```

If a simple name appears at the end of the package declaration, it must repeat the identifier of the package declaration.

If a package declarative item is a type declaration (i.e., a full type declaration whose type definition is a protected type definition), then that protected type definition must not be a protected type body.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

Items declared immediately within a package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause (see 10.4).

NOTE—Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms, deferred constants, or protected type definitions are declared in the package declaration.

Examples:

— A package declaration that needs no package body:

```
package TimeConstants is
        constant tPLH :     Time := 10 ns;
        constant tPHL :     Time := 12 ns;
        constant tPLZ :     Time := 7 ns;
        constant tPZL :     Time := 8 ns;
        constant tPHZ :     Time := 8 ns;
        constant tPZH :     Time := 9 ns;
end TimeConstants ;
```

— A package declaration that needs a package body:

```
package TriState is
        type Tri is ('0', '1', 'Z', 'E');
        function BitVal (Value: Tri) return Bit ;
        function TriVal (Value: Bit) return Tri;
        type TriVector is array (Natural range <>) of Tri ;
        function Resolve (Sources: TriVector) return Tri ;
end package TriState ;
```

2.6 Package bodies

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

```
package_body ::=
        package body package_simple_name is
            package_body_declarative_part
        end [ package body ] [ package_simple_name ] ;

package_body_declarative_part ::=
        { package_body_declarative_item }

package_body_declarative_item ::=
        subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | shared_variable_declaration
      | file_declaration
      | alias_declaration
      | use_clause
      | group_template_declaration
      | group_declaration
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

The simple name at the start of a package body must repeat the package identifier. If a simple name appears at the end of the package body, it must be the same as the identifier in the package declaration.

In addition to subprogram body and constant declarative items, a package body may contain certain other declarative items to facilitate the definition of the bodies of subprograms declared in the interface. Items declared in the body of a package cannot be made visible outside of the package body.

If a given package declaration contains a deferred constant declaration (see 4.3.1.1), then a constant declaration with the same identifier must appear as a declarative item in the corresponding package body. This object declaration is called the *full* declaration of the deferred constant. The subtype indication given in the full declaration must conform to that given in the deferred constant declaration.

Within a package declaration that contains the declaration of a deferred constant, and within the body of that package (before the end of the corresponding full declaration), the use of a name that denotes the deferred constant is only allowed in the default expression for a local generic, local port, or formal parameter. The result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined by the language.

Example:

```
package body TriState is

    function BitVal (Value: Tri) return Bit is
        constant Bits : Bit_Vector := "0100";
    begin
        return Bits(Tri'Pos(Value));
    end;

    function TriVal (Value: Bit) return Tri is
    begin
        return Tri'Val(Bit'Pos(Value));
    end;

    function Resolve (Sources: TriVector) return Tri is
        variable V: Tri := 'Z';
    begin
        for i in Sources'Range loop
            if Sources(i) /= 'Z' then
                if V = 'Z' then
                    V := Sources(i);
                else
                    return 'E';
                end if;
            end if;
        end loop;
        return V;
    end;

end package body TriState ;
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

2.7 Conformance rules

Whenever the language rules either require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place:

— A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

— A simple name can be replaced by an expanded name in which this simple name is the selector if, and only if, at both places the meaning of the simple name is given by the same declaration.

Two subprogram specifications are said to *conform* if, apart from comments and the above allowed variations, both specifications are formed by the same sequence of lexical elements and if corresponding lexical elements are given the same meaning by the visibility rules.

Conformance is likewise defined for subtype indications in deferred constant declarations.

NOTES

1—A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected name. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P.

2—The subprogram specification of an impure function is never conformant to a subprogram specification of a pure function.

3—The following specifications do not conform since they are not formed by the same sequence of lexical elements:

procedure P (X,Y : INTEGER)
procedure P (X: INTEGER; Y : INTEGER)
procedure P (X,Y : **in** INTEGER)

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

3. Types

This clause describes the various categories of types that are provided by the language as well as those specific types that are predefined. The declarations of all predefined types are contained in package STANDARD, the declaration of which appears in Clause 14.

A type is characterized by a set of values and a set of operations. The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type. The remaining operations of a type are the basic operations and the predefined operators (see 7.2). These operations are each implicitly declared for a given type declaration immediately after the type declaration and before the next explicit declaration, if any.

A *basic operation* is an operation that is inherent in one of the following:

— An assignment (in assignment statements and initializations)
— An allocator
— A selected name, an indexed name, or a slice name
— A qualification (in a qualified expression), an explicit type conversion, a formal or actual part in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type
— A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute

There are five classes of types. *Scalar* types are integer types, floating point types, physical types, and types defined by an enumeration of their values; values of these types have no elements. *Composite* types are array and record types; values of these types consist of element values. *Access* types provide access to objects of a given type. *File* types provide access to objects that contain a sequence of values of a given type. *Protected types* provide atomic and exclusive access to variables accessible to multiple processes.

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to satisfy a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained* (it corresponds to a condition that imposes no restriction). The base type of a type is the type itself.

The set of operations defined for a subtype of a given type includes the operations defined for the type; however, the assignment operation to an object having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression) are implicitly defined by a subtype declaration.

The term *subelement* is used in this standard in place of the term element to indicate either an element, or an element of another element or subelement. Where other subelements are excluded, the term *element* is used instead.

A given type must not have a subelement whose type is the given type itself.

A *member* of an object is one of the following:

— A slice of the object
— A subelement of the object
— A slice of a subelement of the object

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

The name of a class of types is used in this standard as a qualifier for objects and values that have a type or nature of the class considered. For example, the term *array object* is used for an object whose type is an array type; similarly, the term *access value* is used for a value of an access type.

NOTE—The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset.

3.1 Scalar types

Scalar types consist of *enumeration types*, *integer types*, *physical types*, and floating point types. Enumeration types and integer types are called *discrete* types. Integer types, floating point types, and physical types are called *numeric* types. All scalar types are ordered; that is, all relational operators are predefined for their values. Each value of a discrete or physical type has a position number that is an integer value.

```
scalar_type_definition ::=
        enumeration_type_definition  | integer_type_definition
      | floating_type_definition     | physical_type_definition

range_constraint ::=  range range

range ::=
        range_attribute_name
      | simple_expression direction simple_expression

direction ::=  to | downto
```

A range specifies a subset of values of a scalar type. A range is said to be a *null* range if the specified subset is empty.

The range L **to** R is called an *ascending* range; if L > R, then the range is a null range. The range L **downto** R is called a *descending* range; if L < R, then the range is a null range. The smaller of L and R is called the *lower bound*, and the larger, the *upper bound*, of the range. The value V is said to *belong to the range* if the relations (*lower bound* <= V) and (V <= *upper bound*) are both true and the range is not a null range. The operators >, <, and <= in the preceding definitions are the predefined operators of the applicable scalar type.

For values of discrete or physical types, a value V1 is said to be *to the left of* a value V2 within a given range if both V1 and V2 belong to the range and either the range is an ascending range and V2 is the successor of V1, or the range is a descending range and V2 is the predecessor of V1. A list of values of a given range is in *left to right order* if each value in the list is to the left of the next value in the list within that range, except for the last value in the list.

If a range constraint is used in a subtype indication, the type of the expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. Otherwise, the range constraint is not compatible with the subtype.

The direction of a range constraint is the same as the direction of its range.

NOTE—Indexing and iteration rules use values of discrete types.

3.1.1 Enumeration types

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=
        ( enumeration_literal { , enumeration_literal }  )

enumeration_literal ::=  identifier | character_literal
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

The identifiers and character literals listed by an enumeration type definition must be distinct within the enumeration type definition. Each enumeration literal is the declaration of the corresponding enumeration literal; for the purpose of determining the parameter and result type profile of an enumeration literal, this declaration is equivalent to the declaration of a parameterless function whose designator is the same as the enumeration literal and whose result type is the same as the enumeration type.

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character literal.

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each additional enumeration literal is one more than that of its predecessor in the list.

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal is determined according to the rules for overloaded subprograms (see 2.3).

Each enumeration type definition defines an ascending range.

Examples:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS) ;

type BIT is ('0','1') ;

type SWITCH_LEVEL is ('0','1','X') ;          -- Overloads '0' and '1'
```

3.1.1.1 Predefined enumeration types

The predefined enumeration types are CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS.

The predefined type CHARACTER is a character type whose values are the 256 characters of the ISO 8859-1: 1987 [B4][1] character set. Each of the 191 graphic characters of this character set is denoted by the corresponding character literal.

The declarations of the predefined types CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS appear in package STANDARD in Clause 14.

NOTES

1—The first 33 nongraphic elements of the predefined type CHARACTER (from NUL through DEL) are the ASCII abbreviations for the nonprinting characters in the ASCII set (except for those noted in Clause 14). The ASCII names are chosen as ISO 8859-1: 1987 [B11] does not assign them abbreviations. The next 32 (C128 through C159) are also not assigned abbreviations, so names unique to VHDL are assigned.

2—Type BOOLEAN can be used to model either active high or active low logic depending on the particular conversion functions chosen to and from type BIT.

[1]The numbers in brackets correspond to those of the bibliography in Annex D.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

3.1.2 Integer types

An integer type definition defines an integer type whose set of values includes those of the specified range.

integer_type_definition ::= range_constraint

An integer type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the integer type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in an integer type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Integer literals are the literals of an anonymous predefined type that is called *universal_integer* in this standard. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a value of type *universal_integer* into the corresponding value (if any) of the integer type (see 7.3.5).

The position number of an integer value is the corresponding value of the type *universal_integer.*

The same arithmetic operators are predefined for all integer types (see 7.2). It is an error if the execution of such an operation (in particular, an implicit conversion) cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type).

An implementation may restrict the bounds of the range constraint of integer types other than type *universal_integer.* However, an implementation must allow the declaration of any integer type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

Examples:

type TWOS_COMPLEMENT_INTEGER **is range** –32768 **to** 32767;

type BYTE_LENGTH_INTEGER **is range** 0 **to** 255;

type WORD_INDEX **is range** 31 **downto** 0;

subtype HIGH_BIT_LOW is BYTE_LENGTH_INTEGER **range** 0 **to** 127;

3.1.2.1 Predefined integer types

The only predefined integer type is the type INTEGER. The range of INTEGER is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range.

NOTE—The range of INTEGER in a particular implementation is determinable from the values of its 'LOW and 'HIGH attributes.

3.1.3 Physical types

Values of a physical type represent measurements of some quantity. Any value of a physical type is an integral multiple of the primary unit of measurement for that type.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

physical_type_definition ::=
range_constraint
units
primary_unit_declaration
{ secondary_unit_declaration }
end units [*physical_type*_simple_name]

primary_unit_declaration ::= identifier ;

secondary_unit_declaration ::= identifier = physical_literal ;

physical_literal ::= [abstract_literal] *unit*_name

A physical type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the physical type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a physical type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Each unit declaration (either the primary unit declaration or a secondary unit declaration) defines a *unit name*. Unit names declared in secondary unit declarations must be directly or indirectly defined in terms of integral multiples of the primary unit of the type declaration in which they appear. The position numbers of unit names need not lie within the range specified by the range constraint.

If a simple name appears at the end of a physical type declaration, it must repeat the identifier of the type declaration in which the physical type definition is included.

The abstract literal portion (if present) of a physical literal appearing in a secondary unit declaration must be an integer literal.

A physical literal consisting solely of a unit name is equivalent to the integer 1 followed by the unit name.

There is a position number corresponding to each value of a physical type. The position number of the value corresponding to a unit name is the number of primary units represented by that unit name. The position number of the value corresponding to a physical literal with an abstract literal part is the largest integer that is not greater than the product of the value of the abstract literal and the position number of the accompanying unit name.

The same arithmetic operators are predefined for all physical types (see 7.2). It is an error if the execution of such an operation cannot deliver the correct result (i.e., if the value corresponding to the mathematical result is not a value of the physical type).

An implementation may restrict the bounds of the range constraint of a physical type. However, an implementation must allow the declaration of any physical type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

Examples:

```
type DURATION is range –1E18 to 1E18
        units
            fs;                         --  femtosecond
            ps     =   1000 fs;         --  picosecond
            ns     =   1000 ps;         --  nanosecond
            us     =   1000 ns;         --  microsecond
            ms     =   1000 us;         --  millisecond
            sec    =   1000 ms;         --  second
            min    =   60 sec;          --  minute
        end units;

type DISTANCE is range 0 to 1E16
        units
        -- primary unit:
            Å;                          --  angstrom

        -- metric lengths:
            nm     =   10 Å;            --  nanometer
            um     =   1000 nm;         -- micrometer (or micron)
            mm     =   1000 um;         -- millimeter
            cm     =   10 mm;           -- centimeter
            m      =   1000 mm;         -- meter
            km     =   1000 m;          -- kilometer

        -- English lengths:
            mil    =   254000 Å;        --  mil
            inch   =   1000 mil;        --  inch
            ft     =   12 inch;         --  foot
            yd     =   3 ft;            --  yard
            fm     =   6 ft;            --  fathom
            mi     =   5280 ft;         --  mile
            lg     =   3 mi;            --  league
        end units DISTANCE;

variable x: distance;  variable y: duration;  variable z: integer;

x := 5 Å + 13 ft – 27 inch;
y := 3 ns + 5 min;
z := ns / ps;
x := z * mi;
y := y/10;
z := 39.34 inch / m;
```

NOTES

1— The 'POS and 'VAL attributes may be used to convert between abstract values and physical values.

2— The value of a physical literal, whose abstract literal is either the integer value zero or the floating point value zero, is the same value (specifically zero primary units) no matter what unit name follows the abstract literal.

3.1.3.1 Predefined physical types

The only predefined physical type is type TIME. The range of TIME is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range. All

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

specifications of delays and pulse rejection limits must be of type TIME. The declaration of type TIME appears in package STANDARD in Clause 14.

By default, the primary unit of type TIME (1 femtosecond) is the *resolution limit* for type TIME. Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. An implementation may allow a given execution of a model (see 12.6) to select a secondary unit of type TIME as the resolution limit. Furthermore, an implementation may restrict the precision of the representation of values of type TIME and the results of expressions of type TIME, provided that values as small as the resolution limit are representable within those restrictions. It is an error if a given unit of type TIME appears anywhere within the design hierarchy defining a model to be executed, and if the position number of that unit is less than that of the secondary unit selected as the resolution limit for type TIME during the execution of the model, unless that unit is part of a physical literal whose abstract literal is either the integer value zero or the floating-point value zero.

NOTE—By selecting a secondary unit of type TIME as the resolution limit for type TIME, it may be possible to simulate for a longer period of simulated time, with reduced accuracy, or to simulate with greater accuracy for a shorter period of simulated time.

Cross-references: Delay and rejection limit in a signal assignment, 8.4; disconnection, delay of a guarded signal, 5.3; function NOW, 14.2; predefined attributes, functions of TIME, 14.1; simulation time, 12.6.2 and 12.6.3; type TIME, 14.2; Updating a projected waveform, 8.4.1; wait statements, timeout clause in, 8.1; Elaboration of a declarative part, 12.3.

3.1.4 Floating point types

Floating point types provide approximations to the real numbers.

floating_type_definition ::= range_constraint

A floating type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the floating type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a floating type definition must be a locally static expression of some floating point type, but the two bounds need not have the same floating point type. (Negative bounds are allowed.)

Floating point literals are the literals of an anonymous predefined type that is called *universal_real* in this standard. Other floating point types have no literals. However, for each floating point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating point type (see 7.3.5).

The same arithmetic operations are predefined for all floating point types (see 7.2). A design is erroneous if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the floating point type).

An implementation must choose a representation for all floating-point types except for *universal_real* that conforms either to IEEE Std 754 or to IEEE Std 854; in either case, a minimum representation size of 64 bits is required for this *chosen representation*.

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

An implementation may restrict the bounds of the range constraint of floating point types other than type *universal_real*. However, an implementation must allow the declaration of any floating point type whose range is wholly contained within the bounds allowed by the chosen representation.

NOTE—An implementation is not required to detect errors in the execution of a predefined floating point arithmetic operation, since the detection of overflow conditions resulting from such operations might not be easily accomplished on many host systems.

3.1.4.1 Predefined floating point types

The only predefined floating point type is the type REAL. The range of REAL is host-dependent, but it is guaranteed to be the largest allowed by the chosen representation. It is defined with an ascending range.

NOTE—The range of REAL in a particular implementation is determinable from the values of its 'LOW and 'HIGH attributes.

3.2 Composite types

Composite types are used to define collections of values. These include both arrays of values (collections of values of a homogeneous type) and records of values (collections of values of potentially heterogeneous types).

composite_type_definition ::=
    array_type_definition
  | record_type_definition

An object of a composite type represents a collection of objects, one for each element of the composite object. It is an error if a composite type contains elements of file types or protected types. Thus an object of a composite type ultimately represents a collection of objects of scalar or access types, one for each noncomposite subelement of the composite object.

3.2.1 Array types

An array object is a composite object consisting of elements that have the same subtype. The name for an element of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of its elements.

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
    array (index_subtype_definition { , index_subtype_definition })
        of *element*_subtype_indication

constrained_array_definition ::=
    array index_constraint **of** *element*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete*_subtype_indication | range

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

An array object is characterized by the number of indices (the dimensionality of the array); the type, position, and range of each index; and the type and possible constraints of the elements. The order of the indices is significant.

A one-dimensional array has a distinct element for each possible index value. A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given index are all the values that belong to the corresponding range; this range of values is called the *index range*.

An unconstrained array definition defines an array type and a name denoting that type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the elements are as in the type definition. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The values of the left and right bounds of each index range are not defined, but must belong to the corresponding index subtype; similarly, the direction of each index range is not defined. The symbol <> (called a *box*) in an index subtype definition stands for an undefined range (different objects of the type need not have the same bounds and direction).

A constrained array definition defines both an array type and a subtype of this type:

— The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the element subtype indication is that of the constrained array definition and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

— The array subtype is the subtype obtained by imposition of the index constraint on the array type.

If a constrained array definition is given for a type declaration, the simple name declared by this declaration denotes the array subtype.

The direction of a discrete range is the same as the direction of the range or the discrete subtype indication that defines the discrete range. If a subtype indication appears as a discrete range, the subtype indication must not contain a resolution function.

Examples:

— Examples of constrained array declarations:

```
type MY_WORD is array (0 to 31) of BIT ;
        -- A memory word type with an ascending range.

type DATA_IN is array (7 downto 0) of FIVE_LEVEL_LOGIC ;
        -- An input port type with a descending range.
```

— Example of unconstrained array declarations:

```
type MEMORY is array (INTEGER range <>) of MY_WORD ;
        -- A memory array type.
```

— Examples of array object declarations:

```
signal DATA_LINE : DATA_IN ;
        -- Defines a data input line.

variable MY_MEMORY : MEMORY (0 to 2**n–1) ;
        -- Defines a memory of 2^n 32-bit words.
```

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004

NOTE—The rules concerning constrained type declarations mean that a type declaration with a constrained array definition such as

type T **is array** (POSITIVE **range** MINIMUM **to** MAX) **of** ELEMENT;

is equivalent to the sequence of declarations

subtype *index_subtype* **is** POSITIVE **range** MINIMUM **to** MAX;

type *array_type* **is array** (index_subtype **range** <>) **of** ELEMENT;

subtype T **is** *array_type (index_subtype)*;

where *index_subtype* and *array_type* are both anonymous. Consequently, T is the name of a subtype and all objects declared with this type mark are arrays that have the same index range.

3.2.1.1 Index constraints and discrete ranges

An index constraint determines the index range for every index of an array type and, thereby, the corresponding array bounds.

For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal or an attribute, and if the type of both bounds (prior to the implicit conversion) is the type *universal_integer.* Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determined independently of the context, but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration scheme (see 8.9) or a generation scheme (see 9.7).

If an index constraint appears after a type mark in a subtype indication, then the type or subtype denoted by the type mark *must* not already impose an index constraint. The type mark *must* denote either an unconstrained array type or an access type whose designated type is such an array type. In either case, the index constraint *must* provide a discrete range for each index of the array type, and the type of each discrete range *must* be the same as that of the corresponding index.

An index constraint is *compatible* with the type denoted by the type mark if, and only if, the constraint defined by each discrete range is compatible with the corresponding index subtype. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no elements. An array value *satisfies* an index constraint if at each index position the array value and the index constraint have the same index range. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.)

The index range for each index of an array object is determined as follows:

— For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and thereby, the index range for each index of the object). The same requirement exists for the subtype indication of an element declaration, if the type of the record element is an array type, and for the element subtype indication of an array type definition, if the type of the array element is itself an array type.

— For a constant declared by an object declaration, the index ranges are defined by the initial value, if the subtype of the constant is unconstrained; otherwise, they are defined by this subtype (in which case the initial value is the result of an implicit subtype conversion).

— For an attribute whose value is specified by an attribute specification, the index ranges are defined by the expression given in the specification, if the subtype of the attribute is unconstrained; otherwise, they are defined by this subtype (in which case the value of the attribute is the result of an implicit subtype conversion).

— For an array object designated by an access value, the index ranges are defined by the allocator that creates the array object (see 7.3.6).

Published by IEC under licence from IEEE. © 2004 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 20:11:09 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 61691-1-1-2004