

IEEE Std 1029.1-1998

(Revision of
IEEE Std 1029.1-1991)

IEEE Std 1029.1-1998

IEEE Standard for VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES) Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

28 May 1999

SH94689

To order IEEE standards...

Call 1. 800. 678. IEEE (4333) in the US and Canada.

Outside of the US and Canada:

1. 732. 981. 0600

To order by fax:

1. 732. 981. 9667

IEEE business hours: 8 a.m.–4:30 p.m. (EST)

For on-line access to IEEE standards information...

Via the World Wide Web:

<http://standards.ieee.org/>

Via ftp:

[stdsbbs.ieee.org](ftp://stdsbbs.ieee.org)

0-7381-1445-6

IEEE Standard for VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES) Language Reference Manual

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 16 September 1998

IEEE-SA Standards Board

Abstract: This standard is a formal notation intended for use in all phases of the development of electronic systems. Because it is both machine-readable and human-readable, it supports the verification and testing of hardware designs; the communication of hardware design and test verification data; and the maintenance, modification, and procurement of hardware systems. This standard provides the syntactic and semantic framework for the unambiguous expression and aggregation of digital test data and timing information necessary to completely describe a test or set of tests for a digital system. WAVES digital test data (stimulus and expected responses) is described at the logic level. Voltage and current values are not described by WAVES and are beyond the scope of this standard.

Keywords: CAD, design automation, design verification, digital design, test vector, VHDL

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 28 May 1999. Printed in the United States of America.

Print: ISBN 0-7381-1445-6 SH94689
PDF: ISBN 0-7381-1446-4 SS94689

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

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

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

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

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 all 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 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

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.

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; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

(This introduction is not a part of IEEE Std 1029.1-1998, IEEE Standard WAVES Language Reference Manual.)

This standard is a formal notation intended for use in all phases of the development of electronic systems. Because it is both machine-readable and human-readable, it supports the verification and testing of hardware designs; the communication of hardware design and test verification data; and the maintenance, modification, and procurement of hardware systems.

This document specifies IEEE Std 1029.1-1998, which is a revision of IEEE Std 1029.1-1991. The WAVES Analysis and Standardization Group (WASG) of the Computer Society of the IEEE started the development of IEEE Std 1029.1-1998 in September 1995. This document is based on the IEEE Std 1076-1993 VHDL Language Reference Manual (VHDL LRM) and follows closely the format and organization of the original VHDL LRM.

The WASG would like to express sincere gratitude to everyone that contributed to the development of the 1993 VHDL standard. Without their ardent efforts and attention to detail this standard would have required considerably more time and work to develop.

The WAVES 98 standardization effort consisted of five major phases: definition of WAVES 98 requirements, language design, language documentation, design validation, and balloting. The result of the requirements definition phase indicated that a significant reimplementation of the language was necessary to fulfill the desired language requirements. The WAVES 98 proposed standard is the embodiment of this redesign and implementation. In bringing the WAVES standard into alignment with VHDL 93, and refocusing the WAVES language around design verification and test bench development, WAVES now supports the design verification of VHDL models at any level of abstraction, from high-level performance model, to detailed gate-level modeling.

The following working documents were developed during each phase of the standardization effort:

Requirements Definition:	WAVES 98 Requirements WAVES 98 Language Issues Document
Language Documentation:	Draft Language Reference Manual
Design Validation:	Various Examples & Tools
Ballot Response Document:	Balloting

Participants

Numerous volunteers in North America, Europe, and Asia-Pacific contributed to the development of WAVES 98. In addition, this standard was developed by a working group consisting of the following members who attended two or more meetings, provided text, or submitted comments on more than two drafts of this standard:

Dave Barton
Todd A. De Long
Steven L. Drager

Christopher J. Flynn
James P. Hanna
Robert G. Hillman
Herb L. Hirsch

Gregory D. Peterson
Lee Shombert
Ranganadha R. Vemuri

The following persons were on the balloting committee:

Mostapha Aboulhamid
Phillip R. Acuff
Guy Adam
John Ainscough
Robert E. Anderson
Stephen A. Bailey
Jean-Yves Barbier
Jeffrey S. Barkley
J. Bhasker
Robert W. Bierig
Kenneth M. Butler
Alexander Cherfas
Luc Claesen
Edmond S. Cooley
Tedd Corman
David C. Crohn
Danny C. Davis
Todd A. DeLong
Steven L. Drager
Douglas D. Dunlop
Hazem El Tahawy
John A. Eldon
Christopher J. Flynn
Geoffrey Frank
Kenji Goto
Brian S. Griffin
Steve Grout
Andrew Guyler
Michael J. Haney
Arthur Hann
William A. Hanna

Gary L. Hardenburg
Randolph E. Harr
Rich Hatcher
Robert G. Hayes
John Hillawi
Robert G. Hillman
Mitsuaki Ishikawa
Patrick Johnson
Masaru Kakimoto
Takashi Kambe
Al Karalius
Osamu Karatsu
Jake Karrfalt
Masamichi Kavarabayashi
Paul B. Knese
Masayuki Koyama
Charles Krumm
Vello Kukk
Arun Kumar
Rolf Lundin
Serge Maginot
Maqsoodul Mannan
Michael S. Martin
Paul J. Menchini
Jean P. Mermet
Gerald T. Michael
Marlyn G. Miner
Egbert Molenkamp
Gabe Moretti
David S. Morris

Gerald Musgrave
James Nagy
Hiroshi Nakamura
Kevin O'Brien
Gaspard Pantano
Curtis Parks
Gregory D. Peterson
Jean Pouilly
Narayanan Ramachandran
William E. Reeves
Robert M. Rolfe
Eric Sacher
Francesco Sforza
Ravi Shankar
John W. Sheppard
C. J. Richard Shi
Toru Shonai
William R. Simpson
Joseph J. Stanco
Peter Ting
Jose A. Torres
Louis Y. Ungar
Ranganadha R. Vemuri
Martin J. Walter
Ronald Waxman
J. Richard Weger
John Willis
Glenn Wright
Will W. Young
Alex N. Zamfirescu
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 16 September 1998, it had the following membership:

Richard J. Holleman, *Chair*

Donald N. Heirman, *Vice Chair*

Judith Gorman, *Secretary*

Satish K. Aggarwal
Clyde R. Camp
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Thomas F. Garrity
Ruben D. Garzon

James H. Gurney
Jim D. Isaak
Lowell G. Johnson
Robert Kennelly
E. G. "Al" Kiener
Joseph L. Koepfinger*
Stephen R. Lambert
Jim Logothetis
Donald C. Loughry

L. Bruce McClung
Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Hans E. Weinrich
Donald W. Zipse

*Member Emeritus

Noelle D. Humenick
IEEE Standards Project Editor

Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	2
1.3	Relationship to other standards.....	2
1.4	Structure and terminology of this document.....	2
2.	WAVES test sets.....	5
2.1	Test set semantics	5
2.2	Test set structure	6
2.3	WAVES header files.....	18
3.	Subprograms and packages.....	20
3.1	Subprogram declarations	20
3.2	Subprogram bodies	23
3.3	Subprogram overloading.....	25
3.4	Package declarations.....	27
3.5	Package bodies.....	28
3.6	Conformance rules	29
4.	Types.....	30
4.1	Scalar types	31
4.2	Composite types.....	37
4.3	Access types.....	42
4.4	File types.....	45
5.	Declarations	48
5.1	Type declarations	48
5.2	Subtype declarations	49
5.3	Objects	50
6.	Names	57
6.1	Definition of names.....	57
6.2	Simple names	58
6.3	Selected names.....	58
6.4	Indexed names	60
6.5	Slice names	61
6.6	Attribute names	61
7.	Expressions	62
7.1	Definition of expressions	62
7.2	Operators.....	63
7.3	Operands	71
7.4	Static expressions.....	78
7.5	Universal expression.....	80

8.	Sequential statements.....	81
8.1	Assertion statement.....	81
8.2	Report statement	82
8.3	Variable assignment statement	82
8.4	Procedure call statement	84
8.5	If statement.....	84
8.6	Case statement	84
8.7	Loop statement.....	86
8.8	Next statement	87
8.9	Exit statement.....	87
8.10	Return statement	87
8.11	Null statement	88
9.	Scope and visibility.....	89
9.1	Declarative region.....	89
9.2	Scope of declarations	89
9.3	Visibility	90
9.4	Use clauses.....	91
9.5	The context of overload resolution	92
10.	Design units and their analysis	94
10.1	Design units	94
10.2	Design libraries	94
10.3	Context clauses	95
10.4	Order of analysis	96
11.	Elaboration and execution.....	97
11.1	Elaboration of a declarative part	97
11.2	Dynamic elaboration.....	99
11.3	Execution of a test set	99
12.	Lexical elements	101
12.1	Character set.....	101
12.2	Lexical elements, separators, and delimiters	103
12.3	Identifiers	104
12.4	Abstract literals	105
12.5	Character literals	107
12.6	String literals.....	107
12.7	Bit string literals.....	107
12.8	Comments	109
12.9	Reserved words.....	109
13.	Predefined language environment.....	113
13.1	Predefined attributes	113
13.2	Package STANDARD	119
13.3	Package TEXTIO.....	126
13.4	Package WAVES_INTERFACE.....	129
13.5	Package WAVES_OBJECTS	133

14.	Predefined WAVES 1164 compatible library packages.....	148
14.1	WAVES_1164_Declarations package.....	148
14.2	WAVES_1164_FRAMES package	149
14.3	WAVES_1164_UTILITIES package	153
14.4	WAVES standard packages	154
15.	Predefined external data file format.....	156
15.1	Data file format	156
15.2	File slice reader	159
16.	Header file.....	162
16.1	Header file format	162
Annex A	(informative) Syntax summary.....	165
Annex B	(informative) Glossary.....	176
Annex C	(informative) Guidelines for translation for electrical test	189
Annex D	(informative) Related standards	191

IEEE Standard for VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES) Language Reference Manual

1. Overview

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

The WAVES language is an application of the VHSIC Hardware Description Language (IEEE Std 1076-1993¹). This document uses the term “VHDL” to refer to the VHSIC Hardware Description Language and the term “VHDL LRM” to refer to IEEE Std 1076-1993.

At the time of its publication, this document was the authoritative definition of WAVES. 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.

The primary audiences of this standard are the implementers 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 manual. These resources generally focus on how to use the language, rather than how a WAVES-compliant tool is required to behave.

1.1 Scope

WAVES provides the syntactic and semantic framework for the unambiguous expression and aggregation of digital test data and timing information necessary to completely describe a test or set of tests for a digital system. WAVES digital test data (stimulus and expected responses) is described at the logic level; voltage and current values are not described by WAVES and are beyond the scope of this standard.

¹IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (www.standards.ieee.org/).

1.2 Purpose

WAVES test sets are applicable to the verification of a VHDL model hierarchy and/or the electrical testing of a physical electronic device, MCM, board, or system. WAVES test sets are also applicable as a common industry exchange format for digital test data and timing.

VHDL was chosen as the basis for WAVES for two reasons:

- a) To allow WAVES to serve as a standard stimulus and response format for VHDL
- b) To obviate the necessity of developing an entirely new language with new syntax and semantics

WAVES, therefore, benefits from its VHDL basis by sharing language attributes. It supports VHDL design verification by providing a standard test methodology and format entirely within the existing VHDL tool base.

1.3 Relationship to other standards

The syntax of WAVES is defined as a proper subset of VHDL and is given in Annex A. Since WAVES is a syntactic subset as well as an application of VHDL, it follows that the WAVES language also conforms to the static and dynamic semantics governing this subset of VHDL. It also follows that any legal WAVES test set can be wholly simulated within all VHDL simulation environments that comply with the VHDL LRM.

Although WAVES is based on the VHDL standard, WAVES is a fully defined, stand-alone language in its own right. WAVES is not only applicable to VHDL environments; any number of tools and/or applications of WAVES to digital electronics design and test may exist.

The standard WAVES predefined library packages described in Clause 14 are based on the IEEE Std 1164-1993 Multi-valued Logic System (MVL9). These library packages fully support the verification of IEEE Std 1164-1993 compliant VHDL models.

The WAVES standard is also compatible with any VHDL package-based standard, trial use standard, or standard practice, provided these packages comply with IEEE 1076-1993 and this standard.

1.4 Structure and terminology of this document

This manual 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 subclauses.

Each clause 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 WAVES, “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 WAVES source text.

- b) To the WAVES test set developer, “must” denotes that the characteristics of WAVES are natural consequences of the language definition. The test set developer is required to adhere to the constraint implied by the characteristic.
- c) To the WAVES test set user, “must” denotes that the characteristics of the test sets are natural consequences of the language definition. The test set user can depend on the characteristics of the test set implied by its WAVES source text.

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

1.4.1 Syntactic description

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

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

formal_parameter_list

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

- b) Boldface words are used to denote reserved words. A complete list of reserved words is given in 12.9. Examples are:

array	type	character	Boolean
frame	delay_time	frame_set	file_slice

A reserved words word is used only in those places indicated by the syntax. Reserved words include all constants and type marks declared in the packages STANDARD, TEXTIO, WAVES_INTERFACE and WAVES_OBJECTS (see 13.2, 13.3, 13.4, and 13.5).

- 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:

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 two following productions are equivalent:

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

- 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:

$$\text{term} ::= \text{factor} \{ \text{multiplying_operator factor} \}$$

$$\text{term} ::= \text{factor} \mid \text{term multiplying_operator factor}$$
- 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.

1.4.2 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 refer to subprograms and enumeration literals defined in the packages STANDARD, TEXTIO, WAVES_INTERFACE, and WAVES_OBJECTS (see 13.2, 13.3, 13.4, and 13.5).

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

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.

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

Prior to this clause are several pieces of introductory material; following the final clause are some annexes and an index. The front matter, annexes, and index serve to orient and otherwise aid the user of this manual but are not part of the definition of WAVES.

Some clauses of this standard contain examples, notes, and cross-references to other clauses of the manual; 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 subclause 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 manual. Examples, notes, and cross-references are not part of the definition of the language.

2. WAVES test sets

This clause describes the semantics of WAVES test sets in both VHDL and electrical test environments, as well as the structure and construction of test sets. This clause also provides a detailed description of all required WAVES type and data declarations, as well as guidelines for their appropriate packaging. A discussion of WAVES external data files is also presented. Finally, test set configuration management issues are discussed, and the WAVES test set header file is described.

2.1 Test set semantics

WAVES provides the syntactic and semantic framework for the unambiguous expression and aggregation of all of the data and information necessary to completely describe a test, or set of tests, for the verification of a VHDL model hierarchy and/or the electrical testing of a physical electronic device, MCM, board, or system. The waveform that is produced by a given WAVES test set is the result of the interaction between the test set and its environment, regardless of whether the environment is VHDL or an electrical test environment. Although this standard defines the intent of a given WAVES test set without regard to the usage environment, the particular application of a WAVES test set in a given environment is not restricted by the standard.

WAVES test sets can be envisioned as sequential programs that, when executed, generate a waveform for all of the signals that the test set describes. This waveform defines the sequence of *events* (timing, logic value, and direction) for all of its signals for the duration of the test set. The waveform is not, however, required to define event traces for every signal. The signals of the generated waveform may be unidirectional (input, output) or bidirectional. It is the responsibility of the environment to recognize when I/O switching occurs and take the appropriate action. Here we define the semantics of the events generated by the WAVES test set. In the remainder of this document we will use the term Unit Under Test (UUT) to refer to both a VHDL model hierarchy and an instance of physical electronic hardware.

- a) Events on the generated waveform with a direction specifier of *stimulus* are applied to the corresponding inputs of the given UUT and define the timing and logic values being driven into the UUT.
- b) Events on the generated waveform with a direction specifier of *response* describe the *expected* timing and logic values of the corresponding outputs of the given UUT.
- c) Events on the generated waveform with a direction specifier of *unspecified only* describe the timing and logic values of the corresponding inputs/outputs of the given UUT.

When taken in aggregate, the WAVES test set defines the collection of input stimulus and the expected UUT responses for the entire duration of the test. Therefore, a WAVES test set is the specification of the functionality of some subset of the entire functionality of a UUT.

2.1.1 VHDL simulation and model verification

In a VHDL test bench environment, a WAVES test set is realized as one or more concurrent procedure calls. The signals generated by the WAVES test set are declared in the VHDL test bench. The WAVES test set, in the context of a VHDL simulation, is simply one or more concurrent processes. The execution of a WAVES test set during a VHDL simulation results in the scheduling of transactions on the drivers associated with the waveform signals. All transactions scheduled as a result of executing a WAVES test set obey the semantics of the VHDL transport delay mechanism. The implementation of the WAVES standard packages defines the semantics of WAVES as executing sequential processes.

2.1.2 Electrical test

In an electrical test environment a WAVES test set is translated into the native timing and pattern file formats of the electrical test station. Each electrical test environment carries unique physical limitations and con-

straints with regard to issues such as: accuracy, edge placement, the number of timing generators available, etc. Therefore, it may be the case that a legal WAVES test set is not translatable to a given test environment. In fact, only a restricted subset of legal WAVES test sets can be expected to be translatable to a given electrical test environment. Issues with regard to generating WAVES test sets that can be translated to common test environments are discussed in Annex C. It is the responsibility of the translation tool to assure that the intent of the WAVES test set, as defined by this standard, is preserved by the translation process and that particular tester constraints are not violated.

2.2 Test set structure

A WAVES test set is the logical collection of data and information necessary to completely describe one or more waveforms. A WAVES test set consists of two or more WAVES files, zero or more *external* data files, and a *header* file.

Figure 1 illustrates the contents of a WAVES test set. Elements of Figure 1 that are drawn with a dotted outline indicate optional files. One of the WAVES files must contain the required *test pins* declaration and another WAVES file must contain the required *waveform generator procedure* declarations. The WAVES files must conform to the syntax and semantics of WAVES. The external data files contain data in a format that can be used by the waveform generator procedure in the construction of the waveform. The WAVES header file provides test set configuration management information and specifies how the test set is constructed from the WAVES files and external data files. The structure and content of the WAVES files, the pre-defined external data file, and the header file are described in the remainder of this clause.



Figure 1—The WAVES test set

2.2.1 Required declarations

There are a number of pieces of information that are required to fully specify a WAVES test set. These include the names associated with each of the UUT pins, logic values, and the timing with which logic values are applied to the UUT.

Issues regarding the grouping or packaging of these declarations are also discussed. Many of these required declarations will be defined once for a particular logic value system and placed into a VHDL library. Clause 14 provides an example of this define-and-reuse approach for the IEEE Std 1164-1993 logic value system. Other logic value systems may be defined to support different levels of modeling abstraction such as token-based system performance modeling. However, doing so may complicate translations to electrical test (see Annex C).

Figure 2 illustrates the structure of the WAVES test set from a functional perspective. Elements of Figure 2 drawn with a dotted outline indicate optional items.

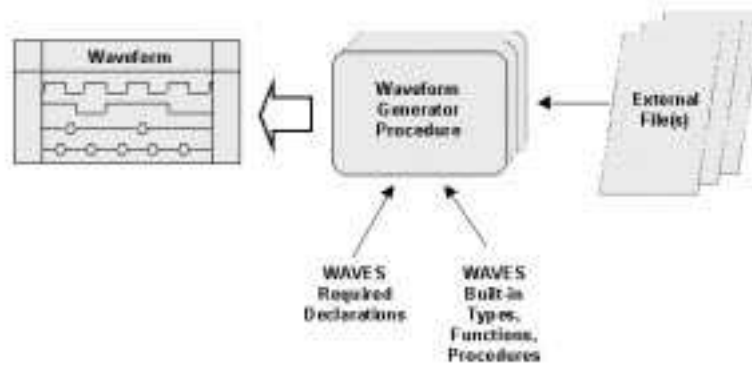


Figure 2—Test set structure

The waveform generator procedure is the main element of the test set and serves to aggregate the required declarations and WAVES built-ins into the procedure that generates the waveform described by the test set. There are seven required declarations:

- a) Logic value
- b) Waves logic vector
- c) Pin codes
- d) Maximum frame events
- e) Frames and frame sets
- f) Test pins
- g) The waveform generator procedure

These required declarations are presented and described in the next subclauses.

2.2.1.1 Logic value

WAVES requires the declaration of the *logic values* used by a WAVES test set. This declaration must define a scalar or composite type or subtype.

```

logic_value_type_declaration ::=
    type LOGIC_VALUE is logic_value_type_definition ;

logic_value_type_definition ::=
    scalar_type_definition
    | composite_type_definition

logic_value_subtype_declaration ::=
    subtype LOGIC_VALUE is logic_value_subtype_indication;

logic_value_subtype_indication ::= logic_value_type_mark [ constraint ];

```

This declaration defines all of the legal values that the signals of the waveform may exhibit. The logic value is the most basic atomic element of WAVES. The sequencing of logic values on a given signal are what gives the waveform its shape and meaning.

Examples:

```
-- A logic value declaration based on IEEE Std 1164-1993.
subtype LOGIC_VALUE is Std_Ulogic;

-- An arbitrary logic value declaration.
type LOGIC_VALUE is ( FETCH, DECODE, EXECUTE, STORE );

-- A token-based logic value declaration.
type TOKEN is
  record
    STANDBY_NORMAL:STANDBY_NORMAL_TYPE;
    MODE           : INTERROGATE_MODE_TYPE;
    COUNT          : INTERROGATE_COUNT_TYPE;
    INTERVAL       : INTERROGATE_INTERVAL_TYPE;
    TRIGGER_SELECT : TRIGGER_SELECT_TYPE;
    TRIGGER        : BOOLEAN;
  end record;

subtype LOGIC_VALUE is TOKEN;
```

NOTE—Logic value should be declared as a subtype of a type defined in a common package and made visible to both the VHDL entity and the WAVES test set when WAVES is to be used for VHDL model verification. This will assure that type conversions are unnecessary within the test bench between the signals of the waveform and the ports of the VHDL entity.

2.2.1.2 WAVES logic vector

WAVES requires the declaration of a composite type named *WAVES logic vector* that is used to aggregate all of the signals of the generated waveform into one object. The WAVES logic vector must be declared as an unconstrained array with a single integer range. The base type of logic vector must be the same as that of the logic value declaration. Signals of this type are used convey the waveform events to the environment.

```
logic_vector_subtype_declaration ::=
  subtype WAVES_LOGIC_VECTOR is array ( integer_type_mark range <> ) of
    LOGIC_VALUE;
  | subtype WAVES_LOGIC_VECTOR is type_mark;
```

Examples:

```
-- A WAVES logic vector declaration based on IEEE Std 1164-1993.
subtype WAVES_LOGIC_VECTOR is Std_Ulogic_vector;

-- An arbitrary WAVES logic vector declaration.
subtype WAVES_LOGIC_VECTOR is array ( positive range <> ) of LOGIC_VALUE;
```

2.2.1.3 Pin codes

The declaration of a string constant named *pin codes* is required by WAVES. This constant declaration defines the legal characters that can appear in pin code strings that are applied to the signals of the waveform. These pin code strings may be generated algorithmically or be read from external data files. The declaration of this constant is also required to constrain data structures in the WAVES standard package WAVES_INTERFACE.

```
pin_codes_constant_declaration ::=
    constant PIN_CODES : STRING := string_expression;
```

These characters may be used to represent one event on a signal or entire sequences of events on a signal of the waveform. The meaning ascribed to the characters in the pin code string constant is defined in the frame set declaration which is discussed later in this clause. This string expression that defines the constant may include any character from the enumerated type CHARACTER that is defined in the package STANDARD (see 13.2). No character may appear in this string expression more than once. Uppercase letters are distinct from lowercase letters. A null pin code string may be declared for cases such as system performance modeling or any case where pin codes are not used. If a null pin code string is declared, no references may be made to the NEW_FRAME_SET function, the NEW_FILE_SLICE function, the READ_FILE_SLICE procedure, or any variant of the apply procedure that has a formal parameter with the type mark of **pin_code_string**.

Examples:

```
-- A pin codes declaration based on IEEE Std 1164-1993.
constant PIN_CODES : STRING := "X01ZWLH-";

-- An arbitrary pin codes declaration.
constant PIN_CODES : STRING := "1234";

-- A null pin codes declaration.
constant PIN_CODES : STRING := "";
```

2.2.1.4 Maximum frame events

WAVES requires the declaration of a positive integer constant named *max frame events*.

```
max_frame_events_constant_declaration ::=
    constant MAX_FRAME_EVENTS : POSITIVE := integer_expression;
```

The declaration of this constant is required to constrain data structures in the WAVES standard package WAVES INTERFACE. The requirement of this declaration is strictly an artifact of implementing WAVES as an application of VHDL. The use of this required WAVES declaration in non-VHDL environments is not defined or restricted in any way by this standard.

Example:

```
constant MAX_FRAME_EVENTS : POSITIVE := 3;
```

2.2.1.5 Frames and frame sets

The *frame* is one of the basic building blocks for expressing signal timing in WAVES (see 13.3). A frame is a sequence of zero or more WAVES events and a *direction specifier*. WAVES events are *logic value, event time* pairs. Frames aggregate events into waveform segments or shapes (e.g. a clock pulse or compare window) and define the direction of the events within the frame. The direction associated with a frame may be UNSPECIFIED, STIMULUS, or RESPONSE. The semantics of the direction specifiers are discussed in 2.1. Frames that contain a single event are declared using the FRAME_EVENT function. Frames that contain multiple events are declared using the FRAME_ELIST function. The details regarding the operation of the FRAME_EVENT function and the FRAME_ELIST function are given in 13.4.

Examples:

```
-- An empty frame declaration.
constant EMPTY_FRAME : frame;

-- A single event frame declaration with a direction
-- specifier of stimulus.
constant NRZ_FRAME : frame := FRAME_EVENT( ('1', 30 ns), STIMULUS );

-- A multiple event frame declaration with a direction
-- specifier of stimulus.
constant CLK_FRAME : frame :=
    FRAME_ELIST( (('0', 0 ns),
                  ('1', 50 ns),
                  ('0', 100 ns)), STIMULUS )

-- A multiple event frame declaration with a direction
-- specifier of response.
constant WINDOW : frame :=
    FRAME_ELIST( (('1', 0 ns),
                  ('1', 5 ns),
                  ('1', 45 ns)), RESPONSE )
```

An empty frame declared as a variable and assigned dynamically.

```
variable TOKEN_DATA : frame;
.
.
.
loop
.
.
.
    read a token

    TOKEN_DATA := FRAME_EVENT( (TOKEN, 0 ns) );
.
.
.
end loop;
```

NOTES:

1 — The first example declares a frame that consists of zero events and a direction specifier that is UNSPECIFIED.

2 — The last example demonstrates the dynamic assignment of a frame that has one event and a direction specifier that is UNSPECIFIED. The value of TOKEN is assumed to be read from a file and then placed in the frame TOKEN_DATA within the execution loop in some WAVES subprogram.

Frames that define similar shapes are typically grouped together into *frame sets*. The number of elements in a frame set is given by the number of characters in the pin code string declaration. The frame set associates one pin code with one possibly empty frame for each character of the pin code string. This association defines the meaning of the pin code with regard to a given frame set. Several unique frame sets may be declared in a given WAVES test, each defining a different interpretation of its pin codes. Frame sets are declared using the NEW_FRAME_SET function. The details regarding the operation of the NEW_FRAME_SET function are given in 13.4.

Examples:

-- A frame set for positive going clock pulses.

constant CLK_FRAMES : **frame_set** :=

```
NEW_FRAME_SET(
  ( ('0', FRAME_EVENT( ('0', 0 ns), STIMULUS ) ),
    ('1', FRAME_ELIST( (('0', 0 ns),
                        ('1', 50 ns),
                        ('0', 100 ns)), STIMULUS ) ),
    ('L', FRAME_EVENT( ('L', 0 ns), STIMULUS ) ),
    ('H', FRAME_ELIST( (('L', 0 ns),
                        ('H', 50 ns),
                        ('L', 100 ns)), STIMULUS ) )
  )
);
```

-- A function that returns a frame set with event times that

-- are based on its input parameters.

function WINDOW(T1, T2 : **event_time**) **return frame_set** is

begin

assert T1 < T2

report "Timing violation in WINDOW frame set. " &
"The inequality: T1 < T2 Must hold."

severity FAILURE;

return NEW_FRAME_SET(

```
( ('X', FRAME_ELIST( (('1', 0 ns), ('X', T1), ('1', T2)), RESPONSE ) ),
  ('0', FRAME_ELIST( (('1', 0 ns), ('0', T1), ('1', T2)), RESPONSE ) ),
  ('1', FRAME_ELIST( (('1', 0 ns), ('1', T1), ('1', T2)), RESPONSE ) ),
  ('Z', FRAME_ELIST( (('1', 0 ns), ('Z', T1), ('1', T2)), RESPONSE ) ),
  ('W', FRAME_ELIST( (('1', 0 ns), ('W', T1), ('1', T2)), RESPONSE ) ),
  ('L', FRAME_ELIST( (('1', 0 ns), ('L', T1), ('1', T2)), RESPONSE ) ),
  ('H', FRAME_ELIST( (('1', 0 ns), ('H', T1), ('1', T2)), RESPONSE ) ),
  ('1', FRAME_EVENT( ('1', 0 ns), RESPONSE ) )
  )
);
```

end WINDOW;

NOTES:

1—The first example illustrates the declaration of a frame that uses the WAVES 1164 logic value system to define a positive-going pulse. Each character code not explicitly listed in the call to the NEW_FRAME_SET function is associated with empty frames by default.

2—A convenient method for defining frame sets is through a WAVES function declaration. This method supports the development of a common reusable frame set library. The second example is from the WAVES 1164 frames package that is given in its entirety in Clause 14.

2.2.1.6 Test pins

The declaration of an enumerated type named *test pins* is required for each WAVES test set. This declaration must contain one enumeration value for each pin specified by the test set. The number of enumerals declared by this type is used to constrain data structures in the WAVES standard package WAVES_OBJECTS.

```
test_pins_type_definition ::=  
    type TEST_PINS is ( enumeration_literal { , enumeration_literal } )
```

The identifiers enumerated by this declaration represent the names of the signals of the UUT. Any legal WAVES enumeration declaration may be used. The order of the enumerals given by this declaration defines the order of the signals of the waveform. This declaration also defines the order of the characters in pin code strings that are applied to the signals of the waveform. If any WAVES predefined external data files are associated with the test set, this declaration also defines the column ordering of the pin codes in the external files. There is a one-to-one correspondence between waveform signals and pin code string characters. This same correspondence exists between waveform signals and the columns of external files. The identifiers given in this declaration are the only legal elements of the WAVES **pin_set** construct. The pin set construct is discussed later in this document.

Examples:

```
-- A test pins declaration for a D Flip Flop.  
type TEST_PINS is ( CLOCK, D, Q, Q_BAR );  
  
-- A test pins declaration for an arbitrarily complex board.  
type TEST_PINS is ( IOBUS_AD_N_63, IOBUS_AD_N_62, IOBUS_AD_N_61,  
    IOBUS_AD_N_60, IOBUS_AD_N_59, IOBUS_AD_N_58,  
        IOBUS_AD_N_57, IOBUS_AD_N_56, IOBUS_AD_N_55,  
        IOBUS_AD_N_54, IOBUS_AD_N_53,  
        .  
        .  
        .  
    PCI_IRDY_N, PCI_TRDY_N, PCI_DEVSEL_N, PCI_PERR_N,  
    PCI_SERR_N, PCI_STOP_N, PCI_RST_N, PCI_REQ0_N,  
    PCI_REQ1_N, PCI_GNT_N, PCI_GA_1, PCI_GA_0,  
    PCI_CLK, P1, P2, P3, P4 );
```

2.2.1.7 Waveform generator procedure

The final WAVES required declaration is that of the waveform generator procedure (WGP). The WGP provides and defines the interface between the WAVES test set and its environment. All of the semantics of the WAVES test set are encapsulated within the waveform generator procedure. This procedure completely describes construction of the waveform by using:

- a) information defined by the other required declarations
- b) information defined by WAVES standard packages
- c) information defined internally
- d) one or more optional external data files

The details of the WGP declaration are presented in this subclause.

The WGP declares interfaces that allow data to flow between the WAVES test set and its environment. These interfaces are WGP interface signal parameters. Waveform generator procedures are the only WAVES subprograms that are allowed to declare signal parameters. Waveform generator procedures are required to declare only the type and number of interface signal parameters that are described in this subclause.

All waveform generator procedure declarations are required to conform to one of two basic forms. The first form defines the interface to a WAVES test set that consists of a single waveform generator procedure. This form requires the declaration of its first parameter as class **signal**, mode **out**, with the type mark of

waves_port_list (see 13.4). This form provides a waveform signal interface constrained to the size of the test pins type declaration.

Example:

```
procedure WAVEFORM( signal WPL : out waves_port_list );
```

The second form defines the interface to a WAVES test set that may consist of multiple waveform generator procedures. This form requires the declaration of its first parameter as class **signal**, mode **out**, with the type mark of WAVES_LOGIC_VECTOR, and its second parameter as class **signal**, mode **out**, with a type mark of **waves_direction_vector** (see 13.4). This form provides a waveform signal interface that is unconstrained.

Example:

```
procedure WAVEFORM( signal WLV : out WAVES_LOGIC_VECTOR;  
                    signal WDV : out waves_direction_vector );  
  
procedure CLOCK_GEN( signal WLV : out WAVES_LOGIC_VECTOR;  
                    signal WDV : out waves_direction_vector );
```

Each waveform generator procedure in a multiple WGP test set contributes a portion of the total set of waveform signals. No signal overlay is allowed between individual WGPs. A WGP that declares only parameters required for conformance to these two basic forms is said to define a *simple* waveform.

A WAVES test set may also provide interfaces that allow the test set to synchronize with events occurring in the environment. WAVES provides two such synchronization mechanisms called *match* and *handshake*. A WAVES test set may define a *match mode* waveform, a *handshake mode* waveform, or a combination of the two.

A WGP that defines a match mode waveform requires the declaration of two additional parameters. These additional parameters must immediately follow the previously described required parameters. The first additional parameter must be of class **signal**, mode **out**, with the type mark **waves_match_request**. The second additional match mode parameter must be of class **signal**, mode **in**, with the type mark **waves_match_ack** (see 13.4).

Example:

```
procedure WAVEFORM( signal WPL : out waves_port_list;  
                    signal WMR : out waves_match_request;  
                    signal ACK : in waves_match_ack );
```

A WGP that defines a handshake mode waveform also requires the declaration of two additional parameters. These additional parameters must immediately follow any previously described required parameters. The first additional parameter must be of class **signal**, mode **out**, with the type mark **waves_handshake_request**. The second additional handshake mode parameter must be of class **signal**, mode **in**, with the type mark **waves_handshake_ack** (see 13.4).

Example:

```
procedure WAVEFORM( signal WPL : out waves_port_list;  
                    signal WHR : out waves_handshake_request;  
                    signal ACK : in waves_handshake_ack );
```

When defining a joint match mode and handshake mode waveform, the declaration of four additional parameters is required. These additional parameters must immediately follow any previously described required parameters. The first additional parameter must be of class **signal**, mode **out**, with the type mark **waves_match_request**. The second additional parameter must be of class **signal**, mode **in**, with the type mark **waves_match_ack**. The third additional parameter must be of class **signal**, mode **out**, with the type mark **waves_handshake_request**. And finally, the fourth additional parameter must be of class **signal**, mode **in**, with the type mark **waves_handshake_ack**.

Example:

```
procedure WAVEFORM( signal WPL : out waves_port_list;  
                    signal WMR : out waves_match_request;  
                    signal M_ACK : in waves_match_ack;  
                    signal WHR : out waves_handshake_request;  
                    signal HS_ACK : in waves_handshake_ack );
```

Waveforms may also include an interface signal that allows textual information to be provided to the environment. This textual information is known as a waveform *tag*. A tag serves only to annotate the waveform; it carries no semantic information. Waveforms that provide tag information to the environment declare one additional parameter. This additional parameter must immediately follow any parameters required to define a simple, match, handshake, or match-handshake waveform. This parameter must be of class **signal**, mode **out**, with the type mark **waves_tag** (see 13.4).

Example:

```
procedure WAVEFORM( signal WPL      : out    waves_port_list;  
                    signal WMR      : out    waves_match_request;  
                    signal M_ACK    : in      waves_match_ack;  
                    signal WHR      : out    waves_handshake_request;  
                    signal HS_ACK   : in      waves_handshake_ack;  
                    signal TAG      : out    waves_tag );
```

In addition to the previously described waveforms, a WGP may represent a *parametric* waveform. A parametric waveform is defined by a WGP that declares any parameters in addition to those previously mentioned. Any number of additional parameters may be declared in parametric waveforms. All additional parameters used in the definition of a parametric waveform must have a class of **constant** and mode **in**.

Example:

```
procedure WAVEFORM( signal WPL      : out    waves_port_list;  
                    constant NOISE : in      REAL );
```

Waveform generator procedures are the only WAVES subprograms that may have signal parameters. They are also the only subprograms that reference the predefined WAVES procedures APPLY, DELAY, MATCH, HANDSHAKE, and TAG (see 13.5). Further, waveform generator procedures must not appear in any subprogram declarative region (must not be nested). Other than these constraints, the waveform generator procedures obey the same rules governing procedure subprograms (see 3.1).

Example:

```

procedure WAVEFORM( signal WPL : out waves_port_list ) is

  variable TIMING : frame_data :=
    BUILD_FRAME_DATA(
      ( ( +CLOCK, PULSE_HIGH( 50 ns, 100 ns ) ),
        ( +D,      NON_RETURN( 40 ns ) ),
        ( +Q,      WINDOW( 75 ns, 95 ns ) )
        ( +Q_BAR, WINDOW( 75 ns, 95 ns ) )
      )
    );

  constant VECTORS : pin_code_list( 1 to 5 ) := ( "1110",
                                                    "0010",
                                                    "1001",
                                                    "1110",
                                                    "1001" );

  begin
    for I in VECTORS'range loop
      APPLY( WPL, VECTORS( I ), TIMING );
      DELAY( 100 ns );
    end loop;
  end WAVEFORM;

```

NOTES:

1—The previous example demonstrates a WGP that applies the pin codes given in VECTORS to a D Flip-Flop using a clock period of 100 ns. The execution loop applies one vector of data, stored in the constant VECTORS, every 100 ns until the entire array of vectors has been exhausted. The scheduling of events is defined by the **frame_data** variable TIMING (see 13.5). The plus signs proceeding the identifiers: CLOCK, D, Q, and Q_BAR are unary **pin_set** operators (see 13.5.1).

2—This WGP uses the WAVES 1164 packages for its definition of all of the required declarations except for test pins. The test pins declaration for the D Flip Flop device is given in the examples in 2.2.1.6. The WAVES 1164 packages are given in Clause 14.

2.2.2 Packaging and visibility issues

As mentioned in 2.2.1, many of the required declarations can be defined once, packaged appropriately, and placed in a library. Several of these required declarations are specific to the logic value system being used and are applicable for all test sets that use the same logic value system. Other declarations are very specific to the UUT for which the test set is intended. All required declarations must be defined within a package declaration. The grouping of these declarations within packages and package visibility issues are discussed in this subclause.

Figure 3 shows the information presented in this subclause graphically. The arrows in Figure 3 represent use clauses in the packages from which they emanate and illustrate package visibility. The two ovals labeled IEEE and WORK represent design libraries and indicate which library each WAVES package is compiled into. It is assumed that the indicated WAVES packages are precompiled into the IEEE library that is distributed with various VHDL tools.

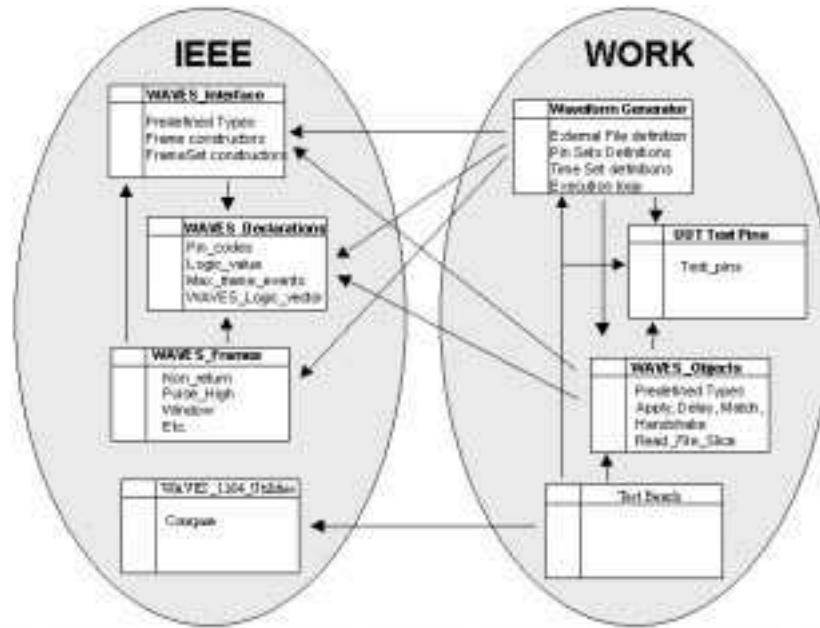


Figure 3—Packaging and visibility

2.2.2.1 Packaging

The reusable declarations should be packaged to provide ease of use and flexibility for their users. It is useful to group these declarations into two packages. In this scenario one declaration package contains the declaration of logic value, WAVES logic vector, pin codes, and max frame events, while another frames package contains the declaration of frame and frame set constants or functions. This is the approach taken in the definition of the WAVES 1164 standard packages given in Clause 14. The reasons for this recommended approach are simplicity and convenience.

This approach to packaging the required declarations for the WAVES 1164 packages provides a simple and convenient way to provide the necessary visibility to the WAVES standard packages. This allows precompiled versions of the declarations package, frames package, and WAVES_INTERFACE package to reside in the IEEE library, greatly simplifying the user's view of the WAVES test set. It must be noted that the WAVES standard packages do not require visibility to anything declared in the frames package. However, the frames package does require visibility to the declarations package and the WAVES_INTERFACE package.

The test pins and waveform generator declarations are required declarations specific to a particular UUT for which the test set is intended. These declarations must reside in different files and packages. The waveform generator procedure must also be declared in its own package.

2.2.2.2 Visibility

The WAVES standard packages require visibility to the required declarations. These visibility requirements are discussed in the context of the packaging approach chosen for the WAVES 1164 packages. The WAVES standard package WAVES_INTERFACE requires visibility to the LOGIC_VALUE, PIN_CODES, and MAX_FRAME_EVENTS declarations. Visibility to each of these declarations is provided by placing context items in the WAVES_INTERFACE package declaration.

Example:

```
library IEEE;
use IEEE.WAVES_1164_DECLARATIONS.all;
package WAVES_INTERFACE is

.
.
.
```

The WAVES standard package WAVES_OBJECTS also requires visibility to the LOGIC_VALUE and PIN_CODES declarations. Visibility to the WAVES_INTERFACE package and the TEST_PINS declaration is also required. The test pins declaration package must also be made visible to the WAVES_OBJECTS package. The test pins visibility requirement implies that the WAVES_OBJECTS package must be compiled along with each WAVES test set. For this reason WAVES_OBJECTS can not reside in a precompiled library. These visibility requirements are satisfied by placing context items in the WAVES_OBJECTS package declaration.

Example:

```
library IEEE;
use IEEE.WAVES_1164_DECLARATIONS.all;
use IEEE.WAVES_INTERFACE.all;
use WORK.UUT_TEST_PINS.all;
package WAVES_OBJECTS is

.
.
.
```

The waveform generator procedure requires visibility to the frames package, WAVES_INTERFACE, WAVES_OBJECTS, and the test pins declaration package. The WGP may also require visibility to the declarations package if any references to logic value, pin codes, or any other required declaration are made within the WGP. Visibility to additional packages may be required based on the requirements of a particular WGP application.

Example:

```
library IEEE;
use STD.TEXTIO.all;
use IEEE.WAVES_1164_FRAMES.all;
use IEEE.WAVES_1164_DECLARATIONS.all;
use IEEE.WAVES_INTERFACE.all;
use WORK.WAVES_OBJECTS.all;
use WORK.UUT_TEST_PINS.all;
package WGP is

.
.
.
```

When a WAVES test set is used for VHDL model verification, the test bench requires visibility to the package in which the waveform generator procedure is declared. Visibility to the test pins package and the WAVES_OBJECTS package are required when match mode and handshake mode waveforms are applied to a VHDL model hierarchy. Visibility to additional packages may be required based on the requirements of a particular test bench application.

Example:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.WAVES_1164_UTILITIES.all;
use WORK.UUT_TEST_PINS.all;
use WORK.WAVES_OBJECTS.all;
use WORK.WGP.all;
entity TEST_BENCH is
.
.
.
```

2.2.3 External data files

A WAVES test set may include one or more external data files that do not conform to the syntax of the WAVES language. These external data files may contain data that conforms to the WAVES predefined data file format or a user-defined data format.

2.2.3.1 Predefined data files

The predefined data file format defined by this standard provides a simple, yet powerful, mechanism for expressing pattern data typically used during design verification, fault simulation, and electrical test. A detailed description of the WAVES predefined data file format is given in Clause 15 along with a complete description of the processing provided by the WAVES built-in read file slice procedure.

2.2.3.2 User-defined data files

User-defined data files are permitted in a WAVES test set. This capability provides the user with the mechanism for defining a general purpose interface to arbitrary data files. This is the mechanism by which users import the data used in system performance modeling and simulation. There are no restrictions on the format of user-defined data files. Any file that can be processed by the WAVES file I/O is considered a legal WAVES external data file.

A test set that includes any user-defined data files is required to provide a procedure that reads the external file and converts the data in the external file into standard WAVES objects. All procedures that read a user-defined external file are required to conform to the syntax and semantics of WAVES. The meaning of the user-defined external file depends on the supplied reader.

2.3 WAVES header files

A WAVES test set may consist of many files which include WAVES packages and external data files. It may also be convenient to consider one or more VHDL model hierarchies as a part of the test set. For this reason WAVES test sets include a header file. This file provides a single source of information regarding the configuration of the test set. The WAVES test set header file specifies information that is not available within the WAVES files or the external files, yet that is necessary or useful to describe the WAVES test set completely.

The header file specifies the file names of the WAVES files, VHDL files, and references standard WAVES units in a legal order for analysis. The order of analysis for WAVES files and standard WAVES units is the same as that of the equivalent VHDL design units. In the case of multiple WAVES units within a single WAVES file, the WAVES units are listed in the proper order for analysis. Clause 16 completely describes the syntax and content of the WAVES header file.

Referenced WAVES units are references to the WAVES standard packages that may not be shipped with the test set. Putting their names in the header file specifies that they are to be included in the analysis order as though the WAVES units themselves appeared there.

The WAVES test set header file also allows the specification of a mapping of logical file names to physical file names for external files. In addition to the previous information, the header file specifies how the test set is to be identified, the names of the waveform generator procedures, and any additional comments as necessary.

3. Subprograms and packages

Subprograms define algorithms for computing values or exhibiting behavior. They are used as computational resources to convert between values of different types, to generate pin code strings (or any other WAVES object), or to define and partition functionality necessary to the definition of a particular test set. 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: general purpose and waveform generator. General purpose subprograms may be procedures or functions. Waveform generator subprograms are procedures. A procedure call is a statement; a function call is an expression and returns a value. Functions return the same value each time they are called with the same values as actual parameters and cannot update objects not within their scope. All WAVES function subprograms are pure. The definition of a subprogram can be given in two parts:

- a) A subprogram declaration defining its calling conventions
- b) 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. Package specifications and package bodies are collectively referred to as design entities or design units in this manual.

3.1 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 ( waveform_generator_formal_parameter_list )
    | procedure designator [ ( formal_parameter_list ) ]
    | [ pure ] function designator [ ( formal_parameter_list ) ]
      return type_mark

waveform_generator_formal_parameter_list ::=
    simple_waveform_interface [ ; match_mode_interface ] [ ; handshake_mode_interface ]
    [ ; tag_interface ] [ parametric_interface_list ] )

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

simple_waveform_interface ::=
    signal identifier : out waves_port_list
    | signal identifier : out waves_logic_vector; signal identifier : out waves_direction_vector

match_mode_interface ::=
    signal identifier : out waves_match_request; signal identifier : in waves_match_ack

handshake_mode_interface ::=
    signal identifier : out waves_handshake_request; signal identifier : in
    waves_handshake_ack

```

```

tag_interface ::= signal identifier : out waves_tag

parametric_interface_list ::= { ; parametric_interface_element }

parametric_interface_element ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]

```

A subprogram is a procedure or a function. 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), and the subtype of the returned value (the *result subtype*). Function specifications may be preceded by the keyword **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 3.3.1). 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.

A waveform generator subprogram is a procedure that is declared with a waveform generator formal parameter list is a waveform generator procedure. The declaration of a waveform generator procedure (WGP) specifies its designator and its formal parameters. A WGP specifies a minimum of one formal interface signal parameter. In addition to the required interface signal parameters, a WGP may specify formal interface signal parameters that support match mode, and/or handshake mode operations. A formal interface signal parameter may also be specified to support the textual annotation (tagging) of the waveform. Finally, a list of formal constant parameters may be specified. No other formal parameters are allowed in a WGP declaration.

NOTE—All subprograms can be called recursively.

3.1.1 Formal parameters

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

```
formal_parameter_list ::= parameter_interface_list
```

Formal parameters of general purpose subprograms may be constants, variables, or files. In the first two cases, the mode of a parameter determines how a given formal parameter may be accessed within the subprogram. The mode and class of a formal parameter may also determine how such access is implemented. In the third 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 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**. If no object class is explicitly given, **constant** is assumed.

In a subprogram call, the actual designator (see 5.3.2.2) associated with a formal of class **variable** must be 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 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.

3.1.1.1 Constant and variable parameters

For parameters of class **constant** or **variable**, only the values of the actual or formal parameters 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 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 5.3.2.2), 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 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 in error if its effect depends on which mechanism is selected by the implementation.

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 in error.

3.1.1.2 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 4.4.1) 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 4.4.1) 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.

After a given subprogram call has executed, the actual file object associated with a given file parameter is open if and only if the formal parameter is also open. Similarly, after a given subprogram call has executed, 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 .

3.1.1.3 Waveform generator signal parameters

Only waveform generator subprograms may declare formal parameters of class **signal**, and only according to the waveform generator subprogram syntax rules. 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**, the actual signal is associated with the corresponding formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, a reference to the formal signal parameter within an expression is equivalent to a reference to the actual signal.

A subprogram contains a driver for each formal signal parameter of mode **out** declared in its subprogram specification.

For a signal parameter of mode **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 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 those 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. It is also an error if the mode of the formal is **in** and if the value of each element of the actual array does not belong to the element subtype of the formal.

3.2 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 }

subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration

subprogram_statement_part ::=
    { sequential_statement }

subprogram_kind ::= procedure | function

```

The declaration of a subprogram is optional. In the absence of a declaration, the subprogram specification of the subprogram body acts as the declaration. For each subprogram declaration, there must be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body must conform (see 3.6) to the subprogram specification of the declaration. Furthermore, both the declaration and the body must occur immediately within the same declarative region (see 9.1).

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 waveform generator subprogram specification or declaration appears in the declarative part of any subprogram. Waveform generator procedures are declared only within packages.

The algorithm performed by a subprogram is defined by the sequence of statements in the subprogram statement part.

A subprogram call invokes the execution of a subprogram body. For this execution, after establishing the association between the formal and actual parameters, the sequence of statements of the body is executed. Upon completion of the body, a return is made to the caller (and any necessary copying back of formal to actual parameters occurs).

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

If a function subprogram is a parent of a given procedure and if that procedure contains a reference to an explicitly declared variable object, a slice, or subelement (or slice thereof), then that object must be declared within the declarative region formed by the function (see 9.1) or within the declarative region formed by the procedure. If a function is the parent of a given procedure, then that procedure must not contain a reference to an explicitly declared file object (see 5.3.1.3).

Similarly, if a function subprogram contains a reference to an explicitly declared variable object, a slice, or subelement (or slice thereof), then that object must be declared within the declarative region formed by the function. A function must not contain a reference to an explicitly declared file object.

The rules of the preceding four paragraphs apply to all function subprograms. Violations of any of these rules are errors.

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 function subprograms, together with the fact that function parameters may only be of mode **in**, imply that a function has no effect other than the computation of the returned value. Thus, a function invoked explicitly as part of the elaboration of a declaration is guaranteed to have no effect on other objects in the description.

3.3 Subprogram overloading

Two formal parameter lists are said to have the same *parameter type profile* 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 in several subprogram specifications. 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 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 specification.

Examples:

Declarations of overloaded subprograms:

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

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

Calls to overloaded subprograms:

```
Dump (Sys_Output, 12) ;
```

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

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).

3.3.1 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 6.2.

The subprogram specification of a unary operator must have a single parameter. The subprogram specification of a binary operator must have 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 .

2—A user-defined operator that has the same designator as a short-circuit operator (that is, that overloads the short-circuit operator) is not invoked in a short-circuit manner (see 7.2). 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 (see 7.3.3). 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 ;

variable Q,R,S: MVL ;

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

3.3.2 Signatures

A signature distinguishes between overloaded subprograms and overloaded enumeration literals based on their parameter and result type profiles.

A signature is said to *match* the parameter and result type profile of a given subprogram only if all of the following conditions hold:

- a) The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram.
- b) 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.
- c) 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 4.4.1.

3.4 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
    | file_declaration

```

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

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 section (see 9.4).

NOTE—Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms or deferred constants 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;  
end package TriState ;
```

3.5 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  
  | file_declaration
```

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 5.3.1.1), 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 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;

end package body TriState ;

```

3.6 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) A numeric literal can be replaced by a different numeric literal if and only if both have the same value.
- b) 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 previously 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 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 );

```

4. 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 packages STANDARD, WAVES_INTERFACE, and WAVES_OBJECTS, the declaration of which appears in Clause 13.

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 four classes of types:

- a) *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.
- b) *Composite* types are array and record types; values of these types consist of element values.
- c) *Access* types provide access to objects of a given type.
- d) *File* types provide access to objects that contain a sequence of values of a given type.

The set of possible values for an object of a given type can be subjected to a condition 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 with an optional 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 manual 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 either:

- a) A slice of the object,
- b) A subelement of the object, or

- c) A slice of a subelement of the object.

The name of a class of types is used in this manual as a qualifier for objects and values that have a type 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.

4.1 Scalar types

Scalar types consist of:

- a) Enumeration types
- b) Integer types
- c) Physical types
- d) 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 ::=
    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* $\leq$ V) and ($V \leq$ *upper bound*) are both true and the range is not a null range. The operators $>$, $<$, and $\leq$ 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 with V2 the successor of V1 or the range is a descending range with V2 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.

4.1.1 Enumeration types

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=  
    ( enumeration_literal { , enumeration_literal } )  
  
enumeration_literal ::= identifier | character_literal
```

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 3.3).

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'
```

4.1.1.1 Predefined enumeration types

The predefined enumeration types are:

- a) CHARACTER
- b) BIT
- c) BOOLEAN
- d) SEVERITY_LEVEL
- e) FILE_OPEN_KIND
- f) FILE_OPEN_STATUS
- g) DIRECTION

The predefined type CHARACTER is a character type whose values are the 256 characters of the ISO 8859-1: 1987 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 12. The declarations of the predefined enumeration type DIRECTION appear in the package WAVES_INTERFACE in Clause 13.

NOTES:

1—The first 17 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 13). The ASCII names are chosen as ISO 8859-1: 1987 does not assign them abbreviations. The next 16 (C128 through C159) are also not assigned abbreviations, so names unique to WAVES 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.

4.1.2 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 anonymous. Its range is selected by the implementation; this range must wholly contain 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 those of an anonymous predefined type 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 $-2\ 147\ 483\ 647$ and $+2\ 147\ 483\ 647$ 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 ;

4.1.2.1 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 $-2\,147\,483\,647$ to $+2\,147\,483\,647$. It is defined with an ascending range.

NOTE—The range of `INTEGER` in a particular implementation may be determined from the 'LOW' and 'HIGH' attributes.

4.1.3 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.

```
physical_type_definition ::=  
    range_constraint  
    units  
        base_unit_declaration  
        { secondary_unit_declaration }  
    end units [ physical_type_simple_name ]  
  
base_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 anonymous. Its range is selected by the implementation; this range must wholly contain 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 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.

A position number corresponds 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 (that is, 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 $-2\,147\,483\,647$ and $+2\,147\,483\,647$ inclusive.

Examples:

type DURATION is range $-1\text{E}18$ to $1\text{E}18$

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 $1\text{E}16$

units

-- primary unit:

```
A;           -- angstrom
```

-- metric lengths:

```
nm = 10 A;-- 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 A; -- 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 A + 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 0 or the floating point value 0 is the same value (specifically 0 primary units), no matter what unit name follows the abstract literal.

4.1.3.1 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 $-2\,147\,483\,647$ to $+2\,147\,483\,647$. It is defined with an ascending range. All specifications of delays must be of type TIME. The declaration of type TIME appears in package STANDARD in Clause 13.

4.1.4 Floating point types

Floating point types provide approximations to the real numbers. Floating point types are useful for test sets in which the precise characterization of a floating point calculation is not important or not determined.

`floating_type_definition ::= range_constraint`

A floating type definition defines both a type and a subtype of that type. The type is anonymous. Its range is selected by the implementation; this range must wholly contain 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 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 those of an anonymous predefined type 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 in error 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 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 $-1.0\text{E}38$ and $+1.0\text{E}38$ inclusive. The representation of floating point types must include a minimum of six decimal digits of precision.

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 may not be easily accomplished on many host systems.

4.1.4.1 Predefined floating point types

The only predefined floating point type is the type REAL. The range of REAL depends on the host, but it is guaranteed to include the range $-1.0\text{E}38$ to $+1.0\text{E}38$ inclusive. It is defined with an ascending range.

NOTE—The range of REAL in a particular implementation may be determined from the 'LOW and 'HIGH attributes.

4.2 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. A composite type must only contain elements of the following types:

- a) Scalar
- b) Composite
- c) Access

Elements of file types are not allowed in a composite type. 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.

4.2.1 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_indicationi.Index subtypes:where used;

index_subtype_definition ::= type_mark range <>

index_constraint ::= ( discrete_range { , discrete_range } )

discrete_range ::= discrete_subtype_indication | range
```

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 ele-

ments 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 $\langle \rangle$ (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:

- a) 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.
- b) 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  $\langle \rangle$ ) of MY_WORD ;
-- A memory array type.
```

Examples of array object declarations:

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

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

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  $\langle \rangle$ ) 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.

4.2.1.1 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 8.7).

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 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 components. 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:

- a) For a variable 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.
- b) 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).
- c) 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).
- d) For an interface object declared with a subtype indication that defines a constrained array subtype, the index ranges are defined by that subtype.
- e) For a formal parameter of a subprogram that is of an unconstrained array type and that is associated in whole (see 5.3.2.2), the index ranges are obtained from the corresponding association element in the applicable subprogram call.
- f) For a formal parameter of a subprogram that is of an unconstrained array type and whose subelements are associated individually (see 5.3.2.2), the index ranges are obtained as follows:

The directions of the index ranges of the formal parameter are that of the type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.

If the index ranges for an interface object or member of an interface object are obtained from the corresponding association element (when associating in whole) or elements (when associating individually), then

they are determined either by the actual parts or by the formal parts of the association elements, depending upon the mode of the interface object, as follows:

- For an interface object or member of an interface object whose mode is in or inout if the actual part includes a conversion function or a type conversion, then the result type of that function or the type mark of the type conversion must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object or value denoted by the actual designators.

Examples:

```
type Word is array (NATURAL range <>) of BIT;
type Memory is array (NATURAL range <>) of Word (31 downto 0);

constant A_Word: Word := "10011";
-- The index range of A_Word is 0 to 4
```

4.2.1.2 Predefined array types

The predefined discrete array types STRING and BIT_VECTOR are defined in package STANDARD in Clause 13.

- a) The values of the predefined type STRING are one-dimensional arrays of predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE.
- b) The values of the predefined type BIT_VECTOR are one-dimensional arrays of the predefined type BIT, indexed by values of the predefined subtype NATURAL.

The predefined array types EVENT_LIST, FRAME_EVENTS, FRAME_LIST, TAGGED_FRAME_LIST, FRAME_SET, and FRAME_SET_ARRAY are defined in the package WAVES_INTERFACE. These predefined types and their use are discussed in detail in Clause 13.

- c) The values of the predefined type EVENT_LIST are one-dimensional arrays of the predefined type EVENT, indexed by values of the predefined subtype POSITIVE.
- d) The values of the predefined type FRAME_EVENTS are one-dimensional arrays of the predefined type EVENT, constrained by the required declaration MAX_FRAME_EVENTS and indexed by values of the predefined subtype POSITIVE.
- e) The values of the predefined type FRAME_LIST are one-dimensional arrays of the predefined type FRAME, indexed by values of the predefined subtype POSITIVE.
- f) The values of the predefined type TAGGED_FRAME_LIST are one-dimensional arrays of the predefined type TAGGED_FRAME, indexed by values of the predefined subtype POSITIVE.
- g) The values of the predefined type FRAME_SET are one-dimensional arrays of the predefined type FRAME, constrained by the length of the required declaration PIN_CODES and indexed by values of the predefined subtype POSITIVE.
- h) The values of the predefined type FRAME_SET_ARRAY are one-dimensional arrays of the predefined type FRAME_SET, indexed by values of the predefined subtype POSITIVE.

The predefined array types FRAME_SET_MAP, WAVE_TIMING_LIST, PINSET, TEST_PINS_LIST, WAVES_DIRECTION_VECTOR, PINS_FORMAT_LIST, PIN_CODE_LIST, INDEX_SLICE_LIST, TIMED_SLICE_LIST, WTIME_SLICE_LIST, and FILE_SLICE_LIST are defined in the package WAVES_OBJECTS. These predefined types and their use are discussed in detail in Clause 13.