BS IEC 62530-2:2021

BSI Standards Publication

SystemVerilog

Part 2: Universal Verification Methodology Language Reference Manual

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

National foreword

This British Standard is the UK implementation of IEC 62530-2:2021.

The UK participation in its preparation was entrusted to Technical Committee EPL/501, Electronic Assembly Technology.

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2021
Published by BSI Standards Limited 2021

ISBN 978 0 539 17207 2

ICS 25.040.01; 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 August 2021.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

IEC 62530-2

Edition 1.0 2021-07

INTERNATIONAL STANDARD IEEE Std 1800.2™

SystemVerilog –
Part 2: Universal Verification Methodology Language Reference Manual

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040.01; 35.060

ISBN 978-2-8322-9974-6

Warning! Make sure that you obtained this publication from an authorized distributor.

® Registered trademark of the International Electrotechnical Commission

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Contents

1. Overview ... 13

1.1 Scope ... 13
1.2 Purpose ... 13
1.3 Conventions used ... 13

2. Normative references ... 16

3. Definitions, acronyms, and abbreviations ... 16

3.1 Definitions ... 16
3.2 Acronyms and abbreviations ... 17

4. UVM class reference ... 18

5. Base classes ... 20

5.1 Overview ... 20
5.2 uvm_void ... 20
5.3 uvm_object ... 20
5.4 uvm_transaction ... 31
5.5 uvm_port_base #(IF) ... 36
5.6 uvm_time ... 40

6. Reporting classes ... 43

6.1 Overview ... 43
6.2 uvm_report_message ... 43
6.3 uvm_report_object ... 46
6.4 uvm_report_handler ... 52
6.5 Report server ... 55
6.6 uvm_report_catcher ... 59

7. Recording classes ... 65

7.1 uvm_tr_database ... 65
7.2 uvm_tr_stream ... 67
7.3 UVM links ... 71

8. Factory classes ... 76

8.1 Overview ... 76
8.2 Factory component and object wrappers ... 76
8.3 UVM factory ... 82

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.
Full document details: BS IEC 62530-2:2021

9. Phasing ... 89

9.1 Overview ... 89
9.2 Implementation ... 89
9.3 Phasing definition classes ... 89
9.4 uvm_domain ... 98
9.5 uvm_bottomup_phase ... 99
9.6 uvm_task_phase ... 100
9.7 uvm_topdown_phase ... 101
9.8 Predefined phases ... 102

10. Synchronization classes ... 107

10.1 Event classes ... 107
10.2 uvm_event_callback ... 110
10.3 uvm_barrier ... 111
10.4 Pool classes ... 113
10.5 Objection mechanism ... 114
10.6 uvm_heartbeat ... 119
10.7 Callbacks classes ... 121

11. Container classes ... 126

11.1 Overview ... 126
11.2 uvm_pool #(KEY,T) ... 126
11.3 uvm_queue #(T) ... 128

12. UVM TLM interfaces ... 131

12.1 Overview ... 131
12.2 UVM TLM 1 ... 131
12.3 UVM TLM 2 ... 148

13. Predefined component classes ... 168

13.1 uvm_component ... 168
13.2 uvm_test ... 181
13.3 uvm_env ... 182
13.4 uvm_agent ... 182
13.5 uvm_monitor ... 183
13.6 uvm_scoreboard ... 183
13.7 uvm_driver #(REQ,RSP) ... 184
13.8 uvm_push_driver #(REQ,RSP) ... 184
13.9 uvm_subscriber ... 185

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

14. Sequences classes ... 187

14.1 uvm_sequence_item ... 187
14.2 uvm_sequence_base ... 191
14.3 uvm_sequence #(REQ,RSP) ... 200
14.4 uvm_sequence_library ... 201

15. Sequencer classes ... 206

15.1 Overview ... 206
15.2 Sequencer interface ... 206
15.3 uvm_sequencer_base ... 211
15.4 Common sequencer API ... 217
15.5 uvm_sequencer #(REQ,RSP) ... 218
15.6 uvm_push_sequencer #(REQ,RSP) ... 219

16. Policy classes ... 220

16.1 uvm_policy ... 220
16.2 uvm_printer ... 222
16.3 uvm_comparer ... 237
16.4 uvm_recorder ... 243
16.5 uvm_packer ... 251
16.6 uvm_copier ... 257

17. Register layer ... 260

17.1 Overview ... 260
17.2 Global declarations ... 260

18. Register model ... 264

18.1 uvm_reg_block ... 264
18.2 uvm_reg_map ... 276
18.3 uvm_reg_file ... 284
18.4 uvm_reg ... 286
18.5 uvm_reg_field ... 303
18.6 uvm_mem ... 314
18.7 uvm_reg_indirect_data ... 328
18.8 uvm_reg_fifo ... 329
18.9 uvm_vreg ... 332
18.10 uvm_vreg_field ... 342
18.11 uvm_reg_cbs ... 347
18.12 uvm_mem_mam ... 352

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.
Full document details: BS IEC 62530-2:2021

19. Register layer interaction with RTL design 361

19.1 Generic register operation descriptors 361
19.2 Classes for adapting between register and bus operations 365
19.3 uvm_reg_predictor 367
19.4 Register sequence classes 369
19.5 uvm_reg_backdoor 376
19.6 UVM HDL back-door access support routines 379

Annex A (informative) Bibliography 381

Annex B (normative) Macros and defines 382

Annex C (normative) Configuration and resource classes 407

Annex D (normative) Convenience classes, interface, and methods 422

Annex E (normative) Test sequences 431

Annex F (normative) Package scope functionality 443

Annex G (normative) Command line arguments 466

Annex H (informative) Participants 471

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

SystemVerilog –

Part 2: Universal Verification Methodology Language Reference Manual

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC document(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation.

   IEEE Standards documents are developed within IEEE Societies and Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of IEEE and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards. Use of IEEE Standards documents is wholly voluntary. *IEEE documents are made available for use subject to important notices and legal disclaimers (see* http://standards.ieee.org/ipr/disclaimers.html *for more information).*

   IEC collaborates closely with IEEE in accordance with conditions determined by agreement between the two organizations. This Dual Logo International Standard was jointly developed by the IEC and IEEE under the terms of that agreement.

2) The formal decisions of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees. The formal decisions of IEEE on technical matters, once consensus within IEEE Societies and Standards Coordinating Committees has been reached, is determined by a balanced ballot of materially interested parties who indicate interest in reviewing the proposed standard. Final approval of the IEEE standards document is given by the IEEE Standards Association (IEEE SA) Standards Board.

3) IEC/IEEE Publications have the form of recommendations for international use and are accepted by IEC National Committees/IEEE Societies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC/IEEE Publications is accurate, IEC or IEEE cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications (including IEC/IEEE Publications) transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC/IEEE Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

5) IEC and IEEE do not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC and IEEE are not responsible for any services carried out by independent certification bodies.

6) All users should ensure that they have the latest edition of this publication.

7) No liability shall attach to IEC or IEEE or their directors, employees, servants or agents including individual experts and members of technical committees and IEC National Committees, or volunteers of IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board, for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC/IEEE Publication or any other IEC or IEEE Publications.

8) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

9) Attention is drawn to the possibility that implementation of this IEC/IEEE Publication may require use of material 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. IEC or IEEE shall not be held responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility.

IEC 62530-2/IEEE Std 1800.2 was processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement. It is an International Standard.

AVA Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

The text of this International Standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1800.2 (2017)	91/1713/FDIS	91/1725/RVD

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

The language used for the development of this International Standard is English.

The IEC Technical Committee and IEEE Technical Committee have decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under webstore.iec.ch in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

IEEE Std 1800.2™-2017

IEEE Standard for Universal Verification Methodology Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 14 February 2017

IEEE-SA Standards Board

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Grateful acknowledgment is made for permission to use the following source material:

Accellera Systems Initiative—*The Universal Verification Methodology (UVM) pre-IEEE Class Reference*.

Abstract: The Universal Verification Methodology (UVM) that can improve interoperability, reduce the cost of using intellectual property (IP) for new projects or electronic design automation (EDA) tools, and make it easier to reuse verification components is provided. Overall, using this standard will lower verification costs and improve design quality throughout the industry. The primary audiences for this standard are the implementors of the UVM base class library, the implementors of tools supporting the UVM base class library, and the users of the UVM base class library.

Keywords: agent, blocking, callback, class, component, consumer, driver, event, export, factory, function, generator, IEEE 1800.2™, member, method, monitor, non-blocking, phase, port, register, resource, sequence, sequencer, transaction level modeling, verification methodology

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/disclaimers.html.

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE-SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

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.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Full document details: BS IEC 62530-2:2021

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive 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 comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

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

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.
Full document details: BS IEC 62530-2:2021

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://ieeexplore.ieee.org/ or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Full document details: BS IEC 62530-2:2021

Introduction

This introduction is not part of IEEE Std 1800.2-2017, IEEE Standard for Universal Verification Methodology Language Reference Manual.

Verification has evolved into a complex project that often spans internal and external teams, but the discontinuity associated with multiple, incompatible methodologies among those teams can limit productivity. The Universal Verification Methodology (UVM) Language Reference Manual (LRM) addresses verification complexity and interoperability within companies and throughout the electronics industry for both novice and advanced teams while also providing consistency. While UVM is revolutionary, being the first verification methodology to be standardized, it is also evolutionary, as it is built on the Open Verification Methodology (OVM), which combined the Advanced Verification Methodology (AVM) with the Universal Reuse Methodology (URM) and concepts from the ***e*** Reuse Methodology (eRM). Furthermore, UVM also infuses concepts and code from the Verification Methodology Manual (VMM), plus the collective experience and knowledge of the over 300 members of the Accellera UVM Working Group to help standardize verification methodology. Finally, the transaction level modeling (TLM) facilities in UVM are based on what was developed by Open SystemC Initiative (OSCI) for SystemC, though they are not an exact replication or re-implementation of the SystemC TLM library.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Full document details: BS IEC 62530-2:2021

IEEE Standard for Universal Verification Methodology Language Reference Manual

1. Overview

1.1 Scope

This standard establishes the Universal Verification Methodology (UVM), a set of application programming interfaces (APIs) that defines a base class library (BCL) definition used to develop modular, scalable, and reusable components for functional verification environments. The APIs and BCL are based on the IEEE standard for SystemVerilog, IEEE Std 1800™.[1]

1.2 Purpose

Verification components and environments are currently created in different forms, making interoperability among verification tools and/or geographically dispersed design environments both time consuming to develop and error prone. The results of the UVM standardization effort will improve interoperability and reduce the cost of repurchasing and rewriting *intellectual property* (IP) for each new project or electronic design automation (EDA) tool, as well as make it easier to reuse verification components. Overall, the UVM standardization effort will lower verification costs and improve design quality throughout the industry.

1.3 Conventions used

The conventions used throughout the document are as follows:

— UVM is case-sensitive.

— Any syntax examples shown in this standard are informative. They are intended to illustrate the usage of UVM constructs in a simple context and do not define the full syntax.

1.3.1 Visual cues (meta-syntax)

Bold shows required keywords and/or special characters, e.g., **uvm_component**.

Italics shows variables or definitions, e.g., *name* or *Globals*.

`Courier` shows SystemVerilog examples, external command names, directories and files, etc., e.g., an implementation needs to call `super.do_copy`.

[1]Information on references can be found in Clause 2.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

The asterisk (*) symbol, when combined with a prefix and/or postfix denoting a part of the construct, represents a series of construct names with exactly this prefix and/or postfix, e.g., `class uvm_*_port`.

1.3.2 Return values

a) Equivalent terms:
   1) "TRUE," "True," and "true" are equivalent to each other and used interchangeably throughout this document.
   2) "FALSE," "False," and "false" are equivalent to each other and used interchangeably throughout this document.
b) A bit value of `1` is treated as TRUE and `0` is treated as FALSE.
c) Conversely, TRUE refers to `1` and FALSE refers to `0` for return values.
d) Datatypes returned:
   1) For a bit or integer, `1` (or `1'b1`) or `0` (`1'b0`) is acceptable.
   2) For an enumerated type, TRUE or FALSE is acceptable.
e) For functions that return `TRUE`/`FALSE`, if only one returned value is defined (e.g., for `TRUE`), then the opposite return value shall be inferred (for all other possibilities).

1.3.3 Inheritance

Class declarations shown in this document may be of the form *class A* extends *B*. These declarations do not imply *class A* and *class B* are adjacent in the inheritance tree; implementations are free to have other classes between *A* and *B* in the inheritance tree, e.g.,

```
class X extends B;
      // body of class X
   endclass
class A extends X;
      // body of class A
   endclass
```

would comply.

The API and the semantics of the API from a base class shall be present in any derived classes, unless that API is overridden by an explicitly documented API within the derived class.

1.3.4 Operation order on equivalent data objects

The functionality described in this document typically operates on a set of data objects. An implementation and/or the underlying run-time engine may choose any operation order or sorting order for "equivalent data" objects within the specified semantics.

As a result of this policy, results returned and/or sequential behavior and/or produced output may differ between implementations and/or different underlying engines.

It is up to the user to establish an operation order if necessary.

1.3.5 uvm_pkg

All properties of UVM, including classes, global methods, and variables, are exported via the `uvm_pkg` package. They may be accessed via `import` or via the Scope Resolution operator (`::`).

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

UVM does not require any specific time unit precision for `uvm_pkg`.

All UVM methods that operate on values of type `time`, such as **uvm_printer::print_time** (see 16.2.3.11), are subject to the time scaling defined in IEEE Std 1800™.

1.3.6 Random stability

Any APIs that result in user code being executed are not guaranteed to be random stable. All other APIs are guaranteed to be random stable, unless otherwise specified.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

2. Normative references

The following referenced documents are indispensable for the application of this standard (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1800™, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.[2, 3]

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause. [4]

3.1 Definitions

agent: An abstract container used to emulate and verify device under test (DUT) devices; agents encapsulate a **driver**, **sequencer**, and **monitor**.

blocking: An interface where tasks block execution until they complete. *See also:* **non-blocking**.

component: A piece of verification intellectual property (VIP) that provides functionality and interfaces.

consumer: A verification component that receives **transaction**s from another **component**.

driver: A component responsible for executing or otherwise processing **transaction**s, usually interacting with the device under test (DUT) to do so.

environment: The container object that defines the **testbench** topology.

export: A transaction level modeling (TLM) interface that provides an implementation of methods used for communication. Used in Universal Verification Methodology (UVM) to connect to a port.

factory method: A classic software design pattern used to create generic code by deferring, until run time, the exact specification of the object to be created.

hook: A method that enables **user**s to customize certain behaviors of a **component**.

generator: A verification component that provides transactions to another **component**. Also referred to as a *producer.*

monitor: A passive entity that samples device under test (DUT) signals, but does not drive them.

non-blocking: A call that returns immediately. *See also:* **blocking**.

policy: A collection of settings used to apply an operation to a class.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

[3]The IEEE standards or products referred to in Clause 2 are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.

[4]*IEEE Standards Dictionary Online* is available at: http://ieeexplore.ieee.org/.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

port: A transaction level modeling (TLM) interface that defines the set of methods used for communication. Used in Universal Verification Methodology (UVM) to connect to an export.

proxy: A class functioning as an interface to another **component** or class.

request: A **transaction** that provides information to initiate the processing of a particular operation.

response: A **transaction** that provides information about the completion or status of a particular operation.

scoreboard: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually, refers to the entire dynamic response-checking structure.

sequence: A Universal Verification Methodology (UVM) object that procedurally defines a set of **transaction**s to be executed and/or controls the execution of other sequences.

sequencer: An advanced stimulus generator that executes **sequence**s that define the **transaction**s provided to the **driver** for execution.

singleton: A design pattern where the creation of the class only has one instance of that class.

test: Specific customization of an environment to exercise required functionality of the device under test (DUT).

testbench: The structural definition of a set of verification components used to verify a device under test (DUT). Also referred to as a *verification environment*.

transaction: A class instance that encapsulates information used to communicate between two or more **component**s.

user: Someone that uses the Universal Verification Methodology (UVM) base class library (BCL).

NOTE—In this standard, **user** uses the classes, functions, methods, or macros defined herein.[5]

3.2 Acronyms and abbreviations

API	application programming interface
BCL	base class library
DPI	direct programming interface
DUT	device under test
EDA	electronic design automation
FIFO	first-in, first-out
HDL	hardware description language
IP	intellectual property
RTL	register transfer level
TLM	transaction level modeling
UVM	Universal Verification Methodology
VIP	verification intellectual property

[5]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

4. UVM class reference

The UVM base class library provides the building blocks needed to quickly develop well constructed and reusable verification components and test environments in SystemVerilog. All UVM API should maintain random stability.

The UVM classes and utilities are divided into the following categories pertaining to their role or function. The subsequent clauses of this standard give a more detailed overview of each category—and the classes that comprise them.

Base—The basic building blocks for all environments are *components*, which do the actual work, *transactions*, which convey information between components, and *ports*, which provide the interfaces used to convey transactions. The UVM's core base classes provide these building blocks. See Clause 5.

Reporting—The reporting classes provide a facility for issuing reports (messages) with consistent formatting and configurable side effects, such as logging to a file or exiting simulation. Reports can also filter out messages based on their verbosity, unique ID, or severity. See Clause 6.

Recording—The recording classes provide a facility to record transactions into a database using a consistent API. Users can configure what gets sent to the back-end database, without knowing exactly how the connection to that database is established. See Clause 7.

Factory—As the name implies, the UVM factory is used to manufacture (create) UVM objects and components. A factory can be configured to produce an object of a given type on a global or instance basis. Factories allow dynamically configurable component hierarchies and object substitutions without having to modify their code or break encapsulation. See Clause 8.

Phasing—This category defines the phasing capability provided by UVM. See Clause 9.

Synchronization—These event and barrier synchronization classes can be used for process synchronization. See Clause 10.

Containers—These classes are type parameterized data structures that provide queue and pool services. The class-based queue and pool types allow for efficient sharing of the data structures compared with their SystemVerilog built-in counterparts. See Clause 11.

UVM TLM—The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each UVM TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use UVM TLM ports and exports to communicate are inherently more reusable, interoperable, and modular. See Clause 12.

Components—Components form the foundation of UVM. They encapsulate the behavior of drivers, scoreboards, and other objects in a testbench. The UVM base class library provides a set of predefined component types, all derived directly or indirectly from **uvm_component**. See Clause 13.

Sequences—Sequences encapsulate user-defined procedures that generate multiple **uvm_sequence_item**-based transactions (see 14.1). Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to a DUT. See Clause 14.

Sequencers—The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of **uvm_sequence_item**-based transactions (see 14.1) generated by one or more **uvm_sequence #(REQ,RSP)**-based sequences (see 14.3). See Clause 15.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

Policies—Each of UVM's policy classes performs a specific task for **uvm_object**-based objects: printing, comparing, recording, packing, and unpacking (see 5.3). They are implemented separately from **uvm_object** to allow for different ways to print, compare, etc., without modifying the object class being utilized; e.g., a user can simply apply a different printer or compare policy to change how an object is printed or compared. See Clause 16.

Register layer—The Register abstraction classes, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification. See Clause 17.

Macros—UVM provides several macros to help increase user productivity. See Annex B.

Configuration and resources—These classes provide a configuration database, which is used to store and retrieve both configuration time and run-time properties. See Annex C.

Package scope—This category defines a small list of types, variables, functions, and tasks defined in the `uvm_pkg` scope. These items are accessible from any scope that imports the `uvm_pkg`. See Annex F.

Command line processor—This a general interface to the command line arguments that were provided for the given simulation. See Annex G.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5. Base classes

5.1 Overview

The UVM base class library defines a set of base classes and utilities that facilitate the design of modular, scalable, and reusable verification environments. The basic building blocks for all environments are components and the transactions they use to communicate.

a) **uvm_object**—All components and transactions derive from **uvm_object** (see 5.3), which defines an interface of core class-based operations: create, copy, compare, print, sprint, record, etc. It also defines interfaces for instance identification (name, type name, unique id, etc.) and random seeding. All derivatives of **uvm_object** are factory enabled, unless otherwise specified.

b) **uvm_component**—The **uvm_component** class (see 13.1) is the base class for all UVM components. Components are quasi-static objects that exist throughout simulation. This allows them to establish structural hierarchy much like modules and program blocks. Components participate in a phased test flow during the course of simulation. Each phase—build, connect, run, etc.—is defined by a callback that is executed in precise order. Finally, the **uvm_component** also defines any configuration, reporting, transaction recording, and factory interfaces.

c) **uvm_transaction**—The **uvm_transaction** (see 5.4) is the root base class for UVM transactions, which, unlike **uvm_component**s (see 13.1), are transient in nature. It extends **uvm_object** (see 5.3) to include a timing and recording interface. Simple transactions can derive directly from **uvm_transaction**, while sequence-enabled transactions derive from **uvm_sequence_item** (see 14.1).

5.2 uvm_void

The **uvm_void** class is the abstract base class for all UVM classes. It is an abstract class with no data members or functions. It allows for creation of generic containers of objects, similar to a void pointer in the C programming language. User classes derived directly from **uvm_void** inherit none of the UVM functionality, but such classes may be placed in **uvm_void**-typed containers along with other UVM objects.

Class declaration

```
virtual class uvm_void
```

5.3 uvm_object

The **uvm_object** class is the abstract base class for all UVM data and hierarchical classes. Its primary role is to define a set of methods for such common operations as **create** (see 5.3.5.1), **copy** (see 5.3.8.1), **compare** (see 5.3.9.1), **print** (see 5.3.6.1), and **record** (see 5.3.7.1). Classes deriving from **uvm_object** need to implement the pure virtual methods such as **create** (see 5.3.5.1) and **get_type_name** (see 5.3.4.7).

5.3.1 Class declaration

```
virtual class uvm_object extends uvm_void
```

5.3.2 Common methods

new

```
function new (
    string name = ""
)
```

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

Creates a new **uvm_object** with the given instance *name*. If *name* is not supplied, the object is unnamed.

5.3.3 Seeding

5.3.3.1 get_uvm_seeding

```
static function bit get_uvm_seeding()
```

Helper method for retrieving the UVM seeding *enable* value via **uvm_coreservice_t::get_uvm_seeding** (see F.4.3).

5.3.3.2 set_uvm_seeding

```
static function void set_uvm_seeding (bit enable)
```

Helper method for setting the UVM seeding *enable* value via **uvm_coreservice_t::set_uvm_seeding** (see F.4.4).

5.3.3.3 reseed

```
function void reseed()
```

This method sets the seed of the object ensuring all objects have unique seeding values.

If the *get_uvm_seeding* method (see 5.3.3.1) returns `0`, then **reseed** does not perform any function.

5.3.4 Identification

5.3.4.1 set_name

```
virtual function void set_name (
    string name
)
```

Specifies the instance *name* of this object, overwriting any previously given *name* from **new** (see 5.3.2) or **set_name**.

5.3.4.2 get_name

```
virtual function string get_name()
```

Returns the name of the object, as provided by the *name* argument in the **new** constructor (see 5.3.2) or **set_name** method (see 5.3.4.1).

5.3.4.3 get_full_name

```
virtual function string get_full_name()
```

Returns the full hierarchical name of this object. The return value is the same as **get_name** (see 5.3.4.2), as **uvm_object**s do not inherently possess hierarchy.

Objects possessing hierarchy, such as **uvm_component**s (see 13.1), override the default implementation. Other objects might be associated with component hierarchy, but are not themselves components. For example, **uvm_sequence #(REQ,RSP)** (see 14.3) classes are typically associated with a **uvm_sequencer**

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

#(REQ,RSP) (see 15.5). In this case, it is useful to override **get_full_name** to return the sequencer's full name concatenated with the sequence's name. This provides the sequence a full context, which is useful when debugging.

5.3.4.4 get_inst_id

```
virtual function int get_inst_id()
```

Returns a distinct integer for each distinct UVM object. At the time an object is created, an object identifier is assigned to it, implicitly or explicitly, depending on which API creates the object. The ID of an object is guaranteed to be unique to that object for the object's lifetime. An implementation may reuse the ID of a previously garbage collected object.

5.3.4.5 get_type

```
static function uvm_object_wrapper get_type()
```

Returns the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation methods (see 8.3.1) take arguments of **uvm_object_wrapper** (see 8.3.2).

This method provides a common API for extensions of **uvm_object** to use when providing factory support. If an extension of **uvm_object** supports factory creation, that extension should implement a static **get_type** method that returns the appropriate **uvm_object_wrapper** (see 8.3.2).

This method is provided automatically when using the **`uvm_object_utils**, **`uvm_object_param_utils, `uvm_component_utils**, and **`uvm_component_param_utils** macros (and their ***_begin** and ***_end** variants). See B.2.1.2 and B.2.1.3.

5.3.4.6 get_object_type

```
virtual function uvm_object_wrapper get_object_type()
```

Returns the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation methods (see 8.3.1) take arguments of **uvm_object_wrapper** (see 8.3.2). This method, if implemented, can be used as convenient means of supplying those arguments. This method is the same as the static **get_type** method (see 5.3.4.5), but it uses an already allocated object to determine the type-proxy to access (instead of using the static object).

The default implementation of this method does a factory lookup of the proxy using the return value from **get_type_name** (see 5.3.4.7). If the type returned by **get_type_name** is not registered with the factory, then a *null* handle is returned.

5.3.4.7 get_type_name

```
virtual function string get_type_name()
```

This function returns the type name of the object, which is typically the type identifier enclosed in quotes. It is used for various debugging functions in the UVM base class library and it is used by the factory for creating objects.

This function shall be defined in every derived class.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.3.5 Creation

5.3.5.1 create

```
virtual function uvm_object create (
    string name = ""
)
```

The **create** method allocates a new object of the same type as this object and returns it via a base **uvm_object** handle. Every class deriving from **uvm_object**, directly or indirectly, shall implement the **create** method.

The default implementation of the **create** method returns *null*. Every class deriving from **uvm_object** shall implement the **create** method to return the newly allocated object of the same type as the derived class.

5.3.5.2 clone

```
virtual function uvm_object clone()
```

The **clone** method creates and returns an exact copy of this object.

5.3.6 Printing

The printing methods, **print** (see 5.3.6.1) and **sprint** (see 5.3.6.2), initiate a new print operation on this object. To ensure correct *printing* operation, and to ensure a consistent output format, the user shall use a **uvm_printer** policy class (see 16.2). That is, instead of using `$display` or string concatenations directly, the **do_execute_op** (see 5.3.13.1) and **do_print** (see 5.3.6.3) implementations shall use the policy's APIs to print fields. See 16.2 for more information on printer output formatting.

5.3.6.1 print

```
function void print (
    uvm_printer printer = null
)
```

Prints this object to the target specified by the policy.

The *printer* argument provides the policy class to be used for this operation. If no *printer* is provided (or the value provided is *null*), the method shall use the default printer policy, as returned by **get_default_printer** (see F.4.1.4.13).

The following steps occur in order:

a) If the policy's active object depth (see 16.1.3.4) is `0`, then **flush** (see 16.2.4.2) is called on the *printer policy*.

b) **print_object** (see 16.2.3.1) is called on the *printer policy*, the *name* sent to **print_object** is determined using **get_root_enabled** (see 16.2.5.8).

c) The value returned by the *printer policy*'s **emit** method (see 16.2.4.1) shall be directed to the *printer*'s current File (see 16.2.5.11).

5.3.6.2 sprint

```
function string sprint (
    uvm_printer printer = null
)
```

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

The **sprint** method works just like the **print** method (see 5.3.6.1), except the output of the *printer policy*'s **emit** method (see 16.2.4.1) is returned in a string rather than displayed.

5.3.6.3 do_print

```
virtual function void do_print (
    uvm_printer printer
)
```

The **do_print** method is a user-definable hook that allows users customization over what is printed beyond the information provided by the field macros (see B.2.2) or **do_execute_op** method (see 5.3.13.1).

5.3.6.4 convert2string

```
virtual function string convert2string()
```

This virtual function has a default implementation that returns an *empty string* (`""`), but may be extended in derived classes to provide object information in the form of a string. The format of the string is user-defined.

5.3.7 Recording

To ensure correct *recording* operation, the user shall use a **uvm_recorder** policy class (see 16.4). That is, instead of using implementation-specific API directly, the **do_execute_op** (see 5.3.13.1) and **do_record** (see 5.3.7.2) implementations shall use the policy's APIs to record fields. See Clause 7 for more information on the recording classes.

5.3.7.1 record

```
function void record (
    uvm_recorder recorder = null
)
```

The **record** method initiates a new *record* operation on this object.

The *recorder* argument provides the policy class to be used for this operation. If no *recorder* is provided (or the value provided is *null*), the call is silently ignored. Otherwise, the object shall pass itself to the **record_object** method (see 16.4.6.4) of the *recorder*.

5.3.7.2 do_record

```
virtual function void do_record (
    uvm_recorder recorder
)
```

The **do_record** method is a user-definable hook that allows users customization over what is recorded beyond the information provided by the field macros (see B.2.2) or **do_execute_op** method (see 5.3.13.1).

5.3.8 Copying

5.3.8.1 copy

```
function void copy (
    uvm_object rhs,
    uvm_copier copier = null
)
```

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

The **copy** method copies the field values from *rhs* into this object.

The *copier* argument provides the policy class to be used for this operation. If no *copier* is provided (or the value provided is *null*), the method shall use the default copier policy, as returned by **get_default_copier** (see F.4.1.4.19).

The following steps occur in order.

a) If the policy's active object depth (see 16.1.3.4) is `0`, then **flush** (see 16.2.4.2) is called on the copier policy.

b) **copy_object** (see 16.6.4.1) is called on the *copier* policy, with this object as *lhs* and rhs as *rhs*.

5.3.8.2 do_copy

```
virtual function void do_copy (
    uvm_object rhs
)
```

The **do_copy** method is the user-definable hook called by the **copy** method (see 5.3.8.1). A derived class can override this method to include its fields in a **copy** operation.

An implementation in a derived class should call `super.do_copy` and `$cast` the *rhs* argument to the derived type before copying.

5.3.9 Comparing

To ensure correct *comparing* operation, theuser shall use a **uvm_comparer** policy class (see 16.3). That is, instead of using implementation-specific API directly, the **do_execute_op** (see 5.3.13.1) and **do_compare** (see 5.3.9.2) implementations shall use the policy's APIs to compare fields.

5.3.9.1 compare

```
function bit compare (
    uvm_object rhs,
    uvm_comparer comparer = null
)
```

Compares the current object to *rhs*.

The *comparer* argument provides the policy class to be used for this operation. If no *comparer* is provided (or the value provided is *null*), the method shall use the default comparer policy, as returned by **get_default_comparer** (see F.4.1.4.16).

The following steps occur in order:

a) If the policy's active object depth (see 16.1.3.4) is `0`, then **flush** (see 16.2.4.2) is called on the *comparer policy*.

b) **compare_object** (see 16.3.3.4) is called on the *comparer policy*, with *lhs* set to this object and *name* set to the return value of this object's **get_name** method (see 5.3.4.2).

c) The value returned by **compare_object** (see 16.3.3.4) shall be returned by **compare**.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5.3.9.2 do_compare

```
virtual function bit do_compare (
   uvm_object rhs,
   uvm_comparer comparer
)
```

The **do_compare** method is a user-definable hook that allows users customization over what is recorded beyond the information provided by the field macros (see B.2.2) or **do_execute_op** method (see 5.3.13.1). A derived class can override this method to include its fields in a *compare* operation. It shall return `1` if the comparison succeeds, `0` otherwise.

A derived class implementation should call `super.do_compare` to ensure its base class' properties, if any, are included in the comparison. Also, the *rhs* argument is provided as a generic **uvm_object**. Thus, the derived class implementation needs to `$cast` *rhs* to the type of this object before comparing.

5.3.10 Packing

5.3.10.1 pack, pack_bytes, pack_ints, and pack_longints

```
function int pack (
       ref bit bitstream[],
   input uvm_packer packer = null
)

function int pack_bytes (
   ref byte unsigned bytestream[],
       input uvm_packer packer = null
)

function int pack_ints (
   ref int unsigned intstream[],
       input uvm_packer packer = null
)

function int pack_longints (
   ref longint unsigned longintstream[],
   input uvm_packer packer = null
)
```

The **pack_*** methods bitwise-concatenate this object's properties into an array of bits, bytes, ints, or longints. The methods are not virtual and shall not be overloaded. To include additional fields in a **pack** operation, derived classes can override the **do_pack** method (see 5.3.10.2).

The optional *packer* argument specifies the packing policy. If this is not provided, the default as returned by **get_default_packer** (see F.4.1.4.15) is used. See 16.5 for more information.

If the policy's active object depth (see 16.1.3.4) is `0`, then **flush** (see 16.5.2.2) is called on the packer policy prior to packing any fields. After processing the object's fields, the packer's state is copied to the *stream* array using the *state retrieval* method (see 16.5.3.2) of the same *stream* type.

The return value is the number of bits packed into the packer, as determined via **uvm_packer::get_packed_size** (see 16.5.3.3).

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.3.10.2 do_pack

```
virtual function void do_pack (
    uvm_packer packer
)
```

The **do_pack** method is the user-definable hook called by the **pack** methods (see 5.3.10.1). A derived class can override this method to include its fields in a **pack** operation.

The *packer* argument is the policy object for packing, which is responsible for generating the final array of packed data from the fields provided. It shall be an error to pass a value of `null` to the *packer* argument of **do_pack**, an implementation of **uvm_object::do_pack** shall generate an error message if a `null` value is detected.

While the unpacking order needs to match the packing order, the packing order does not need to match declaration order itself.

5.3.11 Unpacking

5.3.11.1 unpack, unpack_bytes, unpack_ints, and unpack_longints

```
function int unpack (
        ref bit bitstream[],
    input uvm_packer packer = null
)

function int unpack_bytes (
    ref byte unsigned bytestream[],
        input uvm_packer packer = null
)

function int unpack_ints (
    ref int unsigned intstream[],
        input uvm_packer packer = null
)

function int unpack_longints (
    ref longint unsigned longintstream[],
    input uvm_packer packer = null
)
```

The **unpack_*** methods extract this object's property values from an array of bits, bytes, ints, or longints. The object shall unpack fields in the same order in which they were originally packed.

The **unpack_*** methods are fixed (non-virtual) entry points that are directly callable by the user. To include additional fields in the **unpack** operation, derived classes can override the **do_unpack** method (see 5.3.11.2).

The optional *packer* argument specifies the unpacking policy. If this is not provided, the default as returned by **get_default_packer** (see F.4.1.4.15) is used. See 16.5 for more information.

Prior to unpacking any fields, the object shall set the internal state of the packer using the *state assignment* method (see 16.5.3.1) of the same *stream* type.

The return value is the actual number of bits unpacked from the given array.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5.3.11.2 do_unpack

```
virtual function void do_unpack (
    uvm_packer packer
)
```

The **do_unpack** method is the user-definable hook called by the **unpack** method (see 5.3.11.1). A derived class can override this method to include its fields in an **unpack** operation.

The *packer* argument is the policy object for unpacking, which is responsible for generating field values from an array of packed data. See 16.5 for more information.

It shall be an error to pass a value of `null` to the *packer* argument of **do_unpack**, an implementation of **uvm_object::do_unpack** shall generate an error message if a `null` value is detected.

NOTE—As the underlying storage format of the **uvm_packer** (see 16.5) is unspecified, it is unsafe for users to unpack fields using different types than the fields with which they were packed or to unpack fields in a different order than the fields in which they were packed.

5.3.12 Configuration

set_local

```
virtual function void set_local (
    uvm_resource_base rsrc
)
```

This method provides write access to member properties by using a UVM resource (see C.2). The return value of **get_name** (see 5.3.4.2) for *rsrc* is used to determine the name of the property being accessed. The object designer can choose which, if any, properties are accessible and override this method.

5.3.13 Field operations

In addition to explicit *printing* (see 5.3.6), *recording* (see 5.3.7), *copying* (see 5.3.8), *comparing* (see 5.3.9), *packing* (see 5.3.10), *unpacking* (see 5.3.11), and *configuration* (see 5.3.12) support, **uvm_object** provides a *field operation* mechanism that allows for a centralized definition of all operations that are supported for an object's fields.

5.3.13.1 do_execute_op

```
virtual function void do_execute_op(
    uvm_field_op op
)
```

The **do_execute_op** method is the user-definable hook called by the policy class. A derived class may override this method to include its fields in the execution of the operation. The field macros (see B.2.2) provide a default implementation of the **do_execute_op** method that is available to the user.

5.3.13.2 uvm_field_op

uvm_field_op is the UVM class for describing all operations supported by the **do_execute_op** function (see 5.3.13.1).

5.3.13.2.1 Class declaration

```
class uvm_field_op extends uvm_object
```

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.3.13.2.2 Methods

uvm_field_op has the following methods (see 5.3.13.2.3 to 5.3.13.2.11).

5.3.13.2.3 new

```
protected function new(
    string name=""
)
```

Creates a new object of type **uvm_field_op** with the given instance *name*. If *name* is not supplied, the object is unnamed.

5.3.13.2.4 set

```
virtual function void set(
    uvm_field_flag_t op_type,
    uvm_policy policy = null,
    uvm_object rhs = null
)
```

Sets the operation *op_type*, *policy*, and *rhs* values.

The **set** method takes three arguments as follows:

a) *op_type*—The operation type, as described using **uvm_field_flag_t** (see F.2.1.2). A **uvm_field_op** can only represent a single operation at a time; it shall be an error if the reserved bits (see F.2.1.1) of the *op_type* argument match more than one operation type.

b) *policy*—The policy class to be used for this operation. Operations that do not require a policy object may set the policy to *null*. The default value is *null*.

c) *rhs*—The right hand side value to be used for this operation. Operations that do not require a right hand side object may set the *rhs* to *null*. The default value is *null*.

An error shall be generated if **set** is called twice without a **flush** call (see 5.3.13.2.11) between.

5.3.13.2.5 get_op_name

```
virtual function string get_op_name()
```

Returns the name associated with the operation, based on the assigned operation *type* (see Table 1).

get_op_name shall generate an error message if **set** (see 5.3.13.2.4) has never been called on this **uvm_field_op** or if **flush** (see 5.3.13.2.11) has been called more recently than **set** on this **uvm_field_op**.

The return value for types that are not listed in Table 1 is undefined.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Table 1—Type names returned

Type of operation	Name returned
UVM_PRINT	`print`
UVM_RECORD	`record`
UVM_PACK	`pack`
UVM_UNPACK	`unpack`
UVM_COPY	`copy`
UVM_COMPARE	`compare`
UVM_SET	`set`

5.3.13.2.6 get_op_type

```
virtual function uvm_field_flag_t get_op_type()
```

Returns the type of operation.

get_op_type shall generate an error message if **set** (see 5.3.13.2.4) has never been called on this **uvm_field_op** or if **flush** (see 5.3.13.2.11) has been called more recently than **set** on this **uvm_field_op**.

5.3.13.2.7 get_policy

```
virtual function uvm_policy get_policy()
```

Returns the policy object to be used when executing the operation.

get_policy shall generate an error message if **set** (see 5.3.13.2.4) has never been called on this **uvm_field_op** or if **flush** (see 5.3.13.2.11) has been called more recently than **set** on this **uvm_field_op**.

5.3.13.2.8 get_rhs

```
virtual function uvm_object get_rhs()
```

Returns the right-hand side object to be used when executing the operation.

get_rhs shall generate an error message if **set** (see 5.3.13.2.4) has never been called on this **uvm_field_op** or if **flush** (see 5.3.13.2.11) has been called more recently than **set** on this **uvm_field_op**.

5.3.13.2.9 user_hook_enabled

```
function bit user_hook_enabled()
```

Returns the current value of the user hook enabled bit. The value defaults to `1` and can only be set to `0` via a call to **disable_user_hook** (see 5.3.13.2.10).

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

The user hook enabled bit indicates whether the policy class should call the **do_*** method associated with its operation after calling the **do_execute_op** method (see 5.3.13.1). For example, a **uvm_printer** (see 16.2) will not call **do_print** (see 5.3.6.3) if **user_hook_enabled** returns `0` after the printer has called **do_execute_op**.

5.3.13.2.10 disable_user_hook

```
function void disable_user_hook()
```

Disables the call to the user hook by setting the return value of **user_hook_enabled** (see 5.3.13.2.9) to `0`.

5.3.13.2.11 flush

```
virtual function void flush()
```

Resets the **uvm_field_op**, allowing it to be reused. Future calls to **uvm_field_op::get_*** (see 5.3.13.2.5 to 5.3.13.2.8) shall generate errors unless **set** (see 5.3.13.2.4) is called again.

5.3.14 Active policy

The active policy methods are used to track which policy object (see 16.1) is presently operating on an object.

5.3.14.1 push_active_policy

```
virtual function void push_active_policy(
   uvm_policy policy
)
```

Pushes *policy* on to the internal policy stack for this object, making it the current active policy, as retrieved by **get_active_policy** (see 5.3.14.3). An implementation shall generate an error message if *policy* is *null*, and the request will be ignored.

5.3.14.2 pop_active_policy

```
virtual function uvm_policy pop_active_policy()
```

Pops the current active policy off of the internal policy stack for this object. If the internal policy stack for this object is empty when **pop_active_policy** is called, then *null* is returned.

5.3.14.3 get_active_policy

```
virtual function uvm_policy get_active_policy()
```

Returns the head of the internal policy stack for this object. If the internal policy stack for this object is empty, *null* is returned.

5.4 uvm_transaction

The **uvm_transaction** class is the root base class for UVM transactions. Inheriting all the methods of **uvm_object** (see 5.3), **uvm_transaction** adds a timing and recording interface.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5.4.1 Class declaration

```
virtual class uvm_transaction extends uvm_object
```

5.4.2 Methods

5.4.2.1 new

```
function new (
    string name = ""
    uvm_component initiator = null
)
```

Initializes a new transaction object. The *name* is the instance name of the transaction. If not supplied, then the object is unnamed. The *initiator* is described in **set_initiator** (see 5.4.2.15).

5.4.2.2 accept_tr

```
function void accept_tr (
    time accept_time = 0
)
```

Calling **accept_tr** indicates the transaction item has been received by a consumer component.

"Accept" refers to the consumer receiving an item, whereas "begin" (see 5.4.2.4) refers to the consumer acting on an item. Those may or may not be coincident.

This function shall perform the following actions:

— The transaction's internal accept time is set to the current simulation time, or to *accept_time* if provided and non-zero. The *accept_time* may be any time, past or future. The default value of *accept_time* shall be `0`.

— `accept_tr(0)` is treated as if it is `accept_tr($time)`.

— The event at key *accept* in the transaction's event pool (see 5.4.2.14) is triggered. Any processes waiting on the this event resume in the next delta cycle.

— The **do_accept_tr** method (see 5.4.2.3) is called to allow for any post-accept action in derived classes.

5.4.2.3 do_accept_tr

```
virtual protected function void do_accept_tr()
```

This user-definable callback is called by **accept_tr** (see 5.4.2.2) just before the accept event is triggered. Implementations should call `super.do_accept_tr` to ensure correct operation.

5.4.2.4 begin_tr

```
function int begin_tr (
    time begin_time = 0
)
```

This function indicates the transaction has been started and is not the child of another transaction. Generally, a consumer component begins execution of a transactions it receives.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

See 5.4.2.2 for more information on how the begin-time may differ from when the transaction item was received.

This function shall perform the following actions:

— The transaction's begin time (see 5.4.2.17) is set to the current simulation time, or to *begin_time* if provided and non-zero. The *begin_time* may be any time, past or future, but shall not be less than the accept time. The default value of *begin_time* shall be `0`.
— `begin_tr(0)` means the current time and is valid as well.
— If recording is enabled (see 5.4.2.12), a new database transaction is started with the same *begin_time* via a call to **uvm_tr_stream::open_recorder** (see 7.2.5.1). The record method inherited from **uvm_object** (see 5.3) is then called, which records the current property values to this new transaction.
— The **do_begin_tr** method (see 5.4.2.6) is called to allow for any post-begin action in derived classes.
— The event at key `begin` in the transaction's event pool (see 5.4.2.14) is triggered. Any processes waiting on this event resume in the next delta cycle.

The return value is a transaction handle that is valid (non-zero) only if recording is enabled. The meaning of the handle is implementation specific.

5.4.2.5 begin_child_tr

```
function int begin_child_tr (
    time begin_time = 0,
    int parent_handle = 0
)
```

This function indicates the transaction has been started as a child of a parent transaction given by *parent_handle*. Generally, a consumer component calls this method via **uvm_component::begin_child_tr** (see 13.1.6.4) to indicate the actual start of execution of this transaction.

The parent handle is obtained by a previous call to **begin_tr** (see 5.4.2.4) or **begin_child_tr**. If the *parent_handle* is invalid (`=0`), then this function behaves the same as **begin_tr** (see 5.4.2.4).

This function shall perform the following actions:

— The transaction's begin time (see 5.4.2.17) is set to the current simulation time, or to *begin_time* if provided and non-zero. The *begin_time* may be any time, past or future, but shall not be less than the accept time.
— If recording is enabled (see 5.4.2.12), a new database transaction is started with the same *begin_time* via a call to **uvm_tr_stream::open_recorder** (see 7.2.5.1). The record method inherited from **uvm_object** (see 5.3) is then called, which records the current property values to this new transaction. Finally, the newly started transaction is linked to the parent transaction given by *parent_handle*, using a **uvm_parent_child_link** (see 7.3.2). The default value of *parent_handle* shall be `0`.
— The **do_begin_tr** method (see 5.4.2.6) is called to allow for any post-begin action in derived classes.
— The event at key `begin` in the transaction's event pool (see 5.4.2.14) is triggered. Any processes waiting on this event resume in the next delta cycle.

The return value is a transaction handle that is valid (non-zero) only if recording is enabled. The meaning of the handle is implementation specific. This transaction handle can be used with

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.
Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

uvm_recorder::get_recorder_from_handle (see 16.4.5.2) to retrieve the **uvm_recorder** that was opened for this transaction.

5.4.2.6 do_begin_tr

```
virtual protected function void do_begin_tr()
```

This user-definable callback is called by **begin_tr** (see 5.4.2.4) and **begin_child_tr** (see 5.4.2.5) just before the event at key `begin` in the transaction's event pool (see 5.4.2.14) is triggered. Implementations should call `super.do_begin_tr` to ensure correct operation.

5.4.2.7 end_tr

```
function void end_tr (
   time end_time = 0,
   bit free_handle = 1
)
```

This function indicates the transaction execution has ended. Generally, a consumer component ends execution of the transactions it receives.

begin_tr (see 5.4.2.4) or **begin_child_tr** (see 5.4.2.5) shall have been previously called for this call to be successful.

This function shall perform the following actions:

— The transaction's internal end time is set to the current simulation time, or to *end_time* if provided and non-zero. The *end_time* may be any non-negative time. The default value of *end_time* shall be `0`.

— If recording is enabled and a database transaction is currently active, the record method inherited from **uvm_object** (see 5.3) is called, which records the final property values. The **uvm_recorder** associated with this transaction is closed via a call to **uvm_recorder::close** (see 16.4.4.2). If `free_handle = 1`, the recorder is released and can no longer be linked (if supported by an implementation). The default value of *free_handle* shall be `1`.

— The **do_end_tr** method is called to allow for any post-end action in derived classes.

— The event at key `end` in the transaction's event pool (see 5.4.2.14) is triggered. Any processes waiting on this event resume in the next delta cycle.

5.4.2.8 do_end_tr

```
virtual protected function void do_end_tr()
```

This user-definable callback is called by **end_tr** (see 5.4.2.7) just before the event at key `end` in the transaction's event pool (see 5.4.2.14) is triggered. Implementations should call `super.do_end_tr` to ensure correct operation.

5.4.2.9 get_tr_handle

```
function int get_tr_handle()
```

Returns the handle associated with the transaction, as specified by a previous call to **begin_child_tr** (see 5.4.2.5) or **begin_tr** (see 5.4.2.4) with transaction recording enabled. If **begin_child_tr** or **begin_tr** have not been called, or if **is_active** (see 5.4.2.13) returns `0`, then **get_tr_handle** returns `0`.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.4.2.10 enable_recording

```
function void enable_recording (
    uvm_tr_stream stream
)
```

Turns on recording to the *stream* specified. Only one stream is tracked within the transaction, so further calls to **enable_recording** overwrite the internally stored value.

If transaction recording is on (the default), then a call to **record** (see 5.3.7.1) is made when the transaction is ended.

An error shall be generated if **enable_recording** is called after **accept_tr** (see 5.4.2.2), **begin_tr** (see 5.4.2.4), or **begin_child_tr** (see 5.4.2.5) but before **end_tr** (see 5.4.2.7).

5.4.2.11 disable_recording

```
function void disable_recording()
```

Turns off recording that has been enabled by a call to **enable_recording** (see 5.4.2.10). This is effectively identical to passing *null* to **enable_recording**.

An error shall be generated if **disable_recording** is called after **accept_tr** (see 5.4.2.2), **begin_tr** (see 5.4.2.4), or **begin_child_tr** (see 5.4.2.5) but before **end_tr** (see 5.4.2.7).

5.4.2.12 is_recording_enabled

```
function bit is_recording_enabled()
```

Returns `1` if recording is currently on, `0` otherwise.

5.4.2.13 is_active

```
function bit is_active()
```

Returns `1` if the transaction has been started, but has not been ended. Returns `0` if the transaction has not been started or has ended.

5.4.2.14 get_event_pool

```
function uvm_event_pool get_event_pool()
```

Returns the event pool associated with the transaction (see 10.4.1).

5.4.2.15 set_initiator

```
function void set_initiator (
    uvm_component initiator
)
```

Specifies *initiator* as the initiator of the transaction. The meaning of initiator is up to the user, e.g., the initiator can be the component that produces the transaction. An implementation shall include the initiator when printing (see 5.3.6) or recording (see 5.3.7).

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5.4.2.16 get_initiator

```
function uvm_component get_initiator()
```

Returns the component that produced or started the transaction, as specified by a previous call to **set_initiator** (see 5.4.2.15).

5.4.2.17 get_accept_time, get_begin_time, and get_end_time

```
function time get_accept_time()

function time get_begin_time()

function time get_end_time()
```

Returns the time at which this transaction was accepted, begun, or ended, as by a previous call to **accept_tr** (see 5.4.2.2), **begin_tr** (see 5.4.2.4), **begin_child_tr** (see 5.4.2.5), or **end_tr** (see 5.4.2.7).

5.4.2.18 set_transaction_id

```
function void set_transaction_id(
    int id
)
```

Specifies this transaction's numeric identifier to *id*. If not specified via this method, the transaction ID defaults to `-1`.

5.4.2.19 get_transaction_id

```
function int get_transaction_id()
```

Returns this transaction's numeric identifier, which is `-1` if not specified explicitly by **set_transaction_id** (see 5.4.2.18). `-1` is not allowed as an *id*.

5.5 uvm_port_base #(IF)

Transaction-level communication between components is handled via its ports, exports, imps, and sockets, all of which derive from this class.

The **uvm_port_base** extends **IF**, which is the type of the interface implemented by derived port, export, implementation, or socket. **IF** is also a `type` parameter to **uvm_port_base**.

> **IF**—The interface type implemented by the subtype to this base port.

uvm_port_base possesses the properties of components in that they have a hierarchical instance path and parent.

5.5.1 Class declaration

```
virtual class uvm_port_base #(
    type IF = uvm_void
) extends IF
```

The default value of *IF* shall be `uvm_void`.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.5.2 Methods

5.5.2.1 new

```
function new (
    string name,
    uvm_component parent,
    uvm_port_type_e port_type,
    int min_size = 0,
    int max_size = 1
)
```

name and *parent* are the **uvm_component** (see 13.1) constructor arguments.

port_type can be one of **UVM_PORT**, **UVM_EXPORT**, or **UVM_IMPLEMENTATION** (see F.2.3).

min_size and *max_size* specify the minimum and maximum number of implementation (`imp`) ports to be connected to this port base by the end of elaboration. Setting *max_size* to `-1` specifies no maximum, i.e., an unlimited number of connections are allowed. The default value of *min_size* shall be `0`. The default value of *max_size* shall be `1`.

5.5.2.2 get_name

```
function string get_name()
```

Returns the leaf name of this port.

5.5.2.3 get_full_name

```
virtual function string get_full_name()
```

Returns the full hierarchical name of this port.

5.5.2.4 get_parent

```
virtual function uvm_component get_parent())
```

Returns the handle to this port's parent, or *null* if it has no parent.

5.5.2.5 get_type_name

```
virtual function string get_type_name()
```

Returns the type name to this port. Derived port classes can implement this method to return the concrete type. Otherwise, only a generic "`uvm_port`", "`uvm_export`", or "`uvm_implementation`" is returned.

5.5.2.6 min_size

```
function int min_size()
```

Returns the minimum number of implementation ports to be connected to this port prior to **resolve_bindings** being called (see 5.5.2.15).

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

5.5.2.7 max_size

```
function int max_size()
```

Returns the maximum number of implementation ports to be connected to this port prior to **resolve_bindings** being called (see 5.5.2.15).

5.5.2.8 is_unbounded

```
function bit is_unbounded()
```

Returns `1` if this port has no maximum on the number of implementation ports this port can connect. A port is unbounded when the *max_size* argument (see 5.5.2.1) in the constructor is specified as `-1`.

5.5.2.9 get_connected_to

```
pure virtual function void get_connected_to(
    ref uvm_port_base#(IF) list [string]
)
```

For a port or export type, this function intends to fill a list with all of the ports, exports and implementations that this port is connected to.

5.5.2.10 get_provided_to

```
pure virtual function void get_provided_to(
    ref uvm_port_base#(IF) list [string]
)
```

For an implementation or export type, this function intends to fill a list with all of the ports, exports, and implementations to which this port has provided its implementation.

5.5.2.11 is_port, is_export, and is_imp

```
function bit is_port()

function bit is_export()

function bit is_imp()
```

Returns `1` if this port is of the type given by the method name, `0` otherwise.

5.5.2.12 size

```
function int size()
```

Returns the number of implementation ports connected to this port. The value is not valid before the `end_of_elaboration` phase, as port connections have not yet been resolved.

5.5.2.13 set_default_index

```
function void set_default_index (
    int index
)
```

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

Specifies the default implementation port to use when calling an interface method. This method should only be called on `UVM_EXPORT` types. The value shall not be specified before the `end_of_elaboration` phase. Use **size** (see 5.5.2.12) to retrieve the valid range for *index*. If **set_default_index** is not called, the default shall be `0`.

5.5.2.14 connect

```
virtual function void connect (
    uvm_port_base #(IF) provider
)
```

Connects this port to the given provider port. The ports shall be compatible in the following ways:

a) The types of their *IF* parameters shall be identical.

b) The provider's interface type (blocking, non-blocking, analysis, etc.) shall be compatible, e.g., an `uvm_blocking_put_port #(T)` is compatible with an `uvm_put_export #(T)` and `uvm_blocking_put_imp #(T)` because the `export` and `imp` provide the interface required by the `uvm_blocking_put_port`.

c) Ports of type **UVM_EXPORT** (see F.2.3) shall only connect to other exports or imps.

d) Ports of type **UVM_IMPLEMENTATION** (see F.2.3) shall not be connected, as they are bound to the component that implements the interface at the time of construction.

In addition to type-compatibility checks, the relationship between this port and the provider port is also checked if the port's **check_connection_relationships** configuration has been specified. By default, the parent/child relationship of any port being connected to this port is not checked.

This functionality can be disabled using the configuration and resources classes (see Annex C). The port shall check for a field named `check_connection_relationships` of the resource type `uvm_resource#(uvm_bitstream_t)` with a scope matching the port's full name (see 5.5.2.3). A value of `0` disables the check, any other value enables the check.

Relationships, when enabled, are checked are as follows:

— If this port is an **UVM_PORT** type (see F.2.3), the *provider* shall be a parent port, or a sibling export or implementation port.

— If this port is an **UVM_EXPORT** type (see F.2.3), the *provider* shall be a child export or implementation port.

If any relationship check is violated, a warning shall be issued.

5.5.2.15 resolve_bindings

```
virtual function void resolve_bindings()
```

This method is automatically called just before entering the `end_of_elaboration` phase. It recurses through each port's fanout to determine all the `imp` destinations. It then checks against the required min and max connections. After resolution, **size** (see 5.5.2.12) returns a valid value and **get_if** (see 5.5.2.16) can be used to access a particular `imp`.

5.5.2.16 get_if

```
function uvm_port_base #(
       IF
) get_if(int index=0)
```

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

Returns an implementation (`imp`) port at the given index from the array of `imps` to which this port is connected. Use **size** (see 5.5.2.12) to retrieve the valid range for *index*. This method should only be called at the `end_of_elaboration` phase or after, as port connections are not resolved before then. The default value of *index* shall be `0`.

5.6 uvm_time

Canonical time type that can be used in different time scales.

This time type is used to represent time values in a canonical form that can bridge different time scales and time precisions.

5.6.1 Class declaration

```
class uvm_time
```

5.6.2 Common methods

5.6.2.1 new

```
function new(
   string name = "uvm_time",
   real res = 0
)
```

Initializes a new canonical time object.

The canonical time value of the object is initialized to `0`. If a resolution is not specified, the default resolution, as specified by **set_time_resolution** (see 5.6.2.2), is used.

5.6.2.2 set_default_time_resolution

```
static function void set_default_time_resolution(
   real res
)
```

res specifies the default canonical time resolution; this shall be a power of `10`.

By default, the default resolution is `1.0e-12` (`ps`).

5.6.2.3 get_name

```
function string get_name()
```

Returns the name of this instance.

5.6.2.4 reset

```
function void reset()
```

Resets the canonical time value to `0`.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

5.6.2.5 get_realtime

```
function real get_realtime(
   time scaled,
   real secs = 1.0e-9
)
```

Returns the current canonical time value, scaled for the caller's time scale.

scaled shall be a time literal value that corresponds to the number of seconds specified in *secs* (`1ns` by default); it has to be a time literal value that is greater or equal to the current time scale.

5.6.2.6 incr

```
function void incr(
   real t,
   time scaled,
   real secs = 1.0e-9
)
```

Increments the current canonical time value by the specified number of scaled time units.

t is a time value expressed in the scale and precision of the caller. *scaled* shall be a time literal value that corresponds to the number of seconds specified in *secs* (`1ns` by default); it has to be a time literal value that is greater or equal to the current time scale.

5.6.2.7 decr

```
function void decr(
   real t,
   time scaled,
   real secs = 1.0e-9
)
```

Decrements the current canonical time value by the specified number of scaled time units.

t is a time value expressed in the scale and precision of the caller. *scaled* shall be a time literal value that corresponds to the number of seconds specified in *secs* (`1ns` by default); it has to be a time literal value that is greater or equal to the current time scale.

5.6.2.8 get_abstime

```
function real get_abstime(
   real secs
)
```

Returns the current canonical time value, in the number of specified time units, regardless of the current time scale of the caller.

secs is the number of seconds in the desired time unit, e.g., `1e-9` for nanoseconds.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.
Full document details: BS IEC 62530-2:2021

5.6.2.9 set_abstime

```
function real set_abstime(
   real t,
   real secs
)
```

Specifies the current canonical time value, in the number of specified time units, regardless of the current time scale of the caller.

secs is the number of seconds in the desired time unit, e.g., `1e-9` for nanoseconds.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

6. Reporting classes

6.1 Overview

The reporting classes provide a facility for issuing reports with consistent formatting. Users can configure what actions to take and what files to send output to based on report severity, ID, or both severity and ID. Users can also filter messages based on their verbosity settings.

The primary interface to the UVM reporting facility is the **uvm_report_object** (see 6.3) from which all **uvm_component**s extend (see 13.1). The **uvm_report_object** delegates most tasks to its internal **uvm_report_handler** (see 6.4). If the report handler determines the report is not filtered based on the configured verbosity setting, it sends the report to the central **uvm_report_server** (see 6.5.1) for formatting and processing.

6.2 uvm_report_message

The **uvm_report_message** is the basic UVM object message class. It provides the fields that are common to all messages. The report message object can be initialized with the common fields (see 6.2.2) and passes through the whole reporting system (i.e., report object, report handler, report server, report catcher) as an object. The additional elements can be added/deleted to/from the message object anywhere in the reporting system, and can be printed or recorded along with the common fields.

6.2.1 Class declaration

```
class uvm_report_message extends uvm_object
```

6.2.2 Common methods

6.2.2.1 new

```
function new(
    string name = "uvm_report_message"
)
```

Creates a new **uvm_report_message** object.

6.2.2.2 new_report_message

```
static function uvm_report_message new_report_message(
    string name = "uvm_report_message"
)
```

Creates a new **uvm_report_message** object. This function is the same as **new** (see 6.2.2.1), however this method will preserve the random stability of the calling thread. While it is legal to call this method from a non-thread context, the random stability of the non-thread context is not guaranteed.

6.2.2.3 do_print

```
virtual function void do_print(
    uvm_printer printer
)
```

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

The uvm_report_message implements **uvm_object::do_print** (see 5.3.6.3) such that the **uvm_report_message::print** method provides a UVM printer formatted output of the message.

6.2.3 Infrastructure references

6.2.3.1 get_report_object and set_report_object

```
virtual function uvm_report_object get_report_object()

virtual function void set_report_object(
    uvm_report_object ro
)
```

Returns or specifies the **uvm_report_object** (see 6.3) that originated the message.

6.2.3.2 get_report_handler and set_report_handler

```
virtual function uvm_report_handler get_report_handler()

virtual function void set_report_handler(
    uvm_report_handler rh
)
```

Returns or specifies the **uvm_report_handler** (see 6.4) that is responsible for checking whether the message is enabled, may be upgraded/downgraded, etc.

6.2.3.3 get_report_server and set_report_server

```
virtual function uvm_report_server get_report_server()

virtual function void set_report_server(
    uvm_report_server rs
)
```

Returns or specifies the **uvm_report_server** (see 6.5.1) that is responsible for servicing the message's actions.

6.2.4 Message fields

6.2.4.1 get_severity and set_severity

```
virtual function uvm_severity get_severity()

virtual function void set_severity(
    uvm_severity sev
)
```

Returns or specifies the severity (`UVM_INFO`, `UVM_WARNING`, `UVM_ERROR`, or `UVM_FATAL`) of the message.

6.2.4.2 get_id and set_id

```
virtual function string get_id()

virtual function void set_id(
    string id
)
```

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021

Returns or specifies the id of the message. The value of this field is completely under user discretion. See 6.4.

6.2.4.3 get_message and set_message

```
virtual function string get_message()

virtual function void set_message(
    string msg
)
```

Returns or specifies a user message content string.

6.2.4.4 get_verbosity and set_verbosity

```
virtual function int get_verbosity()

virtual function void set_verbosity(
    int ver
)
```

Returns or specifies the message verbosity threshold value. This value is compared against settings in the **uvm_report_handler** (see 6.4) to determine whether this message is executed.

6.2.4.5 get_filename and set_filename

```
virtual function string get_filename()

virtual function void set_filename(
    string fname
)
```

Returns or specifies the file from which the message originates.

6.2.4.6 get_line and set_line

```
virtual function int get_line()

virtual function void set_line(
    int ln
)
```

Returns or specifies the line in the file from which the message originates.

6.2.4.7 get_context and set_context

```
virtual function string get_context()

virtual function void set_context(
    string cn
)
```

Returns or specifies the optional user-supplied string that is meant to convey the context of the message.

Published by IEC under licence from IEEE. © 2017 IEEE. All rights reserved.

Documents
Document Sharing Hub

Full document details: BS IEC 62530-2:2021

6.2.4.8 get_action and set_action

```
virtual function uvm_action get_action()

virtual function void set_action(
   uvm_action act
)
```

Returns or specifies the action(s) the **uvm_report_server** (see 6.5.1) performs for this message.

6.2.4.9 get_file and set_file

```
virtual function UVM_FILE get_file()

virtual function void set_file(
   UVM_FILE fl
)
```

Returns or specifies the file to which the message is written when the message's action is `UVM_LOG`.

6.2.4.10 set_report_message

```
virtual function void set_report_message(
   uvm_severity severity,
   string id,
   string message,
   int verbosity,
   string filename,
   int line,
   string context_name
)
```

Specifies all the common fields of the report message in one function call.

6.3 uvm_report_object

The **uvm_report_object** provides an interface to the UVM reporting facility. Through this interface, various messages can be issued that occur during simulation. Users can configure what actions are taken and what file(s) are output for individual messages from a particular report object or for all messages from all report objects in the environment. Defaults are applied where there is no explicit configuration.

Most methods in **uvm_report_object** are delegated to an internal instance of a **uvm_report_handler** (see 6.4), which stores the reporting configuration and determines whether an issued message should be displayed based on that configuration. Then, to display a message, the report handler delegates the actual formatting and production of messages to a central **uvm_report_server** (see 6.5.1).

A report consists of the message fields described in 6.2.4. It may optionally include the filename and line number from which the message came. If a report has a verbosity level greater than the configured maximum verbosity level (see 6.3.4.1) of its report object, it is ignored. If a report passes the verbosity filter, in effect, the report's action is determined. If the action includes output to a file, the configured file descriptor(s) are determined.

Documents
Document Sharing Hub
Full document details: BS IEC 62530-2:2021