IEEE STANDARDS ASSOCIATION

IEEE Standard for Universal Verification Methodology Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

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

IEEE Std 1800.2™-2017

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

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

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

Print: ISBN 978-1-5044-4001-1 STDPD22567
PDF: ISBN 978-1-5044-4000-4 STDGT22567

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.

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.

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.

Participants

The Universal Verfication Methodology (UVM) Working Group is entity based. At the time this standard was completed, the Universal Verfication Methodology (UVM) Working Group had the following membership:

Tom Alsop, *Chair*
Hillel Miller, *Vice Chair*
Christeen Gray, *Secretary*
Joe Daniels, *Technical Editor*

Jamsheed Agahi
Mala Bandyopahdyay
Martin Barnasconi
Dennis Brophy
Joel Feldman
Tom Fitzpatrick
Courtney Fricano
Mark Glasser
Christeen Gray
Shuang Han
Shobhit Kapoor
Adiel Khan
Justin Refice
Uwe Simm
Mark Strickland
Srivatsa Vasudevan
Karl Whiting

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

Accellera Systems Initiative, Inc.
Advanced Micro Devices (AMD)
Analog Devices Inc.
Cadence Design Systems, Inc.
Cisco Systems, Inc.
IBM
Intel Corporation
Mentor Graphics
NVIDIA Corporation
NXP Semiconductors
Semifore, Inc.
Southwest Jiaotong University
Synopsys, Inc.
Verific Design Automation, Inc.

When the IEEE-SA Standards Board approved this standard on 14 February 2017, it had the following membership:

Jean-Philippe Faure, *Chair*
Vacant position, *Vice Chair*
John D. Kulick, *Past Chair*
Konstantinos Karachalios, *Secretary*

Chuck Adams
Masayuki Ariyoshi
Ted Burse
Stephen Dukes
Doug Edwards
J. Travis Griffith
Gary Hoffman
Michael Janezic
Thomas Koshy
Joseph L. Koepfinger*
Kevin Lu
Daleep Mohla
Damir Novosel
Ronald C. Petersen
Annette D. Reilly
Robby Robson
Dorothy Stanley
Adrian Stephens
Mehmet Ulema
Phil Wennblom
Howard Wolfman
Yu Yuan

*Member Emeritus

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.

Contents

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.

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 (`::`).

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.

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/.

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.

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.

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.

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 = ""
)
```

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**

#(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.

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
)
```

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
)
```

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**.

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).

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.

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
```

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.

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).

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.

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.

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

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`.

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).

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`.

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).

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
)
```

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)
```

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`.

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.

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.

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
)
```

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
)
```

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.

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.

a) *Actions*—These can be set for *severity*, *id*, and the (*severity,id*) pair. See 6.3.5.2.

b) *Default actions*—The following list highlights the default actions assigned to each severity. These can be overridden by any of the **set_*_action** methods.

   1) `UVM_INFO`—`UVM_DISPLAY`
   2) `UVM_WARNING`—`UVM_DISPLAY`
   3) `UVM_ERROR`—`UVM_DISPLAY|UVM_COUNT`
   4) `UVM_FATAL`—`UVM_DISPLAY|UVM_EXIT`

c) *File descriptors*—These can be specified as (in increasing priority) default, severity level, id, or (severity, id) pair. File descriptors are of `UVM_FILE` type (see F.2.8). It is the user's responsibility to open and close them.

d) *Default file handle*—The default file handle is `0`, which means reports are not sent to a file even if an `UVM_LOG` attribute is specified in the action associated with the report. This can be overridden by any of the **set_*_file** methods.

6.3.1 Class declaration

```
class uvm_report_object extends uvm_object
```

6.3.2 Common methods

new

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

Creates a new report object with the given name.

6.3.3 Reporting

6.3.3.1 uvm_get_report_object

```
function uvm_report_object uvm_get_report_object()
```

Returns this **uvm_report_object**. See also F.3.2.1.

6.3.3.2 uvm_report_enabled

```
function int uvm_report_enabled(
    int verbosity,
    uvm_severity severity = UVM_INFO,
    string id = ""
)
```

Returns `1` if the configured verbosity for this severity/id is greater than or equal to *verbosity*, else returns `0`. The default value of *severity* shall be `UVM_INFO`.

See also 6.3.4.1 and F.3.2.2.

6.3.3.3 uvm_report, uvm_report_info, uvm_report_warning, uvm_report_error, and uvm_report_fatal

```
virtual function void uvm_report(
    uvm_severity severity,
```

```
    string id,
    string message,
    int verbosity = (severity ==
                     uvm_severity'(UVM_ERROR)) ?
                     UVM_NONE : (severity ==
                     uvm_severity'(UVM_FATAL)) ?
                     UVM_NONE : UVM_MEDIUM,
    string filename = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)

virtual function void uvm_report_info(
    string id,
    string message,
    int verbosity = UVM_MEDIUM,
    string filename = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)

virtual function void uvm_report_warning(
    string id,
    string message,
    int verbosity = UVM_MEDIUM,
    string filename = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)

virtual function void uvm_report_error(
    string id,
    string message,
    int verbosity = UVM_NONE,
    string filename = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)

virtual function void uvm_report_fatal(
    string id,
    string message,
    int verbosity = UVM_NONE,
    string filename = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

These are the primary reporting methods in UVM. Using these instead of `$display` and other ad hoc approaches ensures consistent output and central control over where output is directed and any actions that result. All reporting methods have the following arguments, although each has a different default verbosity:

a) *id*—A string containing the id for the report or report group that can be used for identification and therefore targeted filtering. An individual report's actions and output file(s) can be configured using this `id` string.

b) *message*—The message body as a single string.

c) *verbosity*—A string containing the message body, indicating its relative importance. If this number is less than or equal to the maximum verbosity level (see 6.3.4.3), then the report is issued, subject to the configured action and file descriptor specifications. Verbosity is ignored for warnings, errors, and fatals. However, if a warning, error, or fatal is demoted to an info message using the **uvm_report_catcher** (see 6.6), then the verbosity is taken into account.

d) *filename/line* (optional)—A string containing filename and an integer for line number defining the location from which the report was issued. If specified, the location is displayed in the output. The default value of *filename* shall be an *empty string* (`""`) and the default value of *line* shall be `0`.

e) *context_name* (optional)—The string context from where the message is originating. This can be the `%m` of a module, a specific method, etc. The default value of *context_name* shall be an *empty string* (`""`).

f) *report_enabled_checked* (optional)—This bit indicates if the currently provided message has been checked as to whether the message should be processed. If it has not been checked, it will be checked as part of the **uvm_report** function. The default value shall be `0`.

6.3.3.4 uvm_process_report_message

```
virtual function void uvm_process_report_message(
    uvm_report_message report_message
)
```

This method takes a preformed **uvm_report_message** (see 6.2), populates it with the report object, and passes it to the report handler for processing; see 6.4.7.

6.3.4 Verbosity configuration

6.3.4.1 get_report_verbosity_level

```
function int get_report_verbosity_level(
    uvm_severity severity = UVM_INFO,
    string id = ""
)
```

Returns the verbosity level in effect for this object. This function calls the underlying report handler **get_verbosity_level** (see 6.4.3.1).

6.3.4.2 get_report_max_verbosity_level

```
function int get_report_max_verbosity_level()
```

Returns the maximum verbosity level in effect for this report object.

6.3.4.3 set_report_verbosity_level

```
function void set_report_verbosity_level (
    int verbosity_level
)
```

Specifies the maximum verbosity level for reports for this component. This function calls the underlying report handler **set_verbosity_level** (see 6.4.3.2).

6.3.4.4 set_report_id_verbosity and set_report_severity_id_verbosity

```
function void set_report_id_verbosity (
   string id,
   int verbosity
)

function void set_report_severity_id_verbosity (
   uvm_severity severity,
   string id,
   int verbosity
)
```

These methods associate the specified verbosity threshold with reports of the given *severity*, *id*, or *severity-id* pair. These functions call the underlying report handler **set_id_verbosity** or **set_severity_id_verbosity**, respectively (see 6.4.3.3).

6.3.5 Action configuration

6.3.5.1 get_report_action

```
function int get_report_action(
   uvm_severity severity,
   string id
)
```

Returns the action associated with reports having the given *severity* and *id*. This function calls the underlying report handler **get_action** (see 6.4.4.1).

6.3.5.2 set_report_severity_action, set_report_id_action, and set_report_severity_id_action

```
function void set_report_severity_action (
   uvm_severity severity,
   uvm_action action
)

function void set_report_id_action (
   string id,
   uvm_action action
)

function void set_report_severity_id_action (
   uvm_severity severity,
   string id,
   uvm_action action
)
```

These methods associate the specified action or actions with reports of the given *severity*, *id*, or *severity-id* pair. These functions call the underlying report handler **set_severity_action**, **set_id_action**, or **set_severity_id_action**, respectively (see 6.4.4.2).

6.3.6 File configuration

6.3.6.1 get_report_file_handle

```
function int get_report_file_handle(
   uvm_severity severity,
   string id
)
```

Returns the file descriptor associated with reports having the given *severity* and *id*. This function calls the underlying report handler **get_file_handle** (see 6.4.5.1).

6.3.6.2 set_report_default_file, set_report_id_file, set_report_severity_file, and set_report_severity_id_file

```
function void set_report_default_file (
   UVM_FILE file
)

function void set_report_id_file (
   string id,
   UVM_FILE file
)

function void set_report_severity_file (
   uvm_severity severity,
   UVM_FILE file
)

function void set_report_severity_id_file (
   uvm_severity severity,
   string id,
   UVM_FILE file
)
```

These methods configure the report handler to direct some or all of its output to the given *file* descriptor (see F.2.8). These functions call the underlying report handler **set_default_file**, **set_id_file**, **set_severity_file**, or **set_severity_id_file**, respectively (see 6.4.5.2).

6.3.7 Override configuration

set_report_severity_override and **set_report_severity_id_override**

```
function void set_report_severity_override(
   uvm_severity cur_severity,
   uvm_severity new_severity
)

function void set_report_severity_id_override(
   uvm_severity cur_severity,
   string id,
   uvm_severity new_severity
)
```

These methods provide the ability to upgrade or downgrade a message in terms of severity given the *severity* and *id*. These functions call the underlying report handler **set_severity_override** or **set_severity_id_override**, respectively (see 6.4.6).

6.3.8 Report handler configuration

6.3.8.1 get_report_handler

```
function uvm_report_handler get_report_handler()
```

Returns the underlying report handler to which most reporting tasks are delegated.

6.3.8.2 set_report_handler

```
function void set_report_handler(
   uvm_report_handler handler
)
```

Specifies the report handler, overwriting the previous value. This allows more than one component to share the same report handler.

6.3.8.3 reset_report_handler

```
function void reset_report_handler()
```

Resets the underlying report handler to its default settings (see 6.4.2.1). This clears any changes made with the override configuration methods (see 6.3.7).

6.4 uvm_report_handler

The **uvm_report_handler** is the class to which most methods in **uvm_report_object** (see 6.3) delegate. **uvm_report_handler** stores the maximum verbosity, actions, and files that affect the way reports are handled.

6.4.1 Class declaration

```
class uvm_report_handler extends uvm_object
```

6.4.2 Common methods

6.4.2.1 new

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

Creates and initializes a new **uvm_report_handler** object.

6.4.2.2 do_print

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

uvm_report_handler implements **uvm_object::do_print** (see 5.3.6.3) such that the **uvm_report_handler::print** method provides UVM printer formatted output of the current configuration.

6.4.3 Verbosity configuration

6.4.3.1 get_verbosity_level

```
function int get_verbosity_level (
    uvm_severity severity = UVM_INFO,
    string id = ""
)
```

Returns the verbosity level stored in this handler for the given *id* or *severity-id* pair.A verbosity associated with a particular *severity-id* pair takes precedence over a verbosity associated with *id*, which takes precedence over the maximum verbosity.The default value of *severity* shall be `UVM_INFO`.

6.4.3.2 set_verbosity_level

```
function void set_verbosity_level (
    int verbosity_level
)
```

Specifies the maximum verbosity level for this handler. Any report whose verbosity exceeds this maximum is ignored.

6.4.3.3 set_severity_id_verbosity and set_id_verbosity

```
function void set_severity_id_verbosity (
    uvm_severity severity,
    string id,
    int verbosity
)

function void set_id_verbosity (
    string id,
    int verbosity
)
```

These methods associate the specified *verbosity* level with the given *id* or *severity-id* pair. A verbosity associated with a particular *severity-id* pair takes precedence over a verbosity associated with *id*, which takes precedence over the maximum verbosity.

verbosity can be any integer, but is most commonly a predefined **uvm_verbosity** value (**s**ee F.2.2.4).

6.4.4 Action configuration

6.4.4.1 get_action

```
function uvm_action get_action (
    uvm_severity severity,
    string id
)
```

Returns the action (**uvm_action**, see F.2.2.3) stored in this handler for the given *severity*, *id*, or *severity-id* pair. An *action* associated with a particular *severity-id* pair takes precedence over an *action* associated with *id*, which takes precedence over an *action* associated with *severity*.

6.4.4.2 set_severity_id_action, set_id_action, and set_severity_action

```
function void set_severity_id_action (
   uvm_severity severity,
   string id,
   uvm_action action
)

function void set_id_action (
   string id,
   uvm_action action
)

function void set_severity_action (
   uvm_severity severity,
   uvm_action action
)
```

These methods associate the specified action with the given *severity*, *id*, or *severity-id* pair. An *action* associated with a particular *severity-id* pair takes precedence over an *action* associated with *id*, which takes precedence over an *action* associated with *severity*. *action* can take the value `UVM_NO_ACTION` or it can be a bitwise `OR` of any combination of the other **uvm_action_types** enum values, see F.2.2.2.

6.4.5 File configuration

6.4.5.1 get_file_handle

```
function UVM_FILE get_file_handle (
   uvm_severity severity,
   string id
)
```

Returns the file descriptor (`UVM_FILE`, see F.2.8) stored in the handler associated with the given *severity*, *id*, or *severity-id* pair. A *file* associated with a particular *severity-id* pair takes precedence over a *file* associated with *id*, which takes precedence over a *file* associated with a *severity*, which takes precedence over the default *file* descriptor.

6.4.5.2 set_severity_id_file, set_id_file, set_severity_file, and set_default_file

```
function void set_severity_id_file (
   uvm_severity severity,
   string id,
   UVM_FILE file
)

function void set_id_file (
   string id,
   UVM_FILE file
)

function void set_severity_file (
   uvm_severity severity,
   UVM_FILE file
)

function void set_default_file (
   UVM_FILE file
)
```

These methods configure this handler to direct some or all of its output to the given *file* descriptor, where *file* is a multi-channel descriptor (mcd) or a file id compatible with `$fdisplay`.

A *file* descriptor can be associated with reports of the given *severity*, *id*, or *severity-id* pair. A *file* associated with a particular *severity-id* pair takes precedence over a *file* associated with *id*, which takes precedence over a *file* associated with a *severity*, which takes precedence over the default *file* descriptor.

When a report is issued and its associated action has the `UVM_LOG` bit specified, the report is then sent to its associated `file` descriptor. The user is responsible for opening and closing these files.

6.4.6 Override configuration

set_severity_override and **set_severity_id_override**

```
function void set_severity_override (
   uvm_severity cur_severity,
   uvm_severity new_severity
)

function void set_severity_id_override (
   uvm_severity cur_severity,
   string id,
   uvm_severity new_severity
)
```

These methods provide the ability to upgrade or downgrade a message in terms of severity (*new_severity*) given the *cur_severity* and *id*. An upgrade or downgrade for a specific *id* takes precedence over an upgrade or downgrade associated with *cur_severity*.

6.4.7 Message processing

process_report_message

```
virtual function void process_report_message(
   uvm_report_message report_message
)
```

This is the common handler method used by the core reporting methods (e.g., **uvm_report_error**) in **uvm_report_object** (see 6.3.3.3).

6.5 Report server

This subclause covers the classes that define the UVM report server facility.

6.5.1 uvm_report_server

```
virtual class uvm_report_server extends uvm_object
```

Implementations of **uvm_report_server** process all of the reports generated by an **uvm_report_handler** (see 6.4).

The **uvm_report_server** is an abstract class that declares many of its methods as `pure virtual`. UVM uses the **uvm_default_report_server** class (see 6.5.2) as its default report server implementation.

uvm_report_server has the following *Methods*.

6.5.1.1 get_max_quit_count

```
pure virtual function int get_max_quit_count()
```

Intended to return the currently configured max quit count.

6.5.1.2 set_max_quit_count

```
pure virtual function void set_max_quit_count(
   int count,
   bit overridable = 1
)
```

Intended to set the currently configured max quit count.

count is the maximum number of `UVM_QUIT` actions the **uvm_report_server** can tolerate before invoking `client.die`. When `overridable = 0` is passed, the *count* cannot be changed again. The default value of *overridable* shall be `1`.

6.5.1.3 get_quit_count

```
pure virtual function int get_quit_count()
```

Intended to return the current number of `UVM_QUIT` actions already passed through this server.

6.5.1.4 set_quit_count

```
pure virtual function void set_quit_count(
   int quit_count
)
```

Intended to specify the current number of `UVM_QUIT` actions already passed through this **uvm_report_server**.

6.5.1.5 get_severity_count

```
pure virtual function int get_severity_count(
   uvm_severity severity
)
```

Intended to return the count of already passed messages with severity *severity*.

6.5.1.6 set_severity_count

```
pure virtual function void set_severity_count(
   uvm_severity severity,
   int count
)
```

Intended to specify the count of already passed messages with severity *severity* to *count*.

6.5.1.7 get_id_count

```
pure virtual function int get_id_count(
   string id
)
```

Intended to return the count of already passed messages with *id*.

6.5.1.8 get_id_set

```
pure virtual function void get_id_set(
    output string q[$]
)
```

Intended to return the set of id's already used by this **uvm_report_server**. *q* shall be a queue.

6.5.1.9 get_severity_set

```
pure virtual function void get_severity_set(
    output uvm_severity q[$]
)
```

Intended to return the set of severities already used by this **uvm_report_server**. *q* shall be a queue.

6.5.1.10 get_message_database

```
pure virtual function uvm_tr_database get_message_database()
```

Intended to return the **uvm_tr_database** (see 7.1) used for recording messages.

6.5.1.11 set_message_database

```
pure virtual function void set_message_database(
    uvm_tr_database database
)
```

Intended to specify the **uvm_tr_database** (see 7.1) used for recording messages.

6.5.1.12 do_copy

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

Copies all message statistic *severity,id* counts to the destination **uvm_report_server**. The copy is cumulative: items from the source are transferred, existing entries are not deleted, and existing entries/ counts are overridden when they exist in the source set. *rhs* shall be a **uvm_report_server** or derived from one.

6.5.1.13 process_report_message

```
pure virtual function void process_report_message(
    uvm_report_message report_message
)
```

This method is the main entry point for the **uvm_report_server**, processing the provided **uvm_report_message** (see 6.2). The report server shall take the following steps, in order:

a) All report catchers (see 6.6) that are currently registered and active are processed. If any call to **catch** (see 6.6.5) returns `CAUGHT`, **process_report_message** ends immediately.

b) After processing all report catchers, if the message contains an action of `UVM_NO_ACTION`, **process_report_message** ends immediately.

c) If the message actions include one or both of `UVM_LOG` or `UVM_DISPLAY`, **compose_report_message** (see 6.5.1.14) is called.

d) Finally, **execute_report_message** (see 6.5.1.15) is called.

6.5.1.14 compose_report_message

```
pure virtual function string compose_report_message(
    uvm_report_message report_message,
    string report_object_name = ""
)
```

Intended to construct the actual string sent to the file or command line from the severity, component name, report id, and the message itself. *report_object_name* can be specified to override the use of the `report_handler` full name in the message. Users can overload this method to customize report formatting.

6.5.1.15 execute_report_message

```
pure virtual function void execute_report_message(
    uvm_report_message report_message,
    string composed_message
)
```

Processes the provided *report_message* per the actions contained within. *composed_message* gets logged or displayed if the *report_message* calls for that action. Users can overload this method to customize action processing.

6.5.1.16 report_summarize

```
pure virtual function void report_summarize(
    UVM_FILE file = UVM_STDOUT
)
```

Intended to output statistical information on the reports issued by this central report server. This information is sent to the file descriptor *file*. The default value of *file* shall be `UVM_STDOUT`.

6.5.1.17 get_server

```
static function uvm_report_server get_server()
```

Returns the global report server used for reporting.

This method is provided as a wrapper function to conveniently retrieve the report server via the **uvm_coreservice_t::get_report_server** method (see F.4.1.4.4).

6.5.1.18 set_server

```
static function void set_server(
    uvm_report_server server
)
```

Specifies the global report server to use for reporting.

This method is provided as a wrapper function to conveniently specify the report server via the **uvm_coreservice_t::set_report_server** method (see F.4.1.4.4). In addition to specifying the server, this also copies the severity/id counts from the current report server to the new one.

6.5.2 uvm_default_report_server

Default implementation of the UVM report server.

Class declaration

```
class uvm_default_report_server extends uvm_report_server
```

uvm_default_report_server can be extended (from **uvm_report_server**); it provides a full implementation of all **uvm_report_server**'s methods (see 6.5.1).

6.6 uvm_report_catcher

The **uvm_report_catcher** is used to catch messages issued by the UVM report server. Catchers are **uvm_callbacks**, so all facilities in the **uvm_callback** (see 10.7.1) and **uvm_callbacks#(T,CB)** (see 10.7.2) classes are available for registering catchers and controlling catcher state. The **uvm_callbacks#(uvm_report_object,uvm_report_catcher)** class is also aliased to **uvm_report_cb** (see D.4.4).

Multiple report catchers can be registered with a report object. The catchers can be registered as default catchers, which catch all reports on all **uvm_report_object** reporters (see 6.3), or catchers can be attached to specific report objects (e.g., components).

User extensions of **uvm_report_catcher** shall implement the **catch** method (see 6.6.5.), in which the action to be taken on catching the report is specified. The **catch** method can return `CAUGHT`, in which case further processing of the report is immediately stopped, or return `THROW`, in which case the (possibly modified) report is passed on to other registered catchers.

On catching a report, the **catch** method can modify the severity, id, action, verbosity, or the report string itself before the report is finally issued by the report server. The report can be immediately issued from within the catcher class by calling the **issue** method (see 6.6.6.6).

The catcher maintains a count of all reports with `UVM_FATAL`, `UVM_ERROR`, or `UVM_WARNING` (see F.2.2.1) severity and a count of all reports with `UVM_FATAL`, `UVM_ERROR`, or `UVM_WARNING` severity whose severity was lowered. These statistics are reported in the summary of **uvm_report_server** (see 6.5.1).

6.6.1 Class declaration

```
virtual class uvm_report_catcher extends uvm_callback
```

6.6.2 Common methods

new

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

Initializes a new report catcher. The *name* argument is optional, but should generally be provided to aid in debugging.

6.6.3 Current message state

6.6.3.1 get_client

```
function uvm_report_object get_client()
```

Returns the **uvm_report_object** (see 6.3) that has generated the message currently being processed.

6.6.3.2 get_severity

```
function uvm_severity get_severity()
```

Returns the **uvm_severity** (see F.2.2.1) of the message that is currently being processed. If the severity was modified by a previously executed catcher object (which re-threw the message), the returned severity is the modified value.

6.6.3.3 get_context

```
function string get_context()
```

Returns the context name of the message that is currently being processed. This is typically the full hierarchical name of the component that issued the message. However, if user-defined context is specified in **uvm_report_message** (see 6.2), the user-defined context is returned.

6.6.3.4 get_verbosity

```
function int get_verbosity()
```

Returns the verbosity of the message that is currently being processed. If the verbosity was modified by a previously executed catcher (which re-threw the message), the returned verbosity is the modified value.

6.6.3.5 get_id

```
function string get_id()
```

Returns the string id of the message that is currently being processed. If the id was modified by a previously executed catcher (which re-threw the message), the returned id is the modified value.

6.6.3.6 get_message

```
function string get_message()
```

Returns the string message of the message that is currently being processed. If the message was modified by a previously executed catcher (which re-threw the message), the returned message is the modified value.

6.6.3.7 get_action

```
function uvm_action get_action()
```

Returns the **uvm_action** (see F.2.2.3) of the message that is currently being processed. If the action was modified by a previously executed catcher (which re-threw the message), the returned action is the modified value.

6.6.3.8 get_fname

```
function string get_fname()
```

Returns the file name of the message.

6.6.3.9 get_line

```
function int get_line()
```

Returns the line number of the message.

6.6.3.10 get_report_message

```
function uvm_report_message get_report_message()
```

Returns the report message object (see 6.2) for this report.

6.6.4 Change message state

6.6.4.1 set_severity

```
protected function void set_severity(
   uvm_severity severity
)
```

Changes the severity of the message to *severity*. Any other report catchers will see the modified value.

6.6.4.2 set_verbosity

```
protected function void set_verbosity(
   int verbosity
)
```

Changes the verbosity of the message to *verbosity*. Any other report catchers will see the modified value.

6.6.4.3 set_id

```
protected function void set_id(
   string id
)
```

Changes the id of the message to *id*. Any other report catchers will see the modified value.

6.6.4.4 set_message

```
protected function void set_message(
   string message
)
```

Changes the text of the message to *message*. Any other report catchers will see the modified value.

6.6.4.5 set_action

```
protected function void set_action(
    uvm_action action
)
```

Changes the action of the message to *action*. Any other report catchers will see the modified value.

6.6.4.6 set_context

```
protected function void set_context(
    string context_str
)
```

Changes the context of the message to *context_str*. Any other report catchers will see the modified value.

6.6.5 Callback interface

catch

```
typedef enum {UNKNOWN_ACTION, THROW, CAUGHT} action_e
pure virtual function action_e catch()
```

This is the method that is intended to be called for each registered report catcher. There are no arguments to this function. The interface methods in 6.6.3 can be used to access information about the current message being processed.

6.6.6 Reporting

6.6.6.1 uvm_report_fatal

```
protected function void uvm_report_fatal(
    string id,
    string message,
    int verbosity,
    string fname = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

Generates a fatal message using the current message's report object. This message bypasses any message catching callbacks. The default values of *line* and *report_enabled_checked* shall be `0`. The default values of *fname* and *context_name* shall be `an` *empty string* (`""`).

6.6.6.2 uvm_report_error

```
protected function void uvm_report_error(
    string id,
    string message,
    int verbosity,
    string fname = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

Generates an error message using the current message's report object. This message bypasses any message catching callbacks. The default values of *line* and *report_enabled_checked* shall be `0`. The default values of *fname* and *context_name* shall be `an` *empty string* (`""`).

6.6.6.3 uvm_report_warning

```
protected function void uvm_report_warning(
    string id,
    string message,
    int verbosity,
    string fname = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

Issues a warning message using the current message's report object. This message bypasses any message catching callbacks. The default values of *line* and *report_enabled_checked* shall be `0`. The default values of *fname* and *context_name* shall be an *empty string* (`""`).

6.6.6.4 uvm_report_info

```
protected function void uvm_report_info(
    string id,
    string message,
    int verbosity,
    string fname = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

Issues an info message using the current message's report object. This message bypasses any message catching callbacks. The default values of *line* and *report_enabled_checked* shall be `0`. The default values of *fname* and *context_name* shall be an *empty string* (`""`).

6.6.6.5 uvm_report

```
protected function void uvm_report(
    uvm_severity severity,
    string id,
    string message,
    int verbosity,
    string fname = "",
    int line = 0,
    string context_name = "",
    bit report_enabled_checked = 0
)
```

Issues a message using the current message's report object. This message bypasses any message catching callbacks. The default values of *line* and *report_enabled_checked* shall be `0`. The default values of *fname* and *context_name* shall be an *empty string* (`""`).

6.6.6.6 issue

```
protected function void issue()
```

Immediately issues the message that is currently being processed. This is useful if the message is being `CAUGHT`, but should still be emitted.

Issuing a message updates the `report_server` stats, possibly multiple times if the message is not `CAUGHT`.

7. Recording classes

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.

The primary interface to the recording facility is the **uvm_tr_database** (see 7.1), which represents the application-specific mechanism that is recording the transactions. Transactions within the database are grouped logically within streams, which are represented by the **uvm_tr_stream** class (see 7.2). Finally, each transaction in the database is represented by a **uvm_recorder** (see 16.4.1), which additionally serves as the policy that is provided to the **uvm_object::record** method (see 5.3.7.1).

7.1 uvm_tr_database

The **uvm_tr_database** class is pure virtual and needs to be extended with an implementation.

NOTE—The **uvm_tr_database** class is intended to abstract the underlying database implementation from the user, as the details of the database are often specific to the database implementation.

7.1.1 Class declaration

```
virtual class uvm_tr_database extends uvm_object
```

7.1.2 Common methods

new

```
function new(
    string name = "unnamed-uvm_tr_database"
)
```

This is a constructor; it has the following parameter:

name—Instance name.

The default value of *name* shall be `"unnamed-uvm_tr_database"`.

7.1.3 Database API

7.1.3.1 open_db

```
function bit open_db()
```

Opens the back-end connection to the database. If the database is already open, this method returns '`1`'. Otherwise, it calls **do_open_db** (see 7.1.6.1) and returns the result. A return value of '`0`' indicates the database could not be opened.

7.1.3.2 close_db

```
function bit close_db()
```

Closes the back-end connection to the database. Closing a database closes and frees all **uvm_tr_stream**s within the database. If the database is already closed, i.e., **is_open** (see 7.1.3.3) returns '`0`', this method returns '`1`'. Otherwise, it calls **do_close_db** (see 7.1.6.2) and returns the result.

7.1.3.3 is_open

```
function bit is_open()
```

Returns the open/closed status of the database. This method returns '1' if the database has been successfully opened, but not yet closed. A return value of '0' indicates the database is not currently open.

7.1.4 Stream API

7.1.4.1 open_stream

```
function uvm_tr_stream open_stream(
    string name,
    string scope = "",
    string type_name = ""
)
```

Provides a reference to a *stream* within the database; it has the following parameters:

name—A string name for the stream. This is the name associated with the stream in the database.

scope—An optional scope for the stream.

type_name—An optional name describing the type of records to be created in this stream.

This method returns a reference to a **uvm_tr_stream** object (see 7.2) if successful, *null* otherwise.

This method also calls **do_open_stream** (see 7.1.6.3); if a non-*null* stream is returned, then **uvm_tr_stream::do_open** (see 7.2.7.1) is called.

Streams can only be opened if the database is open [per **is_open** (see 7.1.3.3)]; otherwise, the request is ignored and *null* is returned.

7.1.4.2 get_streams

```
function unsigned get_streams(
    ref uvm_tr_stream q[$]
)
```

Provides a queue of all streams within the database; it has the following parameters:

q—A reference to a queue of **uvm_tr_stream**s (see 7.2).

The **get_streams** method returns the size of the queue, such that the user can conditionally process the elements.

7.1.5 Link API

establish_link

```
function void establish_link(
    uvm_link_base link
)
```

Establishes a *link* between two elements in the database.

This method also calls **do_establish_link** (see 7.1.6.4).

7.1.6 Implementation agnostic API

7.1.6.1 do_open_db

```
pure virtual protected function bit do_open_db()
```

Intended to be the back-end implementation of **open_db** (see 7.1.3.1). A return value of '1' indicates the database was successfully opened. A return value of '0' indicates the database could not be opened.

7.1.6.2 do_close_db

```
pure virtual protected function bit do_close_db()
```

Intended to be the back-end implementation of **close_db** (see 7.1.3.2). A return value of '1' indicates the database was successfully closed; whereas, a return value of '0' indicates the database could not be closed.

7.1.6.3 do_open_stream

```
pure virtual protected function uvm_tr_stream do_open_stream(
    string name,
    string scope,
    string type_name
)
```

Intended to be the back-end implementation of **open_stream** (see 7.1.4.1).

7.1.6.4 do_establish_link

```
pure virtual protected function void do_establish_link(
    uvm_link_base link
)
```

Intended to be the back-end implementation of **establish_link** (see 7.1.5).

7.2 uvm_tr_stream

The **uvm_tr_stream** base class is a representation of a stream of records within a **uvm_tr_database** (see 7.1).

The **uvm_tr_stream** class is abstract and needs to be extended with an implementation.

NOTE—The record stream is intended to abstract the underlying database implementation from the user, as the details of the database are often specific to the database implementation.

7.2.1 Class declaration

```
virtual class uvm_tr_stream extends uvm_object
```

7.2.2 Common methods

new

```
function new(
    string name = "unnamed-uvm_tr_stream"
)
```

This is a constructor; it has the following parameter:

name—Stream instance name.

7.2.3 Introspection API

7.2.3.1 get_db

```
function uvm_tr_database get_db()
```

Returns a reference to the database that contains this stream.

A warning shall be issued if **get_db** is called prior to the stream being initialized via **do_open** (see 7.2.7.1).

7.2.3.2 get_scope

```
function string get_scope()
```

Returns the scope supplied when opening this stream.

A warning shall be issued if **get_scope** is called prior to the stream being initialized via **do_open** (see 7.2.7.1).

7.2.3.3 get_stream_type_name

```
function string get_stream_type_name()
```

Returns a string with the type name.

A warning shall be issued if **get_stream_type_name** is called prior to the stream being initialized via **do_open** (see 7.2.7.1).

7.2.4 Stream API

Once a stream has been opened via **uvm_tr_database::open_stream** (see 7.1.4.1), the user can *close* the stream.

The act of *freeing* a stream is a signal from the user to the database developer that it is safe to release any internal references to the stream, as the user will not be accessing it again.

A *link* can be established within the database any time between “Open” and “Free,” however it shall be an error to establish a link after “Freeing” the stream.

7.2.4.1 close

```
function void close()
```

Closes the stream.

Closing a stream closes all open recorders in the stream. This method triggers a **do_close** call (see 7.2.7.2), followed by **uvm_recorder::close** (see 16.4.4.2) on all open recorders within the stream.

7.2.4.2 free

```
function void free()
```

Frees this stream.

Freeing a stream indicates that the database can free any references to the stream (including references to records within the stream). This method triggers a **do_free** call (see 7.2.7.3), followed by **uvm_recorder::free** (see 16.4.4.3) on all recorders within the stream.

7.2.4.3 is_open

```
function bit is_open()
```

Returns `true` if this **uvm_tr_stream** was opened on the database, but has not yet been closed; otherwise returns `false`.

7.2.4.4 is_closed

```
function bit is_closed()
```

Returns `true` if this **uvm_tr_stream** was closed on the database, but has not yet been freed; otherwise returns `false`.

7.2.5 Transaction recorder API

New recorders can be opened prior to the stream being *closed*. Once a stream has been closed, requests to open a new recorder are ignored (**open_recorder** (see 7.2.5.1) returns *null*).

7.2.5.1 open_recorder

```
function uvm_recorder open_recorder(
    string name,
    time open_time = 0,
    string type_name = ""
)
```

Marks the opening of a new transaction recorder on the stream; it has the following parameters:

name—A name for the new transaction.

open_time—The optional time to record as the opening of this transaction.

type_name—The optional type name for the transaction.

The default value of *open_time* shall be `0`.

If *open_time* is omitted (or specified as '`0`'), the stream uses the current time.

This method triggers a **do_open_recorder** call (see 7.2.7.4). If **do_open_recorder** returns a non-*null* value, the **uvm_recorder::do_open** method (see 16.4.7.1) is called in the recorder.

Transaction recorders can only be opened if the stream is *open* on the database [per **is_open** (see 7.2.4.3)]. Otherwise, the request is ignored and *null* is returned.

7.2.5.2 get_recorders

```
function unsigned get_recorders(
    ref uvm_recorder q[$]
)
```

Provides a queue of all transactions within the stream; it has the following parameter:

q—A reference to the queue of **uvm_recorder**s (see 16.4.1).

The **get_recorders** method returns the size of the queue, such that the user can conditionally process the elements.

7.2.6 Handles

7.2.6.1 get_handle

```
function int get_handle()
```

Returns a unique ID for this stream.

A value of `0` indicates the recorder has been *freed* and no longer has a valid ID.

7.2.6.2 get_stream_from_handle

```
static function uvm_tr_stream get_stream_from_handle(
    int id
)
```

Static accessor, returns a stream reference for a given unique id.

If no stream exists with the given *id* or the stream with that *id* has been freed, then *null* is returned.

7.2.7 Implementation agnostic API

7.2.7.1 do_open

```
protected virtual function void do_open(
    uvm_tr_database db,
    string scope,
    string stream_type_name
)
```

Callback triggered via **uvm_tr_database::open_stream** (see 7.1.4.1); it has the following parameters:

db—Database to which the stream belongs.

scope—The optional scope.

stream_type_name—The optional type name for the stream.

The **do_open** callback can be used to initialize any internal state within the stream, as well as providing a location to record any initial information about the stream.

7.2.7.2 do_close

```
protected virtual function void do_close(
```

Callback triggered via **close** (see 7.2.4.1).

The **do_close** callback can be used to specify an internal state within the stream, as well as providing a location to record any closing information.

7.2.7.3 do_free

```
protected virtual function void do_free(
```

Callback triggered via **free** (see 7.2.4.2).

The **do_free** callback can be used to release the internal state within the stream, as well as providing a location to record any "freeing" information.

7.2.7.4 do_open_recorder

```
protected virtual function uvm_recorder do_open_recorder(
    string name,
    time open_time,
    string type_name
)
```

Marks the beginning of a new record in the stream. This is a back-end implementation of **open_recorder** (see 7.2.5.1).

A *null* return value implies that the recorder could not be opened (for whatever reason). Users should do a *null* check on the return value of **open_recorder** (see 7.2.5.1).

7.3 UVM links

The **uvm_link_base** class (see 7.3.1), and its extensions, are provided as a mechanism to allow for compile-time safety when trying to establish links between records within **uvm_tr_database**s (see 7.1).

7.3.1 uvm_link_base

The **uvm_link_base** class presents a simple API for defining a link between any two objects.

Using extensions of this class, a **uvm_tr_database** (see 7.1) can determine the type of links being passed, without relying on any arbitrary string names.

7.3.1.1 Class declaration

```
virtual class uvm_link_base extends uvm_object
```

7.3.1.2 Common methods

new

```
function new(
    string name = "unnamed-uvm_link_base"
)
```

This is a constructor; it has the following parameter:

name—Instance name. The default value of *name* shall be "unnamed-uvm_link_base".

7.3.1.3 Accessors

7.3.1.3.1 get_lhs

```
function uvm_object get_lhs()
```

Returns the left-hand side of the link.

Triggers the **do_get_lhs** callback (see 7.3.1.4.1).

7.3.1.3.2 set_lhs

```
function void set_lhs(
   uvm_object lhs
)
```

Specifies the left-hand side of the link.

Triggers the **do_set_lhs** callback (see 7.3.1.4.2).

7.3.1.3.3 get_rhs

```
function uvm_object get_rhs()
```

Returns the right-hand side of the link.

Triggers the **do_get_rhs** callback (see 7.3.1.4.2).

7.3.1.3.4 set_rhs

```
function void set_rhs(
   uvm_object rhs
)
```

Specifies the right-hand side of the link.

Triggers the **do_set_rhs** callback (see 7.3.1.4.4).

7.3.1.3.5 set

```
function void set(
   uvm_object lhs,
   uvm_object rhs
)
```

This is a convenience method for setting both sides in one call.

Triggers both the **do_set_rhs** (see 7.3.1.4.4) and **do_set_lhs** (see 7.3.1.4.2) callbacks.

7.3.1.4 Implementation hooks

7.3.1.4.1 do_get_lhs

```
pure virtual function uvm_object do_get_lhs()
```

Intended to be the callback for retrieving the left-hand side.

7.3.1.4.2 do_set_lhs

```
pure virtual function void do_set_lhs(
   uvm_object lhs
)
```

Intended to be the callback for setting the left-hand side.

7.3.1.4.3 do_get_rhs

```
pure virtual function uvm_object do_get_rhs()
```

Intended to be the callback for retrieving the right-hand side.

7.3.1.4.4 do_set_rhs

```
pure virtual function void do_set_rhs(
   uvm_object rhs
)
```

Intended to be the callback for setting the right-hand side.

7.3.2 uvm_parent_child_link

The **uvm_parent_child_link** class is used to represent a parent/child relationship between two objects.

7.3.2.1 Class declaration

```
class uvm_parent_child_link extends uvm_link_base
```

7.3.2.2 Common methods

7.3.2.2.1 new

```
function new(
   string name = "unnamed-uvm_parent_child_link"
)
```

This is a constructor; it has the following parameter:

name—Instance name. The default value of *name* shall be `"unnamed-uvm_parent_child_link"`.

7.3.2.2.2 get_link

```
static function uvm_parent_child_link get_link(
   uvm_object lhs,
```

```
    uvm_object rhs,
    string name = "pc_link"
)
```

Constructs a prefilled link; it has the following parameters:

lhs—Left-hand-side reference.

rhs—Right-hand-side reference.

name—Optional name for the link object. The default value of *name* shall be `"pc_link"`.

This allows for simple one-line link creations, e.g.,

```
my_db.establish_link(uvm_parent_child_link::get_link(record1, record2))
```

7.3.3 uvm_cause_effect_link

The **uvm_cause_effect_link** is used to represent a cause/effect relationship between two objects.

7.3.3.1 Class declaration

```
class uvm_cause_effect_link extends uvm_link_base
```

7.3.3.2 Common methods

7.3.3.2.1 new

```
function new(
    string name = "unnamed-uvm_cause_effect_link"
)
```

This is a constructor; it has the following parameter:

name—Instance name. The default value of *name* shall be `"unnamed-uvm_cause_effect_link"`.

7.3.3.2.2 get_link

```
static function uvm_cause_effect_link get_link(
    uvm_object lhs,
    uvm_object rhs,
    string name = "ce_link"
)
```

Constructs a prefilled link; it has the following parameters:

lhs—Left-hand-side reference.

rhs—Right-hand-side reference.

name—Optional name for the link object. The default value of *name* shall be `"ce_link"`.

This allows for simple one-line link creations, e.g.,

```
my_db.establish_link(uvm_cause_effect_link::get_link(record1, record2))
```

7.3.4 uvm_related_link

The **uvm_related_link** is used to represent a generic “is related” link between two objects.

7.3.4.1 Class declaration

```
class uvm_related_link extends uvm_link_base
```

7.3.4.2 Common methods

7.3.4.2.1 new

```
function new(
    string name = "unnamed-uvm_related_link"
)
```

This is a constructor; it has the following parameter:

name—Instance name. The default value of *name* shall be "`unnamed-uvm_related_link`".

7.3.4.2.2 get_link

```
static function uvm_related_link get_link(
    uvm_object lhs,
    uvm_object rhs,
    string name = "ce_link"
)
```

Constructs a prefilled link; it has the following parameters:

lhs—Left-hand-side reference.
rhs—Right-hand-side reference.
name—Optional name for the link object. The default value of *name* shall be "`ce_link`".

This allows for simple one-line link creations, e.g.,

```
my_db.establish_link(uvm_cause_effect_link::get_link(record1, record2))
```

8. Factory classes

8.1 Overview

As the name implies, the **uvm_factory** (see 8.3.1) is used to manufacture (create) UVM objects and components. Only one instance of the factory is present in a given simulation.

User-defined object and component types are registered with the factory via `typedef` or `macro` invocation, see 8.3.1.8. The factory generates and stores lightweight proxies to the user-defined objects and components.

When the user requests a new object or component from the factory (e.g., using `uvm_factory::create_object_by_type`), the factory determines what type of object to create based on its configuration, then asks that type's proxy to create an instance of the type, which is returned to the user.

8.2 Factory component and object wrappers

8.2.1 Introduction

This subclause defines the proxy component and object classes used by the factory. To avoid the overhead of creating an instance of every component and object that are registered, the factory holds lightweight wrappers, or proxies. When a request for a new object is made, the factory calls upon the proxy to create the object it represents.

8.2.2 type_id

All classes derived from **uvm_object** (see 5.3) within the UVM package (see 1.3.5) shall have a proxy declared as **type_id**, unless explicitly stated otherwise.

This **type_id** declaration takes the form of

typedef *proxy_type* **type_id**

where *proxy_type* is one of the following:

a) `uvm_component_registry #(TYPE, "TYPE")` (see 8.2.3)—For non-abstract, non-parameterized derivatives of **uvm_component** (see 13.1).

b) `uvm_abstract_component_registry #(TYPE, "TYPE")` (see 8.2.5.1)—For abstract, non-parameterized derivatives of **uvm_component** (see 13.1).

c) `uvm_component_registry #(TYPE)` (see 8.2.3)—For non-abstract, parameterized derivatives of **uvm_component** (see 13.1).

d) `uvm_abstract_component_registry #(TYPE)` (see 8.2.5.1)—For abstract, parameterized derivatives of **uvm_component** (see 13.1).

e) `uvm_object_registry #(TYPE, "TYPE")` (see 8.2.4)—For non-abstract, non-parameterized derivatives of **uvm_object** (see 5.3) that do not derive from **uvm_component** (see 13.1).

f) `uvm_abstract_object_registry #(TYPE, "TYPE")` (see 8.2.5.2)—For abstract, non-parameterized derivatives of **uvm_object** (see 5.3) that do not derive from **uvm_component** (see 13.1).

g) `uvm_object_registry #(TYPE)` (see 8.2.4)—For non-abstract, parameterized derivatives of **uvm_object** (see 5.3) that do not derive from **uvm_component** (see 13.1).

h) `uvm_abstract_object_registry #(TYPE)` (see 8.2.5.2)—For abstract, parameterized derivatives of **uvm_object** (see 5.3) that do not derive from **uvm_component** (see 13.1).

8.2.3 uvm_component_registry #(T,Tname)

The **uvm_component_registry** serves as a lightweight proxy for a component of type *T* and type name *Tname*, a string. The proxy enables efficient registration with the **uvm_factory** (see 8.3.1). Without it, registration would require an instance of the component itself.

8.2.3.1 Class declaration

```
class uvm_component_registry #(
    type T = uvm_component,
    string Tname = "<unknown>"
) extends uvm_object_wrapper
```

The default value of *Tname* shall be `"<unknown>"`.

8.2.3.2 Methods

8.2.3.2.1 create_component

```
virtual function uvm_component create_component (
    string name,
    uvm_component parent
)
```

Creates a component of type *T* using the provided *name* and *parent*. This is an override of the method in **uvm_object_wrapper** (see 8.3.2). It is called by the factory after determining the type of object to create and the user can then implement it.

NOTE—Users should not call this method directly, they should use **create** instead (see 8.2.3.2.4).

8.2.3.2.2 get_type_name

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

Returns the value given by the string parameter, *Tname* by default.

8.2.3.2.3 get

```
static function uvm_component_registry #(T,Tname) get()
```

Returns a singleton instance.

The singleton instance is registered for initialization via **uvm_init** (see F.3.1.3) during static initialization. If **uvm_init** has been called prior to this registration occurring, the instance's **initialize** method (see 8.2.3.2.7) is called automatically during static initialization.

8.2.3.2.4 create

```
static function T create(
    string name,
```

```
    uvm_component parent,
    string contxt = ""
)
```

Returns an instance of the component type *T*, represented by this proxy, subject to any factory overrides based on the context provided by *contxt* if it is not an *empty string* (`""`) or otherwise provided by the *parent*'s full name. The new instance uses the given leaf *name* and *parent*.

8.2.3.2.5 set_type_override

```
static function void set_type_override (
    uvm_object_wrapper override_type,
    bit replace = 1
)
```

This is a pass-through; it configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type *T*, represented by this proxy, provided no instance override applies. The original type *T* shall be a super class of the *override_type*. *replace* is a pass-through to the **set_type_override_by_type** (see 8.3.1.4.2). The default value of *replace* shall be `1`.

8.2.3.2.6 set_inst_override

```
static function void set_inst_override(
    uvm_object_wrapper override_type,
    string inst_path,
    uvm_component parent = null
)
```

Configures the factory to create a component of the type represented by *override_type* whenever a request is made to create an object of the type *T*, represented by this proxy, with matching instance paths. The original type *T* shall be a super class of the *override_type*.

If *parent* is not specified, *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be specified outside the component classes. If *parent* is specified, *inst_path* is interpreted as being relative to the *parent*'s hierarchical instance path, i.e., `{parent.get_full_name(),".", inst_path}` is the instance path that is registered with the override. *inst_path* may contain wildcards for matching against multiple contexts.

8.2.3.2.7 initialize

```
virtual function void initialize()
```

Registers this proxy object with the current factory (see F.4.1.4.2) via **uvm_factory::register** (see 8.3.1.3).

8.2.4 uvm_object_registry #(T,Tname)

The **uvm_object_registry** serves as a lightweight proxy for an **uvm_object** (see 5.3) of type *T* and type name *Tname*, a string. The proxy enables efficient registration with the **uvm_factory** (see 8.3.1). Without it, registration would require an instance of the object itself.

8.2.4.1 Class declaration

```
class uvm_object_registry #(
    type T = uvm_object,
    string Tname = "<unknown>"
) extends uvm_object_wrapper
```

The default value of *Tname* shall be `"<unknown>"`.

8.2.4.2 Methods

8.2.4.2.1 create_object

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

Creates a component of type *T* and returns it as a handle to **uvm_object** (see 5.3). This is called by the factory after determining the type of object to create and the user can then implement it.

NOTE—Users should not call this method directly, they should use **create** instead (see 8.2.4.2.4).

8.2.4.2.2 get_type_name

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

Returns the value given by the string parameter *Tname* by default. This method overrides the method in **uvm_object_wrapper** (see 8.3.2).

8.2.4.2.3 get

```
static function uvm_object_registry #(T,Tname) get()
```

Returns a singleton instance.

The singleton instance is registered for initialization via **uvm_init** (see F.3.1.3) during static initialization. If **uvm_init** has been called prior to this registration occurring, the instance's **initialize** method (see 8.2.3.2.7) is called automatically during static initialization.

8.2.4.2.4 create

```
static function T create(
    string name = "",
    uvm_component parent = null,
    string contxt = ""
)
```

Returns an instance of the object type *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent*'s full name. The *contxt* argument, if supplied, supersedes the *parent*'s context. The new instance uses the given leaf *name* and *parent*.

8.2.4.2.5 set_type_override

```
static function void set_type_override (
    uvm_object_wrapper override_type,
    bit replace = 1
)
```

Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, provided no instance override applies. The original type *T* is typically a super class of the *override_type*. The default value of *replace* shall be `1`.

8.2.4.2.6 set_inst_override

```
static function void set_inst_override(
   uvm_object_wrapper override_type,
   string inst_path,
   uvm_component parent = null
)
```

Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, with matching instance paths. The original type *T* is typically a super class of the *override_type*.

If *parent* is not specified, *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be specified outside the component classes. If *parent* is specified, *inst_path* is interpreted as being relative to the *parent*'s hierarchical instance path, i.e., `{parent.get_full_name(),".", inst_path}` is the instance path that is registered with the override. *inst_path* may contain wildcards for matching against multiple contexts.

8.2.4.2.7 initialize

```
virtual function void initialize()
```

Registers this proxy object with the current factory (see F.4.1.4.2) via **uvm_factory::register** (see 8.3.1.3).

8.2.5 Abstract registries

UVM additionally supports registration of abstract objects and components with the factory. Since registered classes are abstract, they can not be constructed directly via a call to `new`. As such, the user needs to provide a factory override for any abstract classes that are registered with the factory. It shall be an error to attempt to construct an abstract class for which no overrides have been declared.

The abstract registries should only be used with objects and components that have been declared as virtual types, e.g.,

```
virtual my_component_base extends uvm_component
```

For standard components and objects (i.e., those not declared using the keyword `virtual`), the standard registries should be used (see 8.2.3 and 8.2.4).

8.2.5.1 uvm_abstract_component_registry

This serves as a lightweight proxy for an abstract component of type *T* and type name *Tname*, a string. The proxy enables efficient registration with **uvm_factory** (see 8.3.1). Without it, registration would require an instance of the component itself.

8.2.5.1.1 Class declaration

```
class uvm_abstract_component_registry #(type T=uvm_component,
   string Tname="<unknown>")
   extends uvm_object_wrapper
```

The default value of the parameter *Tname* shall be `"<unknown>"`.

This class has the following *Methods*.

8.2.5.1.2 create_component

```
virtual function uvm_component create_component(
    string name,
    uvm_component parent
)
```

As abstract classes cannot be constructed, this method shall generate an error and return *null*.

8.2.5.1.3 get_type_name

This is the same as **uvm_component_registry::get_type_name** (see 8.2.3.2.2).

8.2.5.1.4 get

This is the same as **uvm_component_registry::get** (see 8.2.3.2.3).

8.2.5.1.5 create

This is the same as **uvm_component_registry::create** (see 8.2.3.2.4).

8.2.5.1.6 set_type_override

This is the same as **uvm_component_registry::set_type_override** (see 8.2.3.2.5).

8.2.5.1.7 set_inst_override

This is the same as **uvm_component_registry::set_inst_override** (see 8.2.3.2.6).

8.2.5.1.8 initialize

This is the same as **uvm_component_registry::initialize** (see 8.2.3.2.7).

8.2.5.2 uvm_abstract_object_registry

This serves as a lightweight proxy for an abstract object of type *T* and type name *Tname*, a string. The proxy enables efficient registration with **uvm_factory** (see 8.3.1). Without it, registration would require an instance of the object itself.

8.2.5.2.1 Class declaration

```
class uvm_abstract_object_registry #(type T=uvm_object,
    string Tname="<unknown>")
    extends uvm_object_wrapper
```

The default value of the parameter *Tname* shall be `"<unknown>"`.

This class has the following *Methods*.

8.2.5.2.2 create_object

```
virtual function uvm_object create_object(
    string name,
    uvm_object parent
)
```

As abstract classes cannot be constructed, this method shall generate an error and return *null*.

8.2.5.2.3 get_type_name

This is the same as **uvm_object_registry::get_type_name** (see 8.2.4.2.2).

8.2.5.2.4 get

This is the same as **uvm_object_registry::get** (see 8.2.4.2.3).

8.2.5.2.5 create

This is the same as **uvm_object_registry::create** (see 8.2.4.2.4).

8.2.5.2.6 set_type_override

This is the same as **uvm_object_registry::set_type_override** (see 8.2.4.2.5).

8.2.5.2.7 set_inst_override

This is the same as **uvm_object_registry::set_inst_override** (see 8.2.4.2.2).

8.2.5.2.8 initialize

This is the same as **uvm_object_registry::initialize** (see 8.2.4.2.7).

8.3 UVM factory

This subclause covers the classes that define the UVM factory facility.

8.3.1 uvm_factory

As the name implies, **uvm_factory** is used to manufacture (create) UVM objects and components. Object and component types are registered with the factory using lightweight proxies to the actual objects and components being created. The **uvm_object_registry #(T,Tname)** (see 8.2.4) and **uvm_component_registry #(T,Tname)** (see 8.2.3) classes are used to proxy **uvm_object**s (see 5.3) and **uvm_component**s (see 13.1), respectively.

The factory provides both name-based and type-based interfaces.

> *type-based*—These interfaces are far less prone to typographical errors in usage. When errors do occur, they are caught at compile-time.

name-based—These interfaces are dominated by string arguments that can be misspelled and provided in the wrong order. Errors in name-based requests might only be caught at the time of the call, if at all. That being said, a name-based factory is required when crossing language boundaries, such as direct programming interface (DPI) or the command line.

A **uvm_factory** is an abstract class that declares many of its methods as `pure virtual`.

See 8.3.1.8 for details on configuring and using the factory.

8.3.1.1 Class declaration

```
virtual class uvm_factory
```

8.3.1.2 Retrieving the factory

8.3.1.2.1 get

```
static function uvm_factory get()
```

This is the static accessor for **uvm_factory**.

The static accessor is provided as a convenience wrapper around retrieving the factory via the **uvm_coreservice_t::get_factory** method (see F.4.1.4.2).

8.3.1.2.2 set

```
static function void set (
    uvm_factory f
)
```

Sets the factory instance to be *f*. This is a convenience wrapper around setting the factory via the **uvm_coreservice_t::set_factory** method (see F.4.1.4.3).

8.3.1.3 Registering types

register

```
pure virtual function void register (
    uvm_object_wrapper obj
)
```

Intended to register the given proxy object, *obj*, with the factory. The proxy object is a lightweight substitute for the component or object it represents. When the factory needs to create an object of a given type, it calls the proxy's **create_object** (see 8.3.2.2.1) or **create_component** (see 8.3.2.2.1) method to do so.

When doing name-based operations, the factory calls the proxy's **get_type_name** method (see 8.2.3.2.2) to match against the *requested_type_name* argument in subsequent calls to **create_component_by_name** and **create_object_by_name** (see 8.3.1.5). If the proxy object's **get_type_name** method returns the *empty string* (**""**), name-based lookup is effectively disabled.

8.3.1.4 Type and instance overrides

8.3.1.4.1 set_inst_override_by_type and set_inst_override_by_name

```
pure virtual function void set_inst_override_by_type (
    uvm_object_wrapper original_type,
    uvm_object_wrapper override_type,
    string full_inst_path
)

pure virtual function void set_inst_override_by_name (
    string original_type_name,
    string override_type_name,
    string full_inst_path
)
```

These methods are intended to configure the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path*.

When overriding by type, *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When overriding by name, the *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the **create_*** methods with the same string and matching instance path produce the type represented by *override_type_name*, which needs to be preregistered with the factory.

The *full_inst_path* is matched against the concatenation of `{parent_inst_path, ".", name}` provided in future **create** requests. *full_inst_path* may include wildcards (`*` and `?`) such that a single instance override can be applied in multiple contexts. A `full_inst_path` of "`*`" is effectively a type override (see 8.3.1.4.2), as it matches all contexts.

When the factory processes instance overrides, the instance queue is processed in order of override registrations and the first override match prevails. Thus, more specific overrides should be registered first, followed by more general overrides.

8.3.1.4.2 set_type_override_by_type and set_type_override_by_name

```
pure virtual function void set_type_override_by_type (
    uvm_object_wrapper original_type,
    uvm_object_wrapper override_type,
    bit replace = 1
)

pure virtual function void set_type_override_by_name (
    string original_type_name,
    string override_type_name,
    bit replace = 1
)
```

These methods are intended to configure the factory to create an object of the override's type whenever a request is made to create an object of the original type, provided no instance override applies. The original type shall be a super class of the *override_type*.

When overriding by type, *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When overriding by name, the *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the **create_*** methods with the same string and matching instance path produce the type represented by *override_type_name*, which needs to be preregistered with the factory.

When *replace* is `1`, a previous override on *original_type_name* is replaced; otherwise, a previous override, if any, remains intact. The default value of *replace* shall be `1`.

8.3.1.5 Creation

create_object_by_type, **create_component_by_type**, **create_object_by_name**, and **create_component_by_name**

```
pure virtual function uvm_object create_object_by_type (
    uvm_object_wrapper requested_type,
```

```
    string parent_inst_path = "",
    string name = ""
)

pure virtual function uvm_component create_component_by_type (
    uvm_object_wrapper requested_type,
    string name,
    uvm_component parent
)

pure virtual function uvm_object create_object_by_name (
    string requested_type_name,
    string parent_inst_path = "",
    string name = ""
)

pure virtual function uvm_component create_component_by_name (
    string requested_type_name,
    string parent_inst_path = "",
    string name,
    uvm_component parent
)
```

These methods are intended to create and return a component or object of the requested type, which may be specified by type or by name. A requested component shall be derived from the **uvm_component** base class (see 13.1), and a requested object shall be derived from the **uvm_object** base class (see 5.3).

When requesting by type, *requested_type* is a handle to the type's proxy object. Preregistration is not required.

When requesting by name, *request_type_name* is a string representing the requested type, which shall have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error shall be generated and a *null* handle returned.

If the optional *parent_inst_path* is provided, the concatenation `{parent_inst_path,".",name}` forms an instance path (context) that is used to search for an instance override. *parent_inst_path* is typically obtained by calling **uvm_component::get_full_name** (see 13.1.3.2) on the parent.

If no instance override is found, the factory then searches for a type override.

Once the final override is found, an instance of that component or object is returned in place of the requested type. New components use the given *name* and *parent*. New objects use the given *name*, if provided.

Override searches are recursively applied, with instance overrides taking precedence over type overrides. If `foo` overrides `bar`, and `xyz` overrides `foo`, then a request for `bar` produces `xyz`. Recursive loops result in an error, in which case the type returned is the one that formed the loop. Using the previous example, if `bar` overrides `xyz`, then `bar` is returned after the error is generated.

8.3.1.6 Name aliases

8.3.1.6.1 set_type_alias

```
pure virtual function void set_type_alias(string alias_type_name,
    uvm_object_wrapper original_type)
```

Intended to allow overrides by type to use the *alias_type_name* as an additional name to refer to *original_type*.

8.3.1.6.2 set_inst_alias

```
pure virtual function void set_inst_alias(string alias_type_name,
   uvm_object_wrapper original_type, string full_inst_path)
```

Intended to allow overrides by name to use the *alias_type_name* as an additional name to refer to *original_type* in the context referred to by *full_inst_path*.

8.3.1.7 Introspection

8.3.1.7.1 find_override_by_type and find_override_by_name

```
pure virtual function uvm_object_wrapper find_override_by_type (
   uvm_object_wrapper requested_type,
   string full_inst_path
)

pure virtual function uvm_object_wrapper find_override_by_name (
   string requested_type_name,
   string full_inst_path
)
```

These methods are intended to return the proxy to the object that would be created given the arguments. *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created, i.e.,
`{ parent.get_full_name(), ".", name }`.

8.3.1.7.2 find_wrapper_by_name

```
pure virtual function uvm_object_wrapper find_wrapper_by_name (
   string type_name
)
```

Intended to return the **uvm_object_wrapper** (see 8.3.2) associated with a given *type_name*.

8.3.1.7.3 is_type_name_registered

```
virtual function bit is_type_name_registered (string type_name)
```

This method checks if the given *type_name* was registered in the factory as the name for a type and returns `1` in this case.

8.3.1.7.4 is_type_registered

```
virtual function bit is_type_registered (uvm_object_wrapper obj)
```

This method checks if the given **uvm_object_wrapper** (see 8.3.2) *obj* was registered in the factory for a type and returns `1` in this case.

8.3.1.7.5 print

```
pure virtual function void print (
    int all_types = 1
)
```

Intended to print the state of the **uvm_factory**, including registered types, instance overrides, and type overrides.

When `all_types` is `0`, only type and instance overrides are displayed. When `all_types` is `1` (the default), all registered user-defined types are printed as well, provided they have names associated with them. When `all_types` is `2`, any UVM types (prefixed with `uvm_`) are included in the list of registered types.

8.3.1.8 Usage

Using the factory involves the following three basic operations:

a) Registering objects and components types with the factory.
b) Designing components to use the factory to create objects or components.
c) Configuring the factory with type and instance overrides, both within and outside components.

More reference information can be found in B.2.1, 8.2.3, 8.2.4, and 13.1.

8.3.2 uvm_object_wrapper

The **uvm_object_wrapper** provides an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every **uvm_object**-based (see 5.3) and **uvm_component**-based object (see 13.1) available in the test environment, are registered with the **uvm_factory** (see 8.3.1). When the factory is called upon to create an object or component, it finds and delegates the request to the appropriate proxy.

8.3.2.1 Class declaration

```
virtual class uvm_object_wrapper
```

8.3.2.2 Methods

8.3.2.2.1 create_object

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

Creates a new object with the optional *name*. An object proxy (see 8.2.4) implements this method to create an object of a specific type *T*.

8.3.2.2.2 create_component

```
virtual function uvm_component create_component (
    string name,
    uvm_component parent
)
```

Creates new component, passing to its constructor the given *name* and *parent*. A component proxy (see 8.2.3) implements this method to create an object of a specific type *T*.

8.3.2.2.3 get_type_name

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

Derived classes implement this method to return the type name of the object created by **create_component** (see 8.3.2.2.2) or **create_object** (see 8.3.2.2.1).

8.3.3 uvm_default_factory

Default implementation of the UVM factory.

Class declaration

```
class uvm_default_factory extends uvm_factory
```

uvm_default_factory can be extended (from **uvm_factory**); it provides a full implementation of all **uvm_factory**'s methods (see 8.3.1).

9. Phasing

9.1 Overview

UVM implements an automated mechanism for phasing the execution of the various components in a testbench.

9.2 Implementation

The API described here provides a general purpose testbench phasing solution, consisting of a phaser machine that traverses a master schedule graph. This machine is built by the integrator from one or more instances of template schedules provided by UVM or by third-party verification intellectual property (VIP), and it supports implicit or explicit synchronization, run-time control of threads, and jumps.

Each schedule node refers to a single phase compatible with that VIP's components and that executes the required behavior via an `IMP` (see F.2.5.1).

9.2.1 Class hierarchy

A single class represents the definition, state, and context of a phase. It is instantiated once as a singleton `IMP` (see F.2.5.1) and one or more times as nodes in a graph that represent serial and parallel phase relationships and store the current state as the phaser progresses, and also as the phase implementation that specifies required component behavior (by extension into the component context if non-default behavior required.).

9.2.2 Phasing related classes

The following classes are part of phasing:

a) **uvm_phase**—The base class for defining a phase's behavior, state, and context. See 9.3.1.
b) **uvm_domain**—A phasing schedule node representing an independent branch of the schedule. See 9.4.
c) **uvm_bottomup_phase**—A phase implementation for bottom-up function phases. See 9.5.
d) **uvm_topdown_phase**—A phase implementation for top-down function phases. See 9.7.
e) **uvm_task_phase**—A phase implementation for task phases. See 9.6.

9.2.3 Common and run-time phases

— The common phases to all **uvm_component**s (see 13.1) are described in 9.8.
— The run-time phases are described in 9.8.2.

9.3 Phasing definition classes

The following class are used to specify a phase and its implied functionality.

9.3.1 uvm_phase

This base class defines everything about a phase: its behavior, state, and context.

To define behavior, UVM or the user extends it to create singleton objects that capture the definition of what the phase does and how it does it. These are then cloned to produce multiple nodes that are hooked up in a

graph structure to provide context—which phases follow which—and to hold the state of the phase throughout its lifetime.

UVM provides default extensions of this class for the standard run-time phases.

NOTE—Users may likewise extend this class to define the phase proxy for a particular component context as required.

9.3.1.1 Phase definition, context, and state

9.3.1.1.1 Phase definition

To create custom phases, use one of the three predefined extended classes that encapsulate behavior for different phase types: **uvm_task_phase** (see 9.6), **uvm_bottomup_phase** (see 9.5), and **uvm_topdown_phase** (see 9.7).

a) Extend one of these classes as appropriate to create a `uvm_`*`YOURNAME`*`_phase` class (or *`YOURPREFIX_NAME`*`_phase` class) for each phase; this new class contains the default implementation of the new phase, is a **uvm_component**-compatible delegate (see 13.1), and may be a *null* implementation.

b) Instantiate a singleton instance of that class for user-code to use when a phase handle is required.

c) If this custom phase depends on methods that are not in **uvm_component**, but are within an extended class, then extend the base *`YOURPREFIX_NAME`*`_phase` class with parameterized component class context as required to create a specialized proxy that calls the user-defined extended component class methods.

This scheme ensures compile safety for any user-defined extended component classes while providing homogeneous base types for APIs and underlying data structures.

9.3.1.1.2 Phase context

A *schedule* is a coherent group of one or mode phase/state nodes linked together by a graph structure, allowing arbitrary linear/parallel relationships to be specified, and executed by stepping through them in the graph order. Each schedule node points to a phase, holds the execution state of that phase, and has optional links to other nodes for synchronization.

The main operations are: construct, add phases, and instantiate hierarchically within another schedule.

Each graph structure is a directed acyclic graph (DAG). Each instance is a node connected to others to form the graph. Each node in the graph has zero or more successors, and zero or more predecessors. No nodes are completely isolated from others. Exactly one node has zero predecessors. This is the *root node*.

Also, since the graph is acyclic, following the forward arrows never lead back to the starting point for any nodes in the graph; but, eventually this leads to a node with no successors.

9.3.1.1.3 Phase state

A given phase may appear multiple times in the complete phase graph, due to the multiple independent domain feature and the ability for different VIP to customize their own phase schedules (perhaps reusing existing phases). Each node instance in the graph maintains its own state of execution.

Phase state is represented by a value of **uvm_phase_state** (see F.2.5.2). A phase object that is not a schedule or a node within a schedule has the phase state value `UVM_PHASE_UNINITIALIZED`. Other phase objects

may progress through the states in the order shown for **uvm_phase_state**, with the exception of responding to a jump (see F.2.5.2).

UVM_PHASE_ENDED transitions to UVM_PHASE_CLEANUP if no jump or UVM_PHASE_JUMPING if there is a jump (see F.2.5.2). Each transition of phase state triggers a callback (see 9.3.3).

9.3.1.2 Class declaration

```
class uvm_phase extends uvm_object
```

9.3.1.3 Methods

9.3.1.3.1 new

```
function new(
   string name = "uvm_phase",
   uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
   uvm_phase parent = null
)
```

Creates a new phase node, with a *name* and *phase_type* (one of UVM_PHASE_IMP, UVM_PHASE_NODE, UVM_PHASE_SCHEDULE, or UVM_PHASE_DOMAIN). The default value of *phase_type* shall be UVM_PHASE_SCHEDULE.

9.3.1.3.2 get_phase_type

```
function uvm_phase_type get_phase_type()
```

Returns the phase type as defined by **uvm_phase_type** (see F.2.5.1).

9.3.1.3.3 set_max_ready_to_end_iterations

```
virtual function void set_max_ready_to_end_iterations(int max)
```

Sets the maximum number of iterations of ready_to_end. A raise and drop of objection while this phase is in **phase_ready_to_end** causes a new iteration of **phase_ready_to_end** if the new iteration count is less than this value (see 13.1.4.3.2). The default value is the value returned from **get_max_ready_to_end_iterations** (see 9.3.1.3.6).

9.3.1.3.4 get_max_ready_to_end_iterations

```
virtual function int get_max_ready_to_end_iterations()
```

Returns the maximum number of iterations of ready_to_end (see 9.3.1.3.3).

9.3.1.3.5 set_default_max_ready_to_end_iterations

```
static function void set_default_max_ready_to_end_iterations(int max)
```

Sets the global default maximum number of iterations of **phase_ready_to_end** (see 9.3.1.3.3). The default value is 20.

9.3.1.3.6 get_default_max_ready_to_end_iterations

```
static function int get_max_ready_to_end_iterations()
```

Returns the default maximum number of iterations of `ready_to_end` (see 9.3.1.3.5).

9.3.1.4 State

9.3.1.4.1 get_state

```
function uvm_phase_state get_state()
```

This is an accessor to return the current state of this phase.

9.3.1.4.2 get_run_count

```
function int get_run_count()
```

This is an accessor to return the integer number of times this phase has executed.

9.3.1.4.3 find_by_name

```
function uvm_phase find_by_name(
   string name,
   bit stay_in_scope = 1
)
```

Locates a phase node with the specified name and returns its handle. When *stay_in_scope* is set to `1`, this only searches within this phase's schedule and domain. The default value of *stay_in_scope* shall be `1`.

9.3.1.4.4 find

```
function uvm_phase find(
   uvm_phase phase,
   bit stay_in_scope = 1
)
```

Locates the phase node with the specified phase `IMP` and returns its handle. When *stay_in_scope* is set to `1`, this only searches within this phase's schedule and domain. The default value of *stay_in_scope* shall be `1`.

9.3.1.4.5 is

```
function bit is(
   uvm_phase phase
)
```

Returns `1` if the containing **uvm_phase** refers to the same phase as the *phase* argument, `0` otherwise.

9.3.1.4.6 is_before

```
function bit is_before(
   uvm_phase phase
)
```

Returns `1` if the containing **uvm_phase** refers to a phase that is earlier than the *phase* argument, `0` otherwise.

9.3.1.4.7 is_after

```
function bit is_after(
   uvm_phase phase
)
```

Returns `1` if the containing **uvm_phase** refers to a phase that is later than the *phase* argument, `0` otherwise.

9.3.1.5 Callbacks

9.3.1.5.1 exec_func

```
virtual function void exec_func(
   uvm_component comp,
   uvm_phase phase
)
```

Implements the proxy functionality for a function phase type *comp*—the component to execute the functionality upon *phase*—the phase schedule that originated this phase call.

9.3.1.5.2 exec_task

```
virtual task exec_task(
   uvm_component comp,
   uvm_phase phase
)
```

Implements the proxy functionality for a task phase type *comp*—the component to execute the functionality upon *phase*—the phase schedule that originated this phase call.

9.3.1.6 Schedule

9.3.1.6.1 add

```
function void add(
   uvm_phase phase,
   uvm_phase with_phase = null,
   uvm_phase after_phase = null,
   uvm_phase before_phase = null,
   uvm_phase start_with_phase = null,
   uvm_phase end_with_phase = null
)
```

Adds *phase* to the schedule or domain. **add** shall be called only from a phase with the type `UVM_PHASE_SCHEDULE` or `UVM_PHASE_DOMAIN`. Optionally, one or more phases may be specified to specify how the new phase aligns with existing phases. The optional phases shall already exist in the schedule or domain. If no optional phases are specified, *phase* is appended to the schedule or domain.

If *with_phase* is not *null*, *phase* is added in parallel with *with_phase*. If *after_phase* is not *null*, *phase* is added as a successor to *after_phase*. If *before_phase* is not *null*, *phase* is added as a predecessor to *before_phase*. If *start_with_phase* is not *null*, *phase* is added as a successor to the predecessor(s) of *start_with_phase*. If *end_with_phase* is not *null*, *phase* is added as a predecessor to the successor(s) of *end_with_phase*. *with_phase*, *after_phase*, and *start_with_phase* specify the predecessor of *phase*; only one of these shall be non-*null*. *with_phase*, *before_phase*, and *end_with_phase* specify the successor of *phase*; only one of these shall be non-*null*.

9.3.1.6.2 get_parent

```
function uvm_phase get_parent()
```

Returns the parent schedule node, if any, for the hierarchical graph traversal.

9.3.1.6.3 get_full_name

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

Returns the full path from the enclosing domain down to this node. The singleton `IMP` phases have no hierarchy.

9.3.1.6.4 get_schedule

```
function string get_schedule(
   bit hier = 0
)
```

Returns the topmost parent schedule node, if any, for the hierarchical graph traversal. The default value of *hier* shall be `0`.

9.3.1.6.5 get_schedule_name

```
function string get_schedule_name(
   bit hier = 0
)
```

Returns the schedule name associated with this phase node. An implementation calls **get_schedule (***hier***)** (see 9.3.1.6.4) and then constructs a hierarchical name including any schedule names above the returned schedule. The default value of *hier* shall be `0`.

9.3.1.6.6 get_domain

```
function uvm_domain get_domain()
```

Returns the enclosing domain or *null* if there is none.

9.3.1.6.7 get_imp

```
function uvm_phase get_imp()
```

Returns the phase implementation for this node. Returns *null* if this phase type is not a `UVM_PHASE_IMP`.

9.3.1.6.8 get_domain_name

```
function string get_domain_name()
```

Returns the domain name associated with this phase node or `"unknown"` if no domain found.

9.3.1.6.9 get_adjacent_predecessor_nodes

```
function void get_adjacent_predecessor_nodes(
   ref uvm_phase pred[]
)
```

Provides an array of nodes that are predecessors to *this* phase node. A predecessor node is defined as any phase node that lies prior to *this* node in the phase graph; an adjacent predecessor node has no nodes between *this* node and the predecessor node.

9.3.1.6.10 get_adjacent_successor_nodes

```
function void get_adjacent_successor_nodes(
   ref uvm_phase pred[]
)
```

Provides an array of nodes that are successors to *this* phase node. A successor node is defined as any phase node that lies after to *this* node in the phase graph, with no nodes between *this* node and the successor node.

9.3.1.7 Phase done objections

Task-based phase nodes within the phasing graph provide a **uvm_objection** based interface (see 10.5.1) for prolonging the execution of the phase. All other phase types do not contain an objection and shall report an error if the user attempts to use **raise_objection** (see 9.3.1.7.2), **drop_objection** (see 9.3.1.7.3), or **get_objection_count** (see 9.3.1.7.4).

9.3.1.7.1 get_objection

```
function uvm_objection get_objection()
```

Returns the **uvm_objection** (see 10.5.1) that gates the termination of the phase.

9.3.1.7.2 raise_objection

```
virtual function void raise_objection (
   uvm_object obj,
   string description = "",
   int count = 1
)
```

Raises an objection to ending this phase, which provides components with greater control over the phase flow for processes that are not implicit objectors to the phase. The default value of *count* shall be `1`. For more details, refer to the **uvm_objection** version of this function (see 10.5.1.3.3).

9.3.1.7.3 drop_objection

```
virtual function void drop_objection (
   uvm_object obj,
   string description = "",
   int count = 1
)
```

Drops an objection to ending this phase. The default value of *count* shall be `1`. For more details, refer to the **uvm_objection** version of this function (see 10.5.1.3.4).

9.3.1.7.4 get_objection_count

```
virtual function int get_objection_count(
   uvm_object obj = null
)
```

This is a pass through to the *get_objection_count* on the objection returned by **get_objection**. See 10.5.1.5.3.

9.3.1.8 Synchronization

The functions **sync** (see 9.3.1.8.1) and **unsync** (see 9.3.1.8.2) add relationships between nodes, such that the node's start and end are synchronized.

9.3.1.8.1 sync

```
function void sync(
    uvm_domain target,
    uvm_phase phase = null,
    uvm_phase with_phase = null
)
```

Synchronizes two domains, fully or partially.

a) *target*—handle of target domain for synchronizing this one.
b) *phase*—optional single phase in this domain to synchronize; otherwise, `sync all`.
c) *with_phase*—optional different target-domain phase with which to synchronize; otherwise, use *phase* in the target domain.

9.3.1.8.2 unsync

```
function void unsync(
    uvm_domain target,
    uvm_phase phase = null,
    uvm_phase with_phase = null
)
```

Removes synchronization between two domains, fully or partially.

a) *target*—handle of target domain from which to remove synchronization.
b) *phase*—optional single phase in this domain to unsynchronize; otherwise, `unsync all`.
c) *with_phase*—optional different target-domain phase with which to unsynchronize; otherwise, use *phase* in the target domain.

9.3.1.8.3 wait_for_state

```
task wait_for_state(
    uvm_phase_state state,
    uvm_wait_op op = UVM_EQ
)
```

Waits until this phase compares with the given *state* and *op* operands.

To wait for the phase to be at the started state or afterward:

```
wait_for_state(UVM_PHASE_STARTED, UVM_GTE)
```

9.3.1.9 Jumping

Phase jumping refers to a change in the normal process of a phase ending and the successor phase(s) starting. A phase can be made to end prematurely and/or which phase is started next can be changed. To

jump all active phases within a domain that are predecessors or successors, directly or indirectly, of the jump target, use **uvm_domain::jump** (see 9.4.2.4). Phase jumping can also be specified for an individual phase instance by using the following functions.

9.3.1.9.1 jump

```
function void jump(
   uvm_phase phase
)
```

Jumps to the specified *phase*. The *phase* shall be in the set of predecessors or successors of the current phase. All active phases that share *phase* as a common successor or predecessor shall also be affected.

9.3.1.9.2 set_jump_phase

```
function void set_jump_phase(
   uvm_phase phase
)
```

Specifies which phase to transition to when this phase completes. Note that this function is part of what **jump** does (see 9.3.1.9.1); unlike **jump**, this does not set the flag to terminate the phase prematurely.

9.3.1.9.3 end_prematurely

```
function void end_prematurely()
```

Specifies a flag to cause the phase to end prematurely. Note that this function is part of what **jump** does (see 9.3.1.9.1); unlike **jump**, this does not set a `jump_phase` to go to after the phase ends.

9.3.1.9.4 get_jump_target

```
function uvm_phase get_jump_target()
```

Returns a handle to the target phase of the current jump or *null*, if no jump is in progress. It is valid from the time **jump** (see 9.3.1.9.1) or **set_jump_phase** (see 9.3.1.9.2) is called until the jump occurs. There is also a callback for `UVM_PHASE_JUMPING` that contains a valid return from this function.

9.3.2 uvm_phase_state_change

This is a phase state transition descriptor, which is used to describe the phase transition that caused a `uvm_phase_cb::state_changed` callback to be invoked.

9.3.2.1 Class declaration

```
class uvm_phase_state_change extends uvm_object
```

9.3.2.2 Methods

9.3.2.2.1 get_state

```
virtual function uvm_phase_state get_state()
```

Returns the state to which the phase just transitioned. This is functionally equivalent to **uvm_phase::get_state** (see 9.3.1.4.1).

9.3.2.2.2 get_prev_state

```
virtual function uvm_phase_state get_prev_state()
```

Returns the state from which the phase just transitioned.

9.3.2.2.3 jump_to

```
function uvm_phase jump_to()
```

If the current state is `UVM_PHASE_ENDED` or `UVM_PHASE_JUMPING` because of a phase jump, this returns the phase that is the target of jump. Otherwise, it returns *null*.

9.3.3 uvm_phase_cb

This class defines a callback method that is invoked by the phaser during the execution of a specific node in the phase graph or for all phase nodes. User-defined callback extensions can be used to integrate data types that are not natively phase-aware with the UVM phasing.

9.3.3.1 Class declaration

```
class uvm_phase_cb extends uvm_callback
```

9.3.3.2 Methods

9.3.3.2.1 new

```
function new(
   string name = "unnamed-uvm_phase_cb"
)
```

This is a constructor. The default value of *name* shall be `"unnamed-uvm_phase_cb"`.

9.3.3.2.2 phase_state_change

```
virtual function void phase_state_change(
   uvm_phase phase,
   uvm_phase_state_change change
)
```

Called whenever a *phase* changes state. The *change* descriptor describes the transition that was just completed. The callback method is invoked immediately after the phase state has changed, but before the phase implementation is executed.

An extension may interact with the phase, such as raising the phase objection to prolong the phase, in a manner that is consistent with the current phase state.

By default, this callback method does nothing. Unless otherwise specified, modifying the phase transition descriptor has no effect on the phasing schedule or execution.

9.4 uvm_domain

This is a phasing schedule node representing an independent branch of the schedule. It is a handle used to assign domains to components or hierarchies in the testbench.

9.4.1 Class declaration

```
class uvm_domain extends uvm_phase
```

9.4.2 Methods

9.4.2.1 new

```
function new(
   string name
)
```

Creates a new instance (`type = UVM_PHASE_DOMAIN`) of a phase domain. The new instance is added to the list of all domains indexed by *name*. It shall be an error to call **new** with a *name* that is already in the list of all domains.

9.4.2.2 get_domains

```
static function void get_domains(
   output uvm_domain domains[string]
)
```

Provides a list of all domains for the *domains* argument. The list of all domains always contains a `domains["common"]` entry that contains the UVM common phases (see 9.8) and a `domains["uvm"]` entry that contains the UVM run-time phases (see 9.8.2).

9.4.2.3 add_uvm_phases

```
static function void add_uvm_phases(
   uvm_phase schedule
)
```

Appends the built-in UVM phases to the given *schedule*.

9.4.2.4 jump

```
function void jump(
   uvm_phase phase
)
```

Jumps all active phases of this domain to *phase* if there is a path between the active phases of this domain and *phase*.

9.5 uvm_bottomup_phase

This is a virtual base class for function phases that operate bottom-up. The virtual function **execute** (see 9.5.2.3) is called for each component. A bottom-up phase invokes the delegate function first on components without any children; once finished, the delegate function is invoked on each of the first sets' parents, etc. A bottom-up function phase completes when the **execute** method has been called and returned on all applicable components in the hierarchy.

9.5.1 Class declaration

```
virtual class uvm_bottomup_phase extends uvm_phase
```

9.5.2 Methods

9.5.2.1 new

```
function new(
   string name
)
```

Initializes a new instance (`type = UVM_PHASE_IMP`) of a bottom-up phase.

9.5.2.2 traverse

```
virtual function void traverse(
   uvm_component comp,
   uvm_phase phase,
   uvm_phase_state state
)
```

Traverses the component tree in bottom-up order and, depending on the *state*, calls `comp.phase_started(phase)`, `execute(comp,phase)`, or `comp.phase_ended(phase)`.

9.5.2.3 execute

```
virtual function void execute(
   uvm_component comp,
   uvm_phase phase
)
```

Calls `uvm_phase::exec_func(comp,phase)`.

9.6 uvm_task_phase

This is the base class for all task phases. It forks a call to **uvm_phase::exec_task** (see 9.3.1.5.2) for each component in the hierarchy.

The completion of these tasks does not imply, nor is it required for, the end of phase. Once the phase completes, any remaining forked **uvm_phase::exec_task** threads are forcibly and immediately killed.

By default, the way for a task phase to extend over time is if there is at least one component that raises an objection, e.g.,

```
class my_comp extends uvm_component
   task main_phase(uvm_phase phase)
      phase.raise_objection(this, "Applying stimulus")
      ...
      phase.drop_objection(this, "Applied enough stimulus")
   endtask
endclass
```

There is however one scenario wherein time advances within a task-based phase without any objections to the phase being raised. If two (or more) phases are synched, or they share a common successor, such as the **uvm_run_phase** (see 9.8.1.5) and the **uvm_post_shutdown_phase** (see 9.8.2.12) sharing the **uvm_extract_phase** (see 9.8.1.6) as a successor, then phase advancement is delayed until all predecessors of the common successor are ready to proceed. Because of this, it is possible for time to advance between the

uvm_component::phase_started (see 13.1.4.3.1) and **uvm_component::phase_ended** (see 13.1.4.3.1) of a task phase without any participants in the phase raising an objection.

A task phase shall not share a successor with a `topdown_phase` or `bottomup_phase`, as that setup could try to make the `topdown_phase` or `bottomup_phase` consume time.

9.6.1 Class declaration

```
virtual class uvm_task_phase extends uvm_phase
```

9.6.2 Methods

9.6.2.1 new

```
function new(
   string name
)
```

Initializes a new instance (`type = UVM_PHASE_IMP`) of a task-based phase.

9.6.2.2 traverse

```
virtual function void traverse(
   uvm_component comp,
   uvm_phase phase,
   uvm_phase_state state
)
```

Traverses the component tree and, depending on the *state*, calls `comp.phase_started(phase)`, `execute(comp,phase)`, `comp.phase_ready_to_end (phase)`, or `comp.phase_ended (phase)`.

9.6.2.3 execute

```
virtual function void execute(
   uvm_component comp,
   uvm_phase phase
)
```

Forks `uvm_phase::exec_task(comp,phase)`.

9.7 uvm_topdown_phase

This is a virtual base class for function phases that operate top-down. The virtual function **execute** (see 9.7.2.3) is called for each component.

A top-down function phase completes when the **execute** method has been called and returned on all applicable components in the hierarchy.

9.7.1 Class declaration

```
virtual class uvm_topdown_phase extends uvm_phase
```

9.7.2 Methods

9.7.2.1 new

```
function new(
   string name
)
```

Initializes a new instance (`type = UVM_PHASE_IMP`) of a top-down phase.

9.7.2.2 traverse

```
virtual function void traverse(
   uvm_component comp,
   uvm_phase phase,
   uvm_phase_state state
)
```

Traverses the component tree in top-down order and, depending on the *state*, calls `comp.phase_started(phase)`, `execute(comp,phase)`, or `comp.phase_ended(phase)`.

9.7.2.3 execute

```
virtual function void execute(
   uvm_component comp,
   uvm_phase phase
)
```

Calls `uvm_phase::exec_func(comp,phase)`.

9.8 Predefined phases

UVM defines some phases. The user is free to create more phases.

The names of the UVM predefined phases (which are returned by **get_name** for a phase instance) match the class names specified in this subclause with the "`uvm_`" and "`_phase`" terms removed. Each UVM predefined phase implements the following *method*.

get

```
static function TYPE get()
```

which returns the singleton phase handle for each phase. The return value of **get** is of the same type as the phase itself, such that `uvm_build_phase::get` has a return type of `uvm_build_phase`, `uvm_run_phase` has a return type of `uvm_run_phase`, and so on.

The UVM predefined phases are classified as common phases and run-time phases. The *common phases* are the set of function and task phases that all **uvm_component**s (see 13.1) execute together. All **uvm_component**s are always synchronized with respect to the common phases. The *run-time phases* execute in a predefined phase schedule that runs concurrently to the common phase **uvm_run_phase** (see 9.8.1.5). By default, all **uvm_component**s (see 13.1) using the run-time schedule are synchronized with respect to the predefined phases in the schedule. It is possible for components to belong to different domains in which case their schedules can be unsynchronized with a call to unsync (see 9.3.1.8.2).

9.8.1 Common phases

The common phases are described in the order of their execution. All of the phases before **uvm_run_phase** (see 9.8.1.5) shall execute at simulation time `0`.

9.8.1.1 uvm_build_phase

```
class uvm_build_phase extends uvm_topdown_phase
```

This is a **uvm_topdown_phase** (see 9.7) whose `exec_func` calls the **uvm_component::build_phase** method (see 13.1.4.1.1). This phase gives components a defined time to create and configure the testbench.

9.8.1.2 uvm_connect_phase

```
class uvm_connect_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::connect_phase** method (see 13.1.4.1.2). This phase gives components a defined time to establish cross-component connections.

9.8.1.3 uvm_end_of_elaboration_phase

```
class uvm_end_of_elaboration_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::end_of_elaboration_phase** method (see 13.1.4.1.3).

9.8.1.4 uvm_start_of_simulation_phase

```
class uvm_start_of_simulation_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::start_of_simulation_phase** method (see 13.1.4.1.4).

9.8.1.5 uvm_run_phase

```
class uvm_run_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::run_phase** virtual method (see 13.1.4.1.5). This phase runs in parallel to the run-time phases, **uvm_pre_reset_phase** through **uvm_post_shutdown_phase** (see 9.8.2). The **uvm_run_phase** shall always be running when time is advancing, so when this phase starts, simulation time is still `0`, and the time when this phase ends shall be the same time that the **uvm_final_phase** ends (see 9.8.1.9).

The run phase starts a global timeout counter thread. The expiration time of the counter shall be implementation-specific, unless set via **uvm_root::set_timeout** (see F.7.2.3) prior to **uvm_run_phase** starting. If the counter expires before **uvm_run_phase** ends, it shall generate a fatal error.

9.8.1.6 uvm_extract_phase

```
class uvm_extract_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::extract_phase** method (see 13.1.4.1.6).

9.8.1.7 uvm_check_phase

```
class uvm_check_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::check_phase** method (see 13.1.4.1.7).

9.8.1.8 uvm_report_phase

```
class uvm_report_phase extends uvm_bottomup_phase
```

This is a **uvm_bottomup_phase** (see 9.5) whose `exec_func` calls the **uvm_component::report_phase** method (see 13.1.4.1.8).

9.8.1.9 uvm_final_phase

```
class uvm_final_phase extends uvm_topdown_phase
```

This is a **uvm_topdown_phase** (see 9.7) whose `exec_func` calls the **uvm_component::final_phase** method (see 13.1.4.1.9).

9.8.2 UVM run-time phases

The run-time phases shall include the following task phases, shown in their default order of execution. Users and implementations may add run-time phases before or after any of these specified phases. The run-time phases shall not start before the end of **uvm_start_of_simulation_phase** (see 9.8.1.4). **uvm_extract_phase** (see 9.8.1.6) shall not start before the run-time phases have ended. These specified phases shall not overlap within an instance of `uvm_domain`.

9.8.2.1 uvm_pre_reset_phase

```
class uvm_pre_reset_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::pre_reset_phase** method (see 13.1.4.2.1).

9.8.2.2 uvm_reset_phase

```
class uvm_reset_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::reset_phase** method (see 13.1.4.2.2).

9.8.2.3 uvm_post_reset_phase

```
class uvm_post_reset_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::post_reset_phase** method (see 13.1.4.2.3).

9.8.2.4 uvm_pre_configure_phase

```
class uvm_pre_configure_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::pre_configure_phase** method (see 13.1.4.2.4).

9.8.2.5 uvm_configure_phase

```
class uvm_configure_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::configure_phase** method (see 13.1.4.2.5).

9.8.2.6 uvm_post_configure_phase

```
class uvm_post_configure_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::post_configure_phase** method (see 13.1.4.2.6).

9.8.2.7 uvm_pre_main_phase

```
class uvm_pre_main_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::pre_main_phase** method (see 13.1.4.2.7).

9.8.2.8 uvm_main_phase

```
class uvm_main_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::main_phase** method (see 13.1.4.2.8).

9.8.2.9 uvm_post_main_phase

```
class uvm_post_main_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::post_main_phase** method (see 13.1.4.2.9).

9.8.2.10 uvm_pre_shutdown_phase

```
class uvm_pre_shutdown_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::pre_shutdown_phase** method (see 13.1.4.2.10).

9.8.2.11 uvm_shutdown_phase

```
class uvm_shutdown_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::shutdown_phase** method (see 13.1.4.2.11).

9.8.2.12 uvm_post_shutdown_phase

```
class uvm_post_shutdown_phase extends uvm_task_phase
```

This is a **uvm_task_phase** (see 9.6) whose `exec_task` calls the **uvm_component::post_shutdown_phase** method (see 13.1.4.2.12).

10. Synchronization classes

UVM provides event and barrier synchronization classes for managing concurrent processes, as follows:

a) **uvm_event#(T)**—UVM's event class (see 10.1.2) augments the SystemVerilog `event` data type with such services as setting callbacks and data delivery.

b) **uvm_barrier**—A barrier is used to prevent a pre-configured number of processes from continuing until all have reached a certain point in simulation (see 10.3).

c) **uvm_event_pool** and **uvm_barrier_pool**—The event and barrier pool classes are specializations of **uvm_pool #(string, T)** (see 11.2) used to store collections of **uvm_event**s (see 10.1) and **uvm_barrier**s (see 10.3), respectively, indexed by string name. Each pool class contains a static, "global" pool instance for sharing across all processes (see 10.4).

d) **uvm_event_callback**—The event callback is used to create callback objects that may be attached to **uvm_event**s (see 10.2).

10.1 Event classes

This subclause defines the **uvm_event_base** class (see 10.1.1) and its derivative **uvm_event#(T)** (see 10.1.2).

10.1.1 uvm_event_base

The **uvm_event_base** class is an abstract wrapper class around the SystemVerilog `event` construct. It provides some additional services such as setting callbacks and maintaining the number of waiters.

10.1.1.1 Class declaration

```
virtual uvm_event_base extends uvm_object
```

10.1.1.2 Methods

10.1.1.2.1 new

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

Creates a new event object.

10.1.1.2.2 wait_on

```
virtual task wait_on (
    bit delta = 0
)
```

Waits for the event to be activated for the first time.

If the event has already been triggered, this task returns immediately. If a `delta` value is specified, the caller is forced to wait a single delta `#0` before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume. The default value of *delta* shall be `0`.

Once an event has been triggered, it will be remain "on" until the event is **reset** (see 10.1.1.2.8).

10.1.1.2.3 wait_off

```
virtual task wait_off (
    bit delta = 0
)
```

If the event has already triggered and is “on,” this task waits for the event to be turned “off” via a call to **reset** (see 10.1.1.2.8).

If the event has not already been triggered, this task returns immediately. If `delta` value is specified, the caller is forced to wait a single delta `#0` before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume. The default value of *delta* shall be `0`.

10.1.1.2.4 wait_trigger

```
virtual task wait_trigger()
```

Waits for the event to be triggered.

If one process calls **wait_trigger** in the same delta as another process calls **uvm_event#(T)::trigger** (see 10.1.2.2.4), a race condition occurs. If the call to wait occurs before the trigger, this method returns in this delta. If the wait occurs after the trigger, this method does not return until the next trigger, which may never occur and, thus, cause a deadlock. This race can be avoided by using **wait_ptrigger** (see 10.1.1.2.5).

10.1.1.2.5 wait_ptrigger

```
virtual task wait_ptrigger()
```

Waits for a persistent trigger of the event. Unlike **wait_trigger** (see 10.1.1.2.4), this views the trigger as persistent within a given time-slice and, thus, avoids certain race conditions. If this method is called after the trigger, but within the same time-slice, the caller returns immediately.

10.1.1.2.6 get_trigger_time

```
virtual function time get_trigger_time()
```

Returns the time that this event was last triggered. If the event has not been triggered or the event has been reset, the trigger time is `0`.

10.1.1.2.7 is_on

```
virtual function bit is_on()
```

Indicates whether the event has been triggered since it was last reset.

A return of `1` indicates the event has triggered.

10.1.1.2.8 reset

```
virtual function void reset (
    bit wakeup = 0
)
```

Resets the event to its off state. If *wakeup* is set, all processes currently blocked waiting on **wait_trigger** (see 10.1.1.2.4) or **wait_ptrigger** (see 10.1.1.2.5) for the event are activated before the reset. The default value of *wakeup* shall be `0`.

No callbacks are called during a reset.

10.1.1.2.9 cancel

```
virtual function void cancel()
```

Decrements the number of waiters on the event.

This is used if a process that is waiting on an event is disabled or activated by some other means.

10.1.1.2.10 get_num_waiters

```
virtual function int get_num_waiters()
```

Returns the number of processes waiting on the event.

10.1.2 uvm_event#(T)

The **uvm_event** class is an extension of the abstract **uvm_event_base** class (see 10.1.1).

The optional parameter *T* allows the user to define a data type that can be passed during an event trigger.

10.1.2.1 Class declaration

```
class uvm_event#(
    type T = uvm_object
) extends uvm_event_base
```

10.1.2.2 Methods

10.1.2.2.1 new

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

Creates a new event object.

10.1.2.2.2 wait_trigger_data

```
virtual task wait_trigger_data (
    output T data
)
```

This method calls **uvm_event_base::wait_trigger** (see 10.1.1.2.4) followed by **get_trigger_data** (see 10.1.2.2.5).

10.1.2.2.3 wait_ptrigger_data

```
virtual task wait_ptrigger_data (
    output T data
)
```

This method calls **uvm_event_base::wait_ptrigger** (see 10.1.1.2.5) followed by **get_trigger_data** (see 10.1.2.2.5).

10.1.2.2.4 trigger

```
virtual function void trigger (
    T data = get_default_data
)
```

Triggers the event, resuming all waiting processes.

An optional *data* argument can be supplied with the enable to provide trigger-specific information. If no data is provided, then **get_trigger_data** (see 10.1.2.2.5) shall return the default data (see 10.1.2.2.6).

10.1.2.2.5 get_trigger_data

```
virtual function T get_trigger_data()
```

Returns the data, if any, provided by the last call to **trigger** (see 10.1.2.2.4).

10.1.2.2.6 default data

```
virtual function void set_default_data (T data)
virtual function T get_default_data()
```

Default trigger data to be used when **trigger** (see 10.1.2.2.4) is called without passing in data. `get_default_data` shall return the most recent *data* assigned via `set_default_data`. The value returned by `get_default_data` prior to calling `set_default_data` is the uninitialized value of type *T*.

10.2 uvm_event_callback

The **uvm_event_callback** class is an abstract class that is used to create callback objects that may be attached to **uvm_event#(T)**s (see 10.1.2). To do so, simply derive a new class and override **pre_trigger** (see 10.2.2.2) and/or **post_trigger** (see 10.2.2.3).

Callbacks are an alternative to using processes that wait on events. When a callback is attached to an event, that callback object's callback function is called each time the event is triggered.

10.2.1 Class declaration

```
virtual class uvm_event_callback#(
    type T = uvm_object
) extends uvm_callback
```

10.2.2 Methods

10.2.2.1 new

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

Initializes a new callback object.

10.2.2.2 pre_trigger

```
virtual function bit pre_trigger (
   uvm_event#(T) e,
   T data
)
```

This callback is called just before triggering the associated event. In a derived class, override this method to implement any pre-trigger functionality.

If the callback returns `1`, the event does not trigger and the post-trigger callback is not called. This provides a way for a callback to prevent the event from triggering.

In this function, *e* is the **uvm_event#(T)** (see 10.1.2) that is being triggered and *data* is the optional data associated with the event trigger.

10.2.2.3 post_trigger

```
virtual function void post_trigger (
   uvm_event#(T) e,
   T data
)
```

This callback is called after triggering the associated event. In a derived class, override this method to implement any post-trigger functionality.

In this function, *e* is the **uvm_event#(T)** (see 10.1.2) that is being triggered and *data* is the optional data associated with the event trigger.

10.3 uvm_barrier

The **uvm_barrier** class provides a multi-process synchronization mechanism. It enables a set of processes to block until the desired number of processes reach the synchronization point, at which time all of the processes are released.

10.3.1 Class declaration

```
class uvm_barrier extends uvm_object
```

10.3.2 Methods

10.3.2.1 new

```
function new (
   string name = "",
   int threshold = 0
)
```

Creates a new barrier object. The default value of *threshold* shall be `0`.

10.3.2.2 wait_for

```
virtual task wait_for()
```

Blocks until the number of blocked **wait_for** calls matches the current threshold.

The number of processes to wait for can be specified by using the **set_threshold** method (see 10.3.2.6).

10.3.2.3 reset

```
virtual function void reset (
   bit wakeup = 1
)
```

Resets the barrier. This sets the waiter count back to zero (`0`).

The threshold is unchanged. After reset, the barrier forces any processes to wait for the threshold again.

If the *wakeup* bit is set to `1`, any currently waiting processes shall be activated. The default value of *wakeup* shall be `1`.

10.3.2.4 set_auto_reset

```
virtual function void set_auto_reset (
   bit value = 1
)
```

Determines if the barrier should reset itself after the threshold is reached.

The default is on, so when a barrier hits its threshold it resets and new processes block until the threshold is reached again.

If auto reset is off, then once the threshold is achieved, new processes pass through without being blocked until the barrier is reset. The default value of *value* shall be `1`.

10.3.2.5 get_threshold

```
virtual function int get_threshold()
```

Returns the current threshold setting for the barrier.

10.3.2.6 set_threshold

```
virtual function void set_threshold (
   int threshold
)
```

Specifies the process threshold.

This determines how many processes are waiting on the barrier before the processes may proceed. Once the *threshold* is reached, all waiting processes are activated.

If *threshold* is set to a value less than the number of currently waiting processes, the barrier is reset and all waiting processes are activated.

10.3.2.7 get_num_waiters

```
virtual function int get_num_waiters()
```

Returns the number of processes currently waiting at the barrier.

10.3.2.8 cancel

```
virtual function void cancel()
```

Decrements the waiter count by one. This is used when a process that is waiting on the barrier is killed or activated by some other means.

10.4 Pool classes

10.4.1 uvm_event_pool

An object used to store collections of **uvm_event**s (see 10.1).

By default, the event pool contains the events: begin, accept, and end. Events can also be added by derivative objects. An event pool is a specialization of an **uvm_pool #(KEY,T)** (see 11.2), e.g., a `uvm_pool #(uvm_event)`.

10.4.1.1 Class declaration

```
class uvm_event_pool extends uvm_pool #(string,uvm_event#(uvm_object))
```

10.4.1.2 Common methods

10.4.1.2.1 new

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

Creates a new event pool object.

10.4.1.2.2 get_global_pool

```
static function uvm_event_pool get_global_pool()
```

Returns the singleton global event pool.

10.4.1.2.3 get_global

```
static function uvm_event_pool get_global (
   string key
)
```

Returns the item instance specified by *key* from the global item pool.

10.4.1.2.4 get

```
virtual function uvm_event get (
   string key
)
```

Returns the item with the given string *key*.

If no item exists by that *key*, a new item is created with that *key* and returned.

10.4.2 uvm_barrier_pool

An object used to store collections of **uvm_barrier**s (see 10.3).

10.4.2.1 Class declaration

```
class uvm_barrier_pool extends uvm_pool #(string,uvm_barrier)
```

10.4.2.2 Common methods

10.4.2.2.1 new

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

Initializes a new barrier pool object.

10.4.2.2.2 get_global_pool

```
static function uvm_barrier_pool get_global_pool()
```

Returns the singleton global barrier pool.

10.4.2.2.3 get_global

```
static function uvm_barrier_pool get_global (
   string key
)
```

Returns the item instance specified by *key* from the global item pool.

10.4.2.2.4 get

```
virtual function uvm_barrier get (
   string key
)
```

Returns the item with the given string *key*.

If no item exists by that *key*, a new item is created with that *key* and returned.

10.5 Objection mechanism

This subclause defines the objection mechanism.

10.5.1 uvm_objection

Objections provide a facility for coordinating status information between two or more participating components, objects, or even module-based IP.

10.5.1.1 Class declaration

```
class uvm_objection extends uvm_report_object
```

10.5.1.2 Common methods

new

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

Creates a new objection instance.

10.5.1.3 Objection control

10.5.1.3.1 get_propagate_mode

```
function bit get_propagate_mode()
```

Returns the propagation mode for this objection, as specified by **set_propagate_mode** (see 10.5.1.3.2). If **set_propagate_mode** has not been called since this objection was created, then **get_propagate_mode** shall return `1`.

10.5.1.3.2 set_propagate_mode

```
function void set_propagate_mode (
    bit prop_mode
)
```

Specifies the propagation mode for this objection. By default, objections support hierarchical propagation for components.

When propagation mode is set to '`0`', all intermediate callbacks between the *source* and *top* shall be skipped. Since the propagation mode changes the behavior of the objection, it can only be safely changed if there are no objections raised or draining. Any attempts to change the mode while objections are raised or draining shall result in an error.

10.5.1.3.3 raise_objection

```
virtual function void raise_objection (
    uvm_object obj = null,
    string description = "",
    int count = 1
)
```

Raises the number of objections for the source *object* by *count*, which defaults to `1`. The *object* is usually the *this* handle of the caller. If *object* is not specified or *null*, the implicit top-level component, **uvm_root** (see F.7), is chosen.

Raising an objection causes the following to occur:

— The source and total objection counts for *object* are increased by *count*. *description* is a string that marks a specific objection and is used in tracing/debug.

— The objection's **raised** virtual method (see 10.5.1.4.1) is called, which calls the **uvm_component::raised** method (see 13.1.5.4) for all of the components up the hierarchy.

10.5.1.3.4 drop_objection

```
virtual function void drop_objection (
    uvm_object obj = null,
    string description = "",
    int count = 1
)
```

Drops the number of objections for the source *object* by *count*, which defaults to `1`. The *object* is usually the *this* handle of the caller. If *object* is not specified or *null*, the implicit top-level component, **uvm_root** (see F.7), is chosen.

Dropping an objection causes the following to occur:

a) The source and total objection counts for *object* are decreased by *count*. *description* is a string that marks a specific objection and is used in tracing/debug. It is error to drop the objection count for *object* below zero (`0`).

b) The objection's **dropped** virtual method (see 10.5.1.4.2) is called, which calls the **uvm_component::dropped** method (see 13.1.5.5) for all of the components up the hierarchy.

c) If the total objection count has not reached zero (`0`) for *object*, the drop shall be propagated up the object hierarchy as with **raise_objection** (see 10.5.1.3.3). Then, each object in the hierarchy shall update their *source* counts, objections that they originated, and *total* counts, the total number of objections by them and all their descendants.

If the total objection count reaches zero (`0`), propagation up the hierarchy is deferred until a configurable drain time (see 10.5.1.3.7) has passed and the **uvm_component::all_dropped** callback (see 13.1.5.6) for the current hierarchy level has returned. The following process occurs for each instance up the hierarchy from the source caller:

d) A process is forked in a non-blocking fashion, allowing the *drop* call to return. The forked process then does the following.
   1) If a drain time was specified for the given *object*, the process waits for that amount of time.
   2) The objection's **all_dropped** virtual method (see 10.5.1.4.3) is called, which calls the **uvm_component::all_dropped** method (see 13.1.5.6) (if *object* is a component).
   3) The process then waits for the **all_dropped** callback to complete.
   4) After the drain time has elapsed and the **all_dropped** callback has completed, propagation of the dropped objection to the parent proceeds as described in **raise_objection** (see 10.5.1.3.3), except as described in item e.

e) If a new objection for this *object* or any of its descendants is raised during the drain time or during execution of the **all_dropped** callback (see 10.5.1.4.3) at any point, the hierarchical chain previously described is terminated and the dropped callback does not go up the hierarchy. The **raised** (see 10.5.1.4.1) objection propagates up the hierarchy, but the number of raised propagated up is reduced by the number of drops that were pending waiting for the **all_dropped**/drain time completion. Thus, if exactly one objection caused the count to go to zero (`0`), and during the drain exactly one new objection comes in, no raises or drops are propagated up the hierarchy.

As an optimization, if the *object* has no set drain time and no registered callbacks, the forked process shall be skipped and propagation proceeds immediately to the parent as described.

10.5.1.3.5 clear

```
virtual function void clear(
    uvm_object obj = null
)
```

Immediately clears the objection state. All counts are cleared and any processes waiting on a call to **wait_for** [with *objt_event* `UVM_ALL_DROPPED` and *obj* the implicit top-level component (see F.7)] are released (see 10.5.1.5.2).

The *obj* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method. The **clear** action does not result in objections being dropped, and therefore, does not result in the standard **::dropped** callback (see 10.5.1.4.2) being executed.

10.5.1.3.6 get_drain_time

```
function time get_drain_time (
   uvm_object obj = null
)
```

Returns the current drain time of the given *object* (default: `0 ns`).

10.5.1.3.7 set_drain_time

```
function void set_drain_time (uvm_object obj=null, time drain)
```

Specifies the drain time on the given *object* to *drain*.

Sets the *drain time*, which is the amount of time between the last remaining objection being dropped and the **all_dropped** callback (see 10.5.1.4.3) being called. If an objection is raised before this drain time expires, **all_dropped** is not called for this iteration.

10.5.1.4 Callback hooks

10.5.1.4.1 raised

```
virtual function void raised (
   uvm_object obj,
   uvm_object source_obj,
   string description,
   int count
)
```

Objection callback that is called when a **raise_objection** (see 10.5.1.3.3) has reached *obj*. The default implementation attempts to cast *obj* to a component, and, if successful, calls the **uvm_component::raised** hook (see 13.1.5.4).

10.5.1.4.2 dropped

```
virtual function void dropped (
   uvm_object obj,
   uvm_object source_obj,
   string description,
   int count
)
```

Objection callback that is called when a **drop_objection** (see 10.5.1.3.4) has reached *obj*. The default implementation attempts to cast *obj* to a component, and, if successful, calls the **uvm_component::dropped** hook (see 13.1.5.5).

10.5.1.4.3 all_dropped

```
virtual task all_dropped (
    uvm_object obj,
    uvm_object source_obj,
    string description,
    int count
)
```

Objection callback that is called when a **drop_objection** (see 10.5.1.3.4) has reached *obj*, and the total count for *obj* goes to zero (0). This callback is executed after the drain time associated with *obj*. The default implementation attempts to cast *obj* to a component, and, if successful, calls the **uvm_component::all_dropped** hook (see 13.1.5.6).

10.5.1.5 Objection status

10.5.1.5.1 get_objectors

```
function void get_objectors(
    ref uvm_object list[$]
)
```

Returns the current list of objecting objects (that raised an objection, but have not dropped it). *list* shall be a queue.

10.5.1.5.2 wait_for

```
task wait_for(
    uvm_objection_event objt_event,
    uvm_object obj = null
)
```

Waits for the **raised** (see 10.5.1.4.1), **dropped** (see 10.5.1.4.2), or **all_dropped** (see 10.5.1.4.3) event to occur in the given *obj*. If *obj* is *null*, the implicit top-level component (see F.7) is used. The task returns after all corresponding callbacks for that event have been executed.

10.5.1.5.3 get_objection_count

```
function int get_objection_count (
    uvm_object obj = null
)
```

Returns the current number of objections raised by the given *object*.

10.5.1.5.4 get_objection_total

```
function int get_objection_total (
    uvm_object obj = null
)
```

Returns the current number of objections raised by the given *object* and all descendants.

10.5.2 uvm_objection_callback

This is the callback type that defines the callback hooks for an objection callback.

NOTE—Users may use **uvm_objection_cbs_t** (see D.4.3) to add callbacks to specific objections.

10.5.2.1 Class declaration

```
class uvm_objection_callback extends uvm_callback
```

10.5.2.2 Methods

10.5.2.2.1 raised

```
virtual function void raised (
   uvm_objection objection,
   uvm_object obj,
   uvm_object source_obj,
   string description,
   int count
)
```

Objection `raised` callback function. Called by **uvm_objection::raised** (see 10.5.1.4.1).

10.5.2.2.2 dropped

```
virtual function void dropped (
   uvm_objection objection,
   uvm_object obj,
   uvm_object source_obj,
   string description,
   int count
)
```

Objection `dropped` callback function. Called by **uvm_objection::dropped** (see 10.5.1.4.2).

10.5.2.2.3 all_dropped

```
virtual task all_dropped (
   uvm_objection objection,
   uvm_object obj,
   uvm_object source_obj,
   string description,
   int count
)
```

Objection `all_dropped` callback function. Called by **uvm_objection::all_dropped** (see 10.5.1.4.3).

10.6 uvm_heartbeat

Heartbeats provide a way for environments to easily ensure their descendants are alive. A **uvm_heartbeat** is associated with a specific objection object. A component that is being tracked by the heartbeat object shall raise (or drop) the synchronizing objection during the heartbeat window.

The **uvm_heartbeat** object has a list of participating components. The heartbeat can be configured so that all components (`UVM_ALL_ACTIVE`), exactly one (`UVM_ONE_ACTIVE`), or any component (`UVM_ANY_ACTIVE`) trigger the objection in order to satisfy the heartbeat condition.

10.6.1 Class declaration

```
class uvm_heartbeat extends uvm_object
```

10.6.2 Methods

10.6.2.1 new

```
function new(
    string name,
    uvm_component cntxt,
    uvm_objection objection = null
)
```

Creates a new heartbeat instance associated with *cntxt*. The context is the hierarchical location where the heartbeat objections flow through and are monitored. The *objection* associated with the heartbeat is optional, if it is left *null*, but it needs to be specified before the heartbeat monitor will activate.

10.6.2.2 set_mode

```
function uvm_heartbeat_modes set_mode (
    uvm_heartbeat_modes mode = UVM_NO_HB_MODE
)
```

Specifies or retrieves the heartbeat mode. The current value for the heartbeat mode is returned. If an argument is specified to change the mode, then the mode is changed to the new value. The default value of *mode* shall be `UVM_NO_HB_MODE`.

10.6.2.3 set_heartbeat

```
function void set_heartbeat (
       uvm_event#(uvm_object) e,
    ref uvm_component comps[$]
)
```

Establishes the heartbeat event and assigns a list of components to watch. The monitoring is started as soon as this method is called. Once the monitoring has been started with a specific event, providing a new monitor event results in an error. To change trigger events, first **stop** (see 10.6.2.7) the monitor and then **start** (see 10.6.2.6) it with a new event trigger.

If the trigger event *e* is *null* and there was no previously set trigger event, the monitoring is not started. Monitoring can be started by explicitly calling **start** (see 10.6.2.6). *comps* shall be a queue.

10.6.2.4 add

```
function void add (
    uvm_component comp
)
```

Adds a single component to the set of components to be monitored. This does not cause monitoring to be started. If monitoring is currently active, this component is immediately added to the list of components and is expected to participate in the currently active event window.

10.6.2.5 remove

```
function void remove (
   uvm_component comp
)
```

Removes a single component to the set of components being monitored. Monitoring is not stopped, even if the last component has been removed (an explicit **stop** (see 10.6.2.7) is required).

10.6.2.6 start

```
function void start (
   uvm_event#(uvm_object) e = null
)
```

Starts the heartbeat monitor. If *e* is *null*, then whatever event was previously set is used. If no event was previously set, a warning shall be issued. It is an error if the monitor is currently running and *e* is specifying a different trigger event than the current event.

10.6.2.7 stop

```
function void stop()
```

Stops the heartbeat monitor. The current state information is reset so that if **start** (see 10.6.2.6) is called again the process waits for the first event trigger to start the monitoring.

10.7 Callbacks classes

This subclause defines the classes used for callback registration, management, and user-defined callbacks.

10.7.1 uvm_callback

The **uvm_callback** class is the base class for user-defined callback classes. Typically, the component developer defines an application-specific callback class that extends from this class. In it, one or more virtual methods are defined (*callback interfaces*) that represent the hooks available for user override.

10.7.1.1 Class declaration

```
class uvm_callback extends uvm_object
```

10.7.1.2 Methods

10.7.1.2.1 new

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

Initializes a new **uvm_callback** object, giving it an optional name.

10.7.1.2.2 callback_mode

```
function bit callback_mode(
   int on = -1
)
```

Enables/disables callbacks: `on==0` disables, `on==1` enables. Any value for *on* other than `0` or `1` has no effect on the enable state of the callback. The default value of *on* shall be `1`.

This returns the value of `1` if the callback was enabled before any change or `0` if the callback was disabled.

It also produces log messages if callback tracing is on.

10.7.1.2.3 is_enabled

```
function bit is_enabled()
```

Returns `1` if the callback is enabled, `0` otherwise.

10.7.2 uvm_callbacks #(T,CB)

The **uvm_callbacks** class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class. To work effectively, the developer of the component class defines a set of "hook" methods that enable users to customize certain behaviors of the component in a manner that is controlled by the component developer. The integrity of the component's overall behavior is intact, while still allowing certain customizable actions by the user.

To enable compile-time type-safety, the class is parameterized on both the user-defined callback interface implementation as well as the object type associated with the callback. The object type-callback type pair is associated using the **`uvm_register_cb** macro (see B.4.1) to define a valid pairing; valid pairings are checked when a user attempts to add a callback to an object.

10.7.2.1 Class declaration

```
class uvm_callbacks #(
    type T = uvm_object,
    type CB = uvm_callback
) extends uvm_object
```

10.7.2.2 Common parameters

10.7.2.2.1 T

This type parameter specifies the base object type with which the **CB** callback objects (see 10.7.2.2.2) are to be registered. This type shall be a derivative of **uvm_object** (see 5.3).

10.7.2.2.2 CB

This type parameter specifies the base callback type to be managed by this callback class. The callback type is typically a interface class, which defines one or more virtual method prototypes that users can override in subtypes. This type shall be a derivative of **uvm_callback** (see 10.7.1). When accessing the add/delete interface (see 10.7.2.3), the **CB** parameter is optional.

10.7.2.3 Add/delete interface

10.7.2.3.1 add

```
static function void add(
    T obj,
    uvm_callback cb,
```

```
    uvm_apprepend ordering = UVM_APPEND
)
```

Registers the given callback object, *cb*, with the given *obj* handle. Callbacks without a specified context are "type-wide," meaning they are called for all objects, as opposed to called for specific instances. If *ordering* is `UVM_APPEND` (the default), the callback is executed after previously added callbacks; otherwise, the callback is executed ahead of previously added callbacks. The *cb* is the callback handle; it shall be non-*null* and if the callback has already been added to the object instance then a warning shall be issued.

10.7.2.3.2 add_by_name

```
static function void add_by_name(
    string name,
    uvm_callback cb,
    uvm_component root,
    uvm_apprepend ordering = UVM_APPEND
)
```

Registers the given callback object, *cb*, with one or more **uvm_component**s (see 13.1). The components need to already exist and be type `T` or a derivative. *root* specifies the location in the component hierarchy to start the search for name. The default value of *ordering* shall be `UVM_APPEND`. See F.7.3.1 for more details on searching by name.

10.7.2.3.3 delete

```
static function void delete(
    T obj,
    uvm_callback cb
)
```

Deletes the given callback object, *cb*, from the queue associated with the given *obj* handle. The *obj* handle can be *null*, which allows de-registration of callbacks without an object context. The *cb* is the callback handle; it shall be non-*null* and if the callback has already been removed from the object instance then a warning shall be issued.

10.7.2.3.4 delete_by_name

```
static function void delete_by_name(
    string name,
    uvm_callback cb,
    uvm_component root
)
```

Removes the given callback object, *cb*, associated with one or more **uvm_component** (see 13.1) callback queues. *root* specifies the location in the component hierarchy to start the search for name. See F.7.3.1 for more details on searching by name.

10.7.2.4 Iterator interface

This set of functions provide an iterator interface for callback queues. A facade class, **uvm_callback_iter** (see D.1) is also available; it is the generally preferred way to iterate over callback queues.

10.7.2.4.1 get_first

```
static function CB get_first (
   ref int itr,
   input T obj
)
```

Returns the first enabled callback of type `CB` that resides in the queue for *obj*. If *obj* is *null*, the type wide queue for *T* is searched. *itr* is the iterator; it shall be updated with a value that can be supplied to **get_next** (see 10.7.2.4.3) to retrieve the next callback object.

If the queue is empty, *null* is returned.

10.7.2.4.2 get_last

```
static function CB get_last (
   ref int itr,
   input T obj
)
```

Returns the last enabled callback of type `CB` that resides in the queue for *obj*. If *obj* is *null*, the type wide queue for *T* is searched. *itr* is the iterator; it shall be updated with a value that can be supplied to **get_prev** (see 10.7.2.4.4) to retrieve the previous callback object.

If the queue is empty, *null* is returned.

10.7.2.4.3 get_next

```
static function CB get_next (
       ref int itr,
   input T obj
)
```

Returns the next enabled callback of type `CB` that resides in the queue for *obj*, using *itr* as the starting point. If *obj* is *null*, the type wide queue for *T* is searched. *itr* is the iterator; it shall be updated with a value that can be supplied to **get_next** (see 10.7.2.4.3) to retrieve the next callback object.

If no more callbacks exist in the queue, *null* is returned. **get_next** shall continue to return *null* in this case until **get_first** (see 10.7.2.4.1) has been used to reset the iterator.

10.7.2.4.4 get_prev

```
static function CB get_prev (
       ref int itr,
   input T obj
)
```

Returns the previous enabled callback of type `CB` that resides in the queue for *ob*, using *itr* as the starting point. If *obj* is *null*, the type wide queue for *T* is searched. *itr* is the iterator; it shall be updated with a value that can be supplied to **get_prev** (see 10.7.2.4.4) to retrieve the previous callback object.

If no more callbacks exist in the queue, *null* is returned. **get_prev** shall continue to return *null* in this case until **get_last** (see 10.7.2.4.2) has been used to reset the iterator.

10.7.2.5 get_all

```
static function void get_all (
   ref CB all_callbacks[$]
)
```

This function populates the end of the *all_callbacks* queue with the list of all registered callbacks (whether they are enabled or disabled).

11. Container classes

11.1 Overview

The container classes are type parameterized data structures. The **uvm_queue #(T)** class (see 11.3) implements a queue data structure similar to the SystemVerilog `queue` construct. And the **uvm_pool #(KEY,T)** class (see 11.2) implements a pool data structure similar to the SystemVerilog *associative array*. The class-based data structures allow the objects to be shared by reference; e.g., passing a **uvm_pool** as an input to a function copies only the class handle into the function, not the entire associative array.

11.2 uvm_pool #(KEY,T)

Implements a class-based dynamic associative array. Allows sparse arrays to be allocated on demand, and passed and stored by reference.

11.2.1 Class declaration

```
class uvm_pool #(
   type KEY = int,
        T = uvm_void
) extends uvm_object
```

11.2.2 Methods

11.2.2.1 new

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

Creates a new pool with the given *name*.

11.2.2.2 get_global_pool

```
static function uvm_pool #(KEY,T) get_global_pool()
```

Returns the singleton global pool for the item type `T`.

This allows items to be shared among components throughout the verification environment.

11.2.2.3 get_global

```
static function T get_global (
   KEY key
)
```

Returns the specified item instance from the global item pool.

11.2.2.4 get

```
virtual function T get (
   KEY key
)
```

Returns the item with the given *key*.

If no item exists by that *key*, a new item is allocated with that *key*, with a value as defined by Table 7-1 of IEEE Std 1800-2012.[6]

11.2.2.5 add

```
virtual function void add (
   KEY key,
   T item
)
```

Adds the given (*key, item*) pair to the pool. If an item already exists at the given *key* it is overwritten with the new *item*.

11.2.2.6 num

```
virtual function int num()
```

Returns the number of uniquely keyed items stored in the pool.

11.2.2.7 delete

```
virtual function void delete (
   KEY key
)
```

Removes the item with the given *key* from the pool.

11.2.2.8 exists

```
virtual function int exists (
   KEY key
)
```

Returns `1` if a item with the given *key* exists in the pool, `0` otherwise.

11.2.2.9 first

```
virtual function int first (
   ref KEY key
)
```

Returns the key of the first item stored in the pool.

If the pool is empty, then *key* is unchanged and `0` is returned.

If the pool is not empty, then *key* is the key of the first item and `1` is returned.

11.2.2.10 last

```
virtual function int last (
   ref KEY key
)
```

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

Returns the key of the last item stored in the pool.

If the pool is empty, then `0` is returned and *key* is unchanged.

If the pool is not empty, then *key* is set to the last key in the pool and `1` is returned.

11.2.2.11 next

```
virtual function int next (
   ref KEY key
)
```

Returns the key of the next item in the pool.

If the input key is the last key in the pool, then *key* is left unchanged and `0` is returned.

If a next key is found, then *key* is updated with that key and `1` is returned.

11.2.2.12 prev

```
virtual function int prev (
   ref KEY key
)
```

Returns the key of the previous item in the pool.

If the input key is the first key in the pool, then *key* is left unchanged and `0` is returned.

If a previous key is found, then key is updated with that key and 1 is returned.

11.3 uvm_queue #(T)

Implements a class-based dynamic queue. Allows queues to be allocated on demand, and passed and stored by reference.

11.3.1 Class declaration

```
class uvm_queue #(
   type T = int,
) extends uvm_object
```

11.3.2 Methods

11.3.2.1 new

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

Creates a new pool with the given *name*.

11.3.2.2 get_global_queue

```
static function uvm_queue #(T) get_global_queue()
```

Returns the singleton global queue for the item type `T`.

This allows items to be shared among components throughout the verification environment.

11.3.2.3 get_global

```
static function T get_global (
    int index
)
```

Returns the specified item instance from the global item queue.

11.3.2.4 get

```
virtual function T get (
    int index
)
```

Returns the item with the given *index*.

If *index* is equal to or greater than the size of the queue (see 11.3.2.5), an implementation shall issue a warning message and return the value for a non-existent array entries of type `T`, as defined by Table 7-1 of IEEE Std 1800-2012.

11.3.2.5 size

```
virtual function int size()
```

Returns the number of items stored in the queue.

11.3.2.6 insert

```
virtual function void insert (
    int index,
    T item
)
```

Inserts the item at the given *index* in the queue. If *index* is equal to or greater than the current size of the queue (see 11.3.2.5), the method call shall have no effect on the queue and an implementation shall issue a warning message.

11.3.2.7 delete

```
virtual function void delete (
    int index = -1
)
```

Removes the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted. The default value of *index* shall be `-1`.

11.3.2.8 pop_front

```
virtual function T pop_front()
```

Returns the first element in the queue (`index=0`). If the queue is empty, an implementation shall issue a warning message and return the value for a non-existent array entries of type `T`, as defined by Table 7-1 of IEEE Std 1800-2012.

11.3.2.9 pop_back

```
virtual function T pop_back()
```

Returns the last element in the queue (`index=size()-1`). If the queue is empty, an implementation shall issue a warning message and return the value for a non-existent array entries of type `T`, as defined by Table 7-1 of IEEE Std 1800-2012.

11.3.2.10 push_front

```
virtual function void push_front(
   T item
)
```

Inserts the given *item* at the front of the queue.

11.3.2.11 push_back

```
virtual function void push_back(
   T item
)
```

Inserts the given *item* at the back of the queue.

11.3.2.12 wait_until_not_empty

```
virtual task wait_until_not_empty()
```

If this queue is empty, blocks until not empty. If the queue is not empty, returns immediately.

12. UVM TLM interfaces

12.1 Overview

UVM provides a collection of classes and interfaces for transaction-level modeling (TLM). These objects enable transaction-level communication between entities, meaning requests are sent and responses received by transmitting transaction objects through various interfaces. The UVM TLM facility consists of two parts. UVM TLM 1 (see 12.2) is concerned with passing messages of arbitrary types through ports and exports. UVM TLM 2 (see 12.3) is concerned with modeling protocols and is based on sockets and a standardized transaction object called a generic payload. Sockets are constructed from ports and are connected in a similar manner (see 12.3.5). Sockets provide both blocking and non-blocking style of communication as well as forward and backward paths.

12.2 UVM TLM 1

12.2.1 General

Each UVM TLM 1 interface is either blocking, non-blocking, or a combination of the two, as follows:

a) *blocking*—A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Because delivery may consume time to complete, the methods in such an interface are declared as *tasks*.

b) *non-blocking*—A non-blocking interface conveys transactions in a non-blocking fashion; the methods return immediately regardless of success. Its methods are declared as *functions*. Because delivery may fail (e.g., the target component is busy and cannot accept the request), the methods may return with failed status.

c) *combination*—A combination interface contains both the blocking and non-blocking variants.

UVM TLM 1's port and export implementations allow connections between ports whose interfaces are not an exact match. For example, an `uvm_blocking_get_port` can be connected to any port, export, or imp port that provides, at a minimum, an implementation of the `blocking_get` interface, which includes the `uvm_get_*` ports, exports, and imps; the `uvm_blocking_get_peek_*` ports, exports, and imps; and the `uvm_get_peek_*` ports, exports, and imps.

UVM provides unidirectional (see 12.2.2) and bidirectional (see 12.2.3) ports, exports, and implementation ports for connecting components via the UVM TLM 1 interfaces.

1) *ports*—Instantiated in components that *require*, or *use*, the associate interface to initiate transaction requests.

2) *exports*—Instantiated by components that *forward* an implementation of the methods defined in the associated interface. An implementation is typically provided by an ***imp*** port in a child component.

3) *imps*—Instantiated by components that *provide* an implementation of or directly *implement* the methods defined in the associated interface.

Finally, the ***analysis*** interface is used to perform non-blocking broadcasts of transactions to connected components. It is typically used by components such as monitors to publish transactions observed on a bus to its subscribers, which are typically scoreboards and response/coverage collectors. See 12.2.10.

12.2.2 Unidirectional interfaces and ports

The unidirectional UVM TLM 1 interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the ***put*** (see 12.2.2.1), ***get***, and ***peek*** (see 12.2.2.2) interfaces, plus a non-blocking ***analysis*** interface (see 12.2.10).

12.2.2.1 put

The ***put*** interfaces are used to send, or *put*, transactions to other components. Successful completion of a ***put*** guarantees its delivery, not its execution.

12.2.2.2 get and peek

The ***get*** interfaces are used to retrieve transactions from other components. The ***peek*** interfaces are used for the same purpose, except the retrieved transaction is not consumed; successive calls to ***peek*** shall return the same object. Combined ***get_peek*** interfaces also can be used.

12.2.2.3 ports, exports, and imps

A summary of the unidirectional ***port***, ***export***, and ***imp*** declarations is as follows:

```
class uvm_*_export #(type T=int)
   extends uvm_port_base #(tlm_if_base #(T,T))
class uvm_*_port #(type T=int)
   extends uvm_port_base #(tlm_if_base #(T,T))
class uvm_*_imp #(type T=int)
   extends uvm_port_base #(tlm_if_base #(T,T))
```

where the asterisk (`*`) can be any of the following:

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek

analysis
```

12.2.3 Bidirectional interfaces and ports

The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the ***transport*** (see 12.2.3.1) and ***master*** and ***slave*** interfaces (see 12.2.3.2).

Bidirectional interfaces involve both a transaction request and response.

12.2.3.1 transport

The ***transport*** interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic. The request and response transactions can be different types.

12.2.3.2 master and slave

The primitive, unidirectional ***put***, ***get***, and ***peek*** interfaces (see 12.2.2) are combined to form bidirectional ***master*** and ***slave*** interfaces. The ***master*** puts requests and gets or peeks responses. The ***slave*** gets or peeks requests and puts responses. Because the ***put*** and the ***get*** come from different function interface methods, the requests and responses are not coupled as they are with the ***transport*** interface (see 12.2.3.1).

12.2.3.3 ports, exports, and imps

A summary of the bidirectional port, export, and imp declarations is as follows:

```
class uvm_*_port #(type REQ=int, RSP=int)
   extends uvm_port_base #(tlm_if_base #(REQ, RSP))
class uvm_*_export #(type REQ=int, RSP=int)
   extends uvm_port_base #(tlm_if_base #(REQ, RSP))
class uvm_*_imp #(type REQ=int, RSP=int)
   extends uvm_port_base #(tlm_if_base #(REQ, RSP))
```

where the asterisk (*) can be any of the following:

```
transport
blocking_transport
nonblocking_transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

12.2.4 uvm_tlm_if_base #(T1,T2)

This class declares all of the methods of the UVM TLM API. Various subsets of these methods are combined to form primitive UVM TLM interfaces, which are then paired in various ways to form more abstract “combination” UVM TLM interfaces. Components requiring a particular interface use ports to convey that requirement. Components providing a particular interface use exports to convey its availability.

Communication between components is established by connecting ports to compatible exports, much like connecting module signal-level output ports to compatible input ports. The difference is UVM TLM ports and exports bind interfaces (groups of methods), not signals and wires. The methods of the interfaces so bound pass data as whole transactions (e.g., objects). The set of primitive and combination UVM TLM interfaces afford many choices for designing components that communicate at the transaction level.

12.2.4.1 Class declaration

```
virtual class uvm_tlm_if_base #(
   type T1 = int,
```

```
    type T2 = int
)
```

12.2.4.2 Methods

12.2.4.2.1 put

```
virtual task put(
    input T1 t
)
```

Sends a user-defined transaction of type `T1`.

Components implementing the **put** method shall block the calling thread if they cannot immediately accept delivery of the transaction.

12.2.4.2.2 get

```
virtual task get(
    output T1 t
)
```

Provides a new transaction of type `T1`.

The calling thread is blocked if the requested transaction cannot be provided immediately. The new transaction is returned in the provided *output* argument. An implementation of **get** needs to regard the transaction as consumed. Subsequent calls to **get** shall return a different transaction instance.

12.2.4.2.3 peek

```
virtual task peek(
    output T1 t
)
```

Obtains a new transaction without consuming it.

If a transaction is available, it is written to the provided *output* argument. If a transaction is not available, the calling thread is blocked until one is available. The returned transaction is not consumed. A subsequent **peek** or **get** (see 12.2.4.2.2) shall return the same transaction.

12.2.4.2.4 try_put

```
virtual function bit try_put(
    input T1 t
)
```

Sends a transaction of type `T1`, if possible.

If the component is ready to accept the transaction argument, it does so and returns `1`; otherwise, it returns `0`.

12.2.4.2.5 can_put

```
virtual function bit can_put()
```

Returns `1` if the component is ready to accept the transaction; otherwise, it returns `0`.

12.2.4.2.6 try_get

```
virtual function bit try_get(
   output T2 t
)
```

Provides a new transaction of type `T2`.

If a transaction is immediately available, it is written to the *output* argument and `1` is returned. Otherwise, the *output* argument is not modified and `0` is returned.

12.2.4.2.7 can_get

```
virtual function bit can_get()
```

Returns `1` if a new transaction can be provided immediately upon request; otherwise, it returns `0`.

12.2.4.2.8 try_peek

```
virtual function bit try_peek(
   output T2 t
)
```

Provides a new transaction without consuming it.

If available, a transaction is written to the *output* argument and `1` is returned. A subsequent **peek** (see 12.2.4.2.3) or **get** (see 12.2.4.2.2) shall return the same transaction. If a transaction is not available, the *output* argument is unmodified and `0` is returned.

12.2.4.2.9 can_peek

```
virtual function bit can_peek()
```

Returns `1` if a new transaction is available; otherwise, it returns `0`.

12.2.4.2.10 transport

```
virtual task transport(
   input T1 req,
   output T2 rsp
)
```

Executes the given request and returns the response in the given *output* argument.

The calling thread may block until the operation is complete.

12.2.4.2.11 nb_transport

```
virtual function bit nb_transport(
   input T1 req,
   output T2 rsp
)
```

Executes the given request and returns the response in the given *output* argument.

Completion of this operation needs to occur without blocking. If the operation can not be executed immediately, a `0` shall be returned; otherwise, it returns `1`.

12.2.4.2.12 write

```
virtual function void write(
   input T1 t
)
```

Broadcasts a user-defined transaction of type `T1` to any number of listeners.

The operation needs to complete without blocking.

12.2.5 Port classes

The following classes define the UVM TLM 1 port classes.

12.2.5.1 uvm_*_port #(T)

These unidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its `min_size` is `0`, a port shall be connected to at least one implementation of its associated interface.

The asterisk (`*`) in **uvm_*_port** is any of the following:

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameter

> *T*—The type of transaction to be communicated by the export. The type *T* is not restricted to class handles and may be a value type such as `int`, `enum`, struct, or something similar.

Ports are connected to interface implementations directly via **uvm_*_imp #(T,IMP)** ports (see 12.2.7.1) or indirectly via hierarchical connections to **uvm_*_port #(T)** and **uvm_*_export #(T)** ports (see 12.2.6.1).

uvm_*_port #(T) has the following *methods*:

new

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

name and *parent* are the standard **uvm_component** constructor arguments (see 13.1). *min_size* and *max_size* specify the minimum and maximum number of interfaces that shall have been connected to this port by the end of elaboration. The default value of both *min_size* and *max_size* shall be `1`.

12.2.5.2 uvm_*_port #(REQ,RSP)

These bidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its `min_size` is `0`, a port shall be connected to at least one implementation of its associated interface.

The asterisk (`*`) in **uvm_*_port** is any of the following:

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Type parameters

REQ—The type of request transaction to be communicated by the export.

RSP—The type of response transaction to be communicated by the export.

Ports are connected to interface implementations directly via **uvm_*_imp #(REQ,RSP,IMP,REQ_IMP,RSP_IMP)** ports (see 12.2.7.2) or indirectly via hierarchical connections to **uvm_*_port #(REQ,RSP)** and **uvm_*_export #(REQ,RSP)** ports (see 12.2.6.2).

uvm_*_port #(REQ,RSP) has the following *methods*:

new

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

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. *min_size* and *max_size* specify the minimum and maximum number of interfaces that shall have been connected to this port by the end of elaboration. The default value of both *min_size* and *max_size* shall be `1`.

12.2.6 Export classes

The following classes define the UVM TLM 1 export classes.

12.2.6.1 uvm_*_export #(T)

This is a unidirectional port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port, and shall ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk (`*`) is any of the following:

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameter

T—The type of transaction to be communicated by the export.

Exports are connected to interface implementations directly via **uvm_*_imp #(T,IMP)** ports (see 12.2.7.1) or indirectly via hierarchical other **uvm_*_export #(T)** exports.

uvm_*_export #(T) has the following *methods*:

new

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

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. *min_size* and *max_size* specify the minimum and maximum number of interfaces that shall have been supplied to this port by the end of elaboration. The default value of both *min_size* and *max_size* shall be `1`.

12.2.6.2 uvm_*_export #(REQ,RSP)

This is a bidirectional port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port, and shall ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk (`*`) is any of the following:

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Type parameters

REQ—The type of request transaction to be communicated by the export.

RSP—The type of response transaction to be communicated by the export.

Exports are connected to interface implementations directly via **uvm_*_imp #(REQ,RSP,IMP,REQ_IMP,RSP_IMP)** ports (see 12.2.7.2) or indirectly via other **uvm_*_export #(REQ,RSP)** exports.

uvm_*_export #(REQ,RSP) has the following *methods*:

new

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

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. *min_size* and *max_size* specify the minimum and maximum number of interfaces that shall have been supplied to this port by the end of elaboration. The default value of both *min_size* and *max_size* shall be `1`.

12.2.7 Implementation (imp) classes

The following classes define the UVM TLM 1 implementation (imp) classes.

12.2.7.1 uvm_*_imp #(T,IMP)

This is a unidirectional imp port that provides access to an implementation of the associated interface to all connected ports and exports. Each imp port instance shall be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections are prohibited, e.g., to other ports and exports.

The asterisk (`*`) in **uvm_*_imp** is any of the following:

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameters

T—The type of transaction to be communicated by the export.

IMP—The type of the component implementing the interface, i.e., the class to which this imp delegates.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

uvm_*_imp #(T,IMP) has the following *methods*:

new

```
function new (string name, IMP parent)
```

Creates a new unidirectional imp port with the given *name* and *parent*. The *parent* shall implement the interface associated with this port. Its type shall be the type specified in the imp's type-parameter *IMP*.

12.2.7.2 uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)

This is a bidirectional imp port that provides access to an implementation of the associated interface to all connected ports and exports. Each imp port instance shall be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections are prohibited, e.g., to other ports and exports.

The interface represented by the asterisk (`*`) is any of the following:

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Type parameters

REQ—Request transaction type.

RSP—Response transaction type.

IMP—Component type that implements the interface methods, typically the parent of this imp port.

REQ_IMP—Component type that implements the request side of the interface. Defaults to `IMP`. For master and slave imps only.

RSP_IMP—Component type that implements the response side of the interface. Defaults to `IMP`. For master and slave imps only.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

The master and slave imps have two modes of operation.

— A single component of type *IMP* implements the entire interface for both requests and responses.
— Two sibling components of type *REQ_IMP* and *RSP_IMP* implement the request and response interfaces, respectively. In this case, the `IMP` parent instantiates this imp port and both the `REQ_IMP` and `RSP_IMP` components.

  This second mode is needed when a component instantiates more than one imp port, as for the **uvm_tlm_req_rsp_channel #(REQ,RSP)** channel (see 12.2.9.1).

uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP) has the following *methods*:

new

Creates a new unidirectional imp port with the given *name* and *parent*. The *parent*, whose type is specified by *IMP* type parameter, shall implement the interface associated with this port.

12.2.7.2.1 Transport imp constructor

```
function new (string name, IMP imp)
```

12.2.7.2.2 Master and slave imp constructor

The optional *req_imp* and *rsp_imp* arguments, which are available to master and slave imp ports, allow the requests and responses to be handled by different subcomponents. If they are specified, they shall point to the underlying component that implements the request and response methods, respectively.

```
function new (string name, IMP imp,
   REQ_IMP req_imp=imp, RSP_IMP rsp_imp=imp)
```

12.2.8 FIFO classes

The following classes define the UVM TLM 1-based FIFO (first-in, first-out) classes.

12.2.8.1 uvm_tlm_fifo_base#(T)

This class is the base for **uvm_tlm_fifo#(T)** (see 12.2.8.2). It defines the UVM TLM 1 exports through which all transaction-based FIFO operations occur. It also defines default implementations for each interface method provided by these exports.

The interface methods provided by **put_export** (see 12.2.8.1.3) and **get_peek_export** (see 12.2.8.1.4) are detailed in 12.2.2. See also Clause 12 for a general discussion of UVM TLM 1 interface definition and usage.

Type parameter

T—The type of transaction to be stored by this FIFO.

12.2.8.1.1 Class declaration

```
virtual class uvm_tlm_fifo_base#(
   type T = int
) extends uvm_component
```

12.2.8.1.2 Ports

uvm_tlm_fifo_base#(T) has the following ports (see 12.2.8.1.3 to 12.2.8.1.6).

12.2.8.1.3 put_export

This provides both the blocking and non-blocking `put` interface methods to any attached port:

```
task put (input T t)
function bit can_put()
function bit try_put (input T t)
```

Any `put` port variant can connect and send transactions to the FIFO via this export, provided the transaction types match. See 12.2.2 for more information on each of the above interface methods.

12.2.8.1.4 get_peek_export

This provides all the blocking and non-blocking `get` and `peek` interface methods:

```
task get (output T t)
function bit can_get()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the FIFO via this export, provided the transaction types match. See 12.2.4 for more information on each of the above interface methods.

12.2.8.1.5 put_ap

Transactions passed via `put` or `try_put` [via any port connected to the **put_export** (see 12.2.8.1.3)] are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps shall receive `put` transactions. See 12.2.2 for more information on the `write` method.

12.2.8.1.6 get_ap

Transactions passed via `get`, `try_get`, `peek`, or `try_peek` [via any port connected to the **get_peek_export** (see 12.2.8.1.4)] are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps shall receive `get` transactions. See 12.2.2 for more information on the `write` method.

12.2.8.1.7 Methods

new

```
function new(
   string name,
   uvm_component parent = null
)
```

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. The *parent* should be *null* if the **uvm_tlm_fifo** (see 12.2.8.2) is going to be used in a statically elaborated construct (e.g., a module).

12.2.8.2 uvm_tlm_fifo#(T)

This class provides storage of transactions between two independently running processes. Transactions are put into the FIFO via **put_export** (see 12.2.8.1.3). Transactions are fetched from the FIFO in the order they arrived via **get_peek_export** (see 12.2.8.1.4). The **put_export** and **get_peek_export** are inherited from the **uvm_tlm_fifo_base #(T)** super class (see 12.2.8.1.5), and the interface methods provided by these exports are noted in 12.2.2.

uvm_tlm_fifo #(T) has the following *methods*.

12.2.8.2.1 new

```
function new(
    string name,
    uvm_component parent = null,
    int size = 1
)
```

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. The *parent* should be *null* if the **uvm_tlm_fifo#(T)** is going to be used in a statically elaborated construct (e.g., a module). The *size* indicates the maximum size of the FIFO; a value of zero (`0`) indicates no upper bound. The default value of *size* shall be `1`.

12.2.8.2.2 size

```
virtual function int size()
```

Returns the capacity of the FIFO, i.e., the number of entries the FIFO is capable of holding. A return value of `0` indicates the FIFO capacity has no limit.

12.2.8.2.3 used

```
virtual function int used()
```

Returns the number of entries put into the FIFO.

12.2.8.2.4 is_empty

```
virtual function bit is_empty()
```

Returns `1` when there are no entries in the FIFO, `0` otherwise.

12.2.8.2.5 is_full

```
virtual function bit is_full()
```

Returns `1` when the number of entries in the FIFO is equal to its **size** (see 12.2.8.2.2), `0` otherwise.

12.2.8.2.6 flush

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

Removes all entries from the FIFO, after which **used** (see 12.2.8.2.3) returns `0` and **is_empty** (see 12.2.8.2.4) returns `1`.

12.2.8.3 uvm_tlm_analysis_fifo#(T)

This class is a **uvm_tlm_fifo#(T)** (see 12.2.8.2) with an unbounded size and a write interface. It can be used any place a **uvm_analysis_imp** (see 12.2.10.2) is used, e.g., as a buffer between a **uvm_analysis_port** (see 12.2.10.1) in an initiator component and a UVM TLM 1 target component.

12.2.8.3.1 Ports

analysis_export #(T)

This provides the write method to all connected analysis ports and parent exports:

```
function void write (T t)
```

Typically, access via ports bound to this export is used for writing to an analysis FIFO. See the `write` method noted in 12.2.2 for more information.

12.2.8.3.2 Methods

new

```
function new(
    string name,
    uvm_component parent = null
)
```

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. *name* is the local name of this component. The *parent* should be left unspecified when this component is instantiated in statically elaborated constructs and needs to be specified when this component is a child of another UVM component.

12.2.9 Channel classes

The following classes define the built-in UVM TLM 1 channel classes.

12.2.9.1 uvm_tlm_req_rsp_channel #(REQ,RSP)

This contains a request FIFO of type *REQ* and a response of type *RSP*. These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

Type parameters

REQ—The type of request transactions conveyed by this channel.

RSP—The type of response transactions conveyed by this channel.

12.2.9.1.1 Class declaration

```
class uvm_tlm_req_rsp_channel #(
    type REQ = int,
    type RSP = REQ
) extends uvm_component
```

12.2.9.1.2 Ports

uvm_tlm_req_rsp_channel #(REQ,RSP) has the following ports (see 12.2.9.1.3 to 12.2.9.1.10).

12.2.9.1.3 put_request_export

This provides both the blocking and non-blocking `put` interface methods to the request FIFO:

```
task put (input T t)
function bit can_put()
function bit try_put (input T t)
```

Any `put` port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

12.2.9.1.4 get_peek_response_export

This provides all the blocking and non-blocking `get` and `peek` interface methods to the response FIFO:

```
task get (output T t)
function bit can_get()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

12.2.9.1.5 get_peek_request_export

This provides all the blocking and non-blocking `get` and `peek` interface methods to the request FIFO:

```
task get (output T t)
function bit can_get()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the request FIFO via this export, provided the transaction types match.

12.2.9.1.6 put_response_export

This provides both the blocking and non-blocking `put` interface methods to the response FIFO:

```
task put (input T t)
function bit can_put()
function bit try_put (input T t)
```

Any `put` port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

12.2.9.1.7 request_ap

Transactions passed via `put` or `try_put` [via any port connected to the **put_request_export** (see 12.2.9.1.3)] are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps shall receive these transactions.

12.2.9.1.8 response_ap

Transactions passed via `put` or `try_put` [via any port connected to the **put_response_export** (see 12.2.9.1.6)] are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps shall receive these transactions.

12.2.9.1.9 master_export

Exports a single interface that allows a master to `put` requests and `get` or `peek` responses. It is a combination of **put_request_export** (see 12.2.9.1.3) and **get_peek_response_export** (see 12.2.9.1.4).

12.2.9.1.10 slave_export

Exports a single interface that allows a slave to `get` or `peek` requests and to `put` responses. It is a combination of **put_response_export** (see 12.2.9.1.6) and **get_peek_request_export** (see 12.2.9.1.5).

12.2.9.1.11 Methods

new

```
function new (
    string name,
    uvm_component parent = null,
    int request_fifo_size = 1,
    int response_fifo_size = 1
)
```

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. The *parent* shall be *null* if this component is defined within a static component such as a module, program block, or interface. The last two arguments specify the request and response FIFO sizes, which have default values of `1`.

12.2.9.2 uvm_tlm_transport_channel #(REQ,RSP)

A **uvm_tlm_transport_channel** is a **uvm_tlm_req_rsp_channel #(REQ,RSP)** (see 12.2.9.1) that implements the transport interface. It is useful when modeling a non-pipelined bus at the transaction level. Because the requests and responses have a tightly coupled one-to-one relationship, the request and response FIFO sizes are both set to one (`1`).

12.2.9.2.1 Class declaration

```
class uvm_tlm_transport_channel #(
    type REQ = int,
    type RSP = REQ
) extends uvm_tlm_req_rsp_channel #(REQ, RSP)
```

12.2.9.2.2 Ports

transport_export

This provides both the blocking and non-blocking `transport` interface methods to the response FIFO:

```
task transport(REQ request, output RSP response)
function bit nb_transport(REQ request, output RSP response)
```

Any `transport` port variant can connect to and send requests and retrieve responses via this export, provided the transaction types match. Upon return, the *response* argument carries the response to the request.

12.2.9.2.3 Methods

new

```
function new (
    string name,
    uvm_component parent = null,
)
```

name and *parent* are the standard **uvm_component** (see 13.1) constructor arguments. The *parent* shall be *null* if this component is defined within a statically elaborated construct such as a module, program block, or interface.

12.2.10 Analysis ports

This subclause defines the port, export, and imp classes used for transaction analysis.

12.2.10.1 uvm_analysis_port

Broadcasts a value to all subscribers implementing a **uvm_analysis_imp** (see 12.2.10.2).

12.2.10.1.1 Class declaration

```
class uvm_analysis_port # (
    type T = int
) extends uvm_port_base # (uvm_tlm_if_base #(T,T))
```

12.2.10.1.2 Methods

write

```
function void write (
    input T t
)
```

Sends the specified value to all connected interfaces.

12.2.10.2 uvm_analysis_imp

Receives all transactions broadcasted by a **uvm_analysis_port** (see 12.2.10.1). This serves as the termination point of an analysis port/export/imp connection. The component attached to the *imp* class—called a *subscriber*—implements the analysis interface.

This invokes the `write(T)` method in the parent component. An implementation of the `write(T)` method shall not modify the value passed to it.

Class declaration

```
class uvm_analysis_imp #(
    type T = int,
    type IMP = int
) extends uvm_port_base #(uvm_tlm_if_base #(T,T))
```

12.2.10.3 uvm_analysis_export

Exports a lower-level **uvm_analysis_imp** (see 12.2.10.2) to its parent.

12.2.10.3.1 Class declaration

```
class uvm_analysis_export # (
    type T = int
) extends uvm_port_base # (uvm_tlm_if_base #(T,T))
```

12.2.10.3.2 Methods

new

```
function new (
    string name,
    uvm_component parent = null
)
```

Instantiates the export.

12.3 UVM TLM 2

12.3.1 General

UVM TLM 2 defines a generic payload (see 12.3.4), which is the base type for transport interfaces that may be blocking or non-blocking. The interface is categorized as a port (see 12.3.6), export (see 12.3.7), or implementation (see 12.3.8). The interface may also be implemented in sockets (see 12.3.5), which provide both a forward and a backward path.

12.3.2 uvm_tlm_if: transport interfaces

UVM TLM 2 provides the following two transport interfaces (see 12.3.2.2):

a) *Blocking* (**b_transport**)—completes the entire transaction within a single method call.
b) *Non-blocking* (**nb_transport**)—describes the progress of a transaction using multiple **nb_transport** method calls going back-and-forth between initiator and target.

In general, any component might modify a transaction object during its lifetime (subject to the rules of the protocol). Significant timing points during the lifetime of a transaction (e.g., start-of-response phase) are indicated by calling **nb_transport** in either forward or backward direction, the specific timing point being given by the *phase* argument. Protocol-specific rules for reading or writing the attributes of a transaction can be expressed relative to the phase. The phase can be used for flow control, and for that reason might have a different value at each hop taken by a transaction; the phase is not an attribute of the transaction object.

A call to **nb_transport** (see 12.2.4.2.11) always represents a phase transition. However, the return from **nb_transport** might or might not do so; the choice being indicated by the value returned from the function [**UVM_TLM_ACCEPTED** versus **UVM_TLM_UPDATED** (see 12.3.3.2)].

Generally, the completion of a transaction over a particular hop is shown by using the value of the *phase* argument. As a shortcut, a target might indicate the completion of the transaction by returning a special value of **UVM_TLM_COMPLETED** (see 12.3.3.2). However, this is optional.

The transaction object itself does not contain any timing information by design or even events and the status from the API. Delays can be passed as arguments to **b_transport**/**nb_transport** (see 12.3.2.2); this pushes the actual realization of any delay in the simulator kernel downstream and defers it (for simulation speed).

uvm_tlm_if is the base class type to define the transport methods (see 12.3.2.2).

12.3.2.1 Class declaration

```
class uvm_tlm_if #(
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e
)
```

12.3.2.2 Transport methods

Each of the interface methods (see 12.3.2.2.1 to 12.3.2.2.3) take a handle to the transaction to be transported and a reference argument for the delay. In addition, the non-blocking interfaces take a reference argument for the phase.

12.3.2.2.1 nb_transport_fw

```
virtual function uvm_tlm_sync_e nb_transport_fw(
          T t,
       ref P p,
    input uvm_tlm_time delay
)
```

This is a forward path call. The first call to this method for a transaction marks the initial timing point. Every call to this method may mark a timing point in the execution of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the forward path is used. The final timing point of a transaction may be marked by a call to **nb_transport_bw** (see 12.3.4) or a return from this call or a subsequent call to **nb_transport_fw**.

See 12.3.2 for more details on the semantics and rules of the non-blocking transport interface.

12.3.2.2.2 nb_transport_bw

```
virtual function uvm_tlm_sync_e nb_transport_bw(
          T t,
       ref P p,
    input uvm_tlm_time delay
)
```

This is an implementation of a backward path. This function shall be implemented in the `INITIATOR` component class.

Every call to this method may mark a timing point, including the final timing point, in the execution of the transaction. The timing annotation argument allows the timing point to be offset from the simulation times at which the backward path is used. The final timing point of a transaction may be marked by a call to **nb_transport_fw** (see 12.3.2.2.1) or a return from this call or a subsequent call to **nb_transport_bw**.

See 12.3.2 for more details on the semantics and rules of the non-blocking transport interface.

12.3.2.2.3 b_transport

```
virtual task b_transport(
   T t,
   uvm_tlm_time delay
)
```

This executes a blocking transaction. Once this method returns, the transaction is presumed to have been executed. Whether that execution is successful or not shall be indicated by the transaction itself.

The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class. The initiator may reuse a transaction object from one call to the next and across calls to **b_transport**.

The call to **b_transport** shall mark the first timing point of the transaction. The return from **b_transport** shall mark the final timing point of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the task call and return are executed.

12.3.3 Enumerations

12.3.3.1 uvm_tlm_phase_e

Designates non-blocking transport synchronization state values between an initiator and a target.

UNINITIALIZED_PHASE—Defaults for the constructor.

BEGIN_REQ—Beginning of the request phase.

END_REQ—End of the request phase.

BEGIN_RESP—Beginning of the response phase.

END_RESP—End of the response phase.

12.3.3.2 uvm_tlm_sync_e

These are the predefined phase state values for the non-blocking transport base protocol between an initiator and a target.

UVM_TLM_ACCEPTED—The transaction has been accepted.

UVM_TLM_UPDATED—The transaction has been modified.

UVM_TLM_COMPLETED—Execution of the transaction is complete.

12.3.4 Generic payload and extensions

The *generic payload* transaction represents a generic bus read/write access. It is used as the default transaction in UVM TLM 2 blocking and non-blocking transport interfaces.

12.3.4.1 Globals

Defines constants and enums.

12.3.4.1.1 uvm_tlm_command_e

This specifies the command attribute type definition.

UVM_TLM_READ_COMMAND—Bus read operation.
UVM_TLM_WRITE_COMMAND—Bus write operation.
UVM_TLM_IGNORE_COMMAND—No bus operation.

12.3.4.1.2 uvm_tlm_response_status_e

This specifies the response status attribute type definition.

UVM_TLM_OK_RESPONSE—Bus operation completed successfully.
UVM_TLM_INCOMPLETE_RESPONSE—Transaction was not delivered to target.
UVM_TLM_GENERIC_ERROR_RESPONSE—Bus operation had an error.
UVM_TLM_ADDRESS_ERROR_RESPONSE—Invalid address specified.
UVM_TLM_COMMAND_ERROR_RESPONSE—Invalid command specified.
UVM_TLM_BURST_ERROR_RESPONSE—Invalid burst specified.
UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE—Invalid byte enabling specified.

12.3.4.2 uvm_tlm_generic_payload

This class provides a transaction definition commonly used in memory-mapped bus-based systems. It is intended to be a general purpose transaction class that lends itself to many applications. The class is derived from **uvm_sequence_item** (see 14.1), which enables it to be generated in sequences and transported to drivers through sequencers.

12.3.4.2.1 Class declaration

```
class uvm_tlm_generic_payload extends uvm_sequence_item
```

12.3.4.2.2 Common methods

uvm_tlm_generic_payload has the following common methods (see 12.3.4.2.3 to 12.3.4.2.11).

12.3.4.2.3 new

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

Creates a new instance of the generic payload. This also initializes all the members to their default values.

12.3.4.2.4 m_address

```
rand bit [63:0] m_address
```

This is the address for the bus operation. It should be specified or read using the **set_address** (see 12.3.4.2.19) and **get_address** (see 12.3.4.2.19) methods. This variable should be used only when constraining.

For a read or write command, the target shall interpret the current value of the address attribute as the start address in the system memory map of the contiguous block of data being read or written. The address associated with any given byte in the data array is dependent upon the address attribute, the array index, the streaming width attribute, the endianness, and the width of the physical bus.

If the target is unable to execute the transaction with the given address attribute (because the address is out-of-range, for example) it shall generate a standard error response. The recommended response status is `UVM_TLM_ADDRESS_ERROR_RESPONSE`.

12.3.4.2.5 m_command

```
rand uvm_tlm_command_e m_command
```

This is the bus operation type. It should be specified using the **set_command** (see 12.3.4.2.14), **set_read** (see 12.3.4.2.16), or **set_write** (see 12.3.4.2.18) methods and read using the **get_command** (see 12.3.4.2.13), **is_read** (see 12.3.4.2.15), or **is_write** (see 12.3.4.2.17) methods. This variable should be used only when constraining.

If the target is unable to execute a read or write command, it shall generate a standard error response. The recommended response status is `UVM_TLM_COMMAND_ERROR_RESPONSE`.

On the receipt of a generic payload transaction where the command attribute is equal to `UVM_TLM_IGNORE_COMMAND`, the target shall not execute a write command or a read command that does not modify any data. The target may, however, use the value of any attribute in the generic payload, including any extensions.

The command attribute shall be specified by the initiator and shall not be overwritten by any interconnect.

12.3.4.2.6 m_data

```
rand byte unsigned m_data[]
```

This is data read or to be written. It should be specified and read using the **set_data** (see 12.3.4.2.22) or **get_data** (see 12.3.4.2.21) methods. The variable should be used only when constraining.

For a read command or a write command, the target shall copy data to or from the data array, respectively, honoring the semantics of the remaining attributes of the generic payload.

For a write command or `UVM_TLM_IGNORE_COMMAND`, the contents of the data array shall be specified by the initiator, and shall not be overwritten by any interconnect component or target. For a read command, the contents of the data array shall only be overwritten by the target (honoring the semantics of the byte enable).

Arbitrary data types may be converted to and from a byte array using the streaming operator and **uvm_object** objects (see 5.3) may be further converted using the **uvm_object::pack_bytes** and **uvm_object::unpack_bytes** methods (see 5.3.10.1). Simply use a consistent mechanism to both fill the payload data array and later extract data from it.

12.3.4.2.7 m_length

```
rand int unsigned m_length
```

This is the number of bytes to be copied to or from the **m_data** array (see 12.3.4.2.6), inclusive of any bytes disabled by the **m_byte_enable** attribute (see 12.3.4.2.9).

The data length attribute shall be specified by the initiator and shall not be overwritten by any interconnect component or target.

The data length attribute shall not be set to `0`. In order to transfer zero bytes, the **m_command** attribute (see 12.3.4.2.5) should be specified as `UVM_TLM_IGNORE_COMMAND`.

12.3.4.2.8 m_response_status

```
rand uvm_tlm_response_status_e m_response_status
```

This is the status of the bus operation. It should be specified using the **set_response_status** method (see 12.3.4.2.34) and read using the **get_response_status** (see 12.3.4.2.33), **get_response_string** (see 12.3.4.2.37), **is_response_ok** (see 12.3.4.2.35), or **is_response_error** (see 12.3.4.2.36) methods. This variable should be used only when constraining.

The response status attribute shall be specified to `UVM_TLM_INCOMPLETE_RESPONSE` by the initiator and may be overwritten by the target. The response status attribute should not be overwritten by any interconnect component, because the default value `UVM_TLM_INCOMPLETE_RESPONSE` indicates the transaction was not delivered to the target.

The target may specify the response status attribute as `UVM_TLM_OK_RESPONSE` to indicate it was able to execute the command successfully or specify one of the five error responses to indicate an error. The target should choose the appropriate error response depending on the cause of the error. If a target detects an error, but is unable to select a specific error response, it may specify the response status as `UVM_TLM_GENERIC_ERROR_RESPONSE`.

The target shall be responsible for specifying the response status attribute at the appropriate point in the lifetime of the transaction. In the case of the blocking transport interface, this means before returning control from **b_transport** (see 12.3.2.2.3). In the case of the non-blocking transport interface and the base protocol, this means before sending the *BEGIN_RESP* phase or returning a value of `UVM_TLM_COMPLETED`.

It is recommended that the initiator always checks the response status attribute on receiving a transition to the *BEGIN_RESP* phase or after the completion of the transaction. An initiator may choose to ignore the response status if it is known in advance the value will be `UVM_TLM_OK_RESPONSE`—say it is known this initiator is only connected to targets that always return `UVM_TLM_OK_RESPONSE`—but, in general, this will not be the case. In other words, the initiator can only ignore the response status at its own risk.

12.3.4.2.9 m_byte_enable

```
rand byte unsigned m_byte_enable[]
```

Indicates valid **m_data** (see 12.3.4.2.6) array elements. Should be specified and read using the **set_byte_enable** (see 12.3.4.2.28) or **get_byte_enable** (see 12.3.4.2.27) methods The variable should be used only when constraining.

The elements in the byte enable array shall be interpreted as follows. A value of `8'h00` indicates the corresponding byte is disabled and a value of `8'hFF` indicates the corresponding byte is enabled.

Byte enables may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat or to place words in selected byte lanes of a bus. At a more abstract level, byte enables may be used to create “lacy bursts” where the data array of the generic payload has an arbitrary pattern of holes punched in it.

The byte enable mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array. The byte enable array may be empty, in which case byte enables shall not be used for the current transaction.

The byte enable array shall be specified by the initiator and shall not be overwritten by any interconnect component or target.

If the byte enable pointer is not empty, the target shall implement the semantics of the byte enable defined as follows or generate a standard error response. The recommended response status is `UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE`.

In the case of a write command, any interconnect component or target should ignore the values of any disabled bytes in the **m_data** array (see 12.3.4.2.6). In the case of a read command, any interconnect component or target should not modify the values of disabled bytes in the **m_data** array.

12.3.4.2.10 m_byte_enable_length

```
rand byte unsigned m_byte_enable_length
```

This is the number of elements in the **m_byte_enable** array (see 12.3.4.2.9).

It shall be specified by the initiator, and shall not be overwritten by any interconnect component or target.

12.3.4.2.11 m_streaming_width

```
rand byte unsigned m_streaming_width
```

This is the number of bytes transferred on each beat. Should be specified and read using the **set_streaming_width** (see 12.3.4.2.26) or **get_streaming_width** (see 12.3.4.2.25) methods. This variable should be used only when constraining.

Streaming affects the way a component should interpret the data array. A stream consists of a sequence of data transfers occurring on successive notional beats, each beat having the same start address as given by the generic payload address attribute. The streaming width attribute shall determine the width of the stream, i.e., the number of bytes transferred on each beat. In other words, streaming affects the local address associated with each byte in the data array. In all other respects, the organization of the data array is unaffected by streaming.

The bytes within the data array have a corresponding sequence of local addresses within the component accessing the generic payload transaction. The lowest address is given by the value of the address attribute. The highest address is given by the formula *address_attribute* + *streaming_width* − `1`. The address to or from which each byte is being copied in the target shall be specified as the value of the address attribute at the start of each beat.

With respect to the interpretation of the data array, a single transaction with a streaming width shall be functionally equivalent to a sequence of transactions each having the same address as the original transaction, each having a data length attribute equal to the streaming width of the original, and each with a data array that is a different subset of the original data array on each beat. This subset effectively steps down the original data array maintaining the sequence of bytes.

A streaming width of 0 indicates a streaming transfer is not required. This is equivalent to a streaming width value greater than or equal to the size of the **m_data** array (see 12.3.4.2.6).

Streaming may be used in conjunction with byte enables, where the streaming width would typically be equal to the byte enable length. It would also make sense to have the streaming width be a multiple of the byte enable length. Having the byte enable length be a multiple of the streaming width implies different bytes were enabled on each beat.

If the target is unable to execute the transaction with the given streaming width, it shall generate a standard error response. The recommended response status is `TLM_BURST_ERROR_RESPONSE`.

12.3.4.2.12 Accessors

The accessor functions (see 12.3.4.2.13 to 12.3.4.2.37) can specify and retrieve each of the members of the generic payload. All of the accessor methods are virtual.

12.3.4.2.13 get_command

```
virtual function uvm_tlm_command_e get_command()
```

Returns the value of the **m_command** variable (see 12.3.4.2.5).

12.3.4.2.14 set_command

```
virtual function void set_command(
   uvm_tlm_command_e command
)
```

Specifies the value of the **m_command** variable (see 12.3.4.2.5).

12.3.4.2.15 is_read

```
virtual function bit is_read()
```

Returns *true* if the current value of the **m_command** variable (see 12.3.4.2.5) is `UVM_TLM_READ_COMMAND`.

12.3.4.2.16 set_read

```
virtual function void set_read()
```

Specifies the current value of the **m_command** variable (see 12.3.4.2.5) to `UVM_TLM_READ_COMMAND`.

12.3.4.2.17 is_write

```
virtual function bit is_write()
```

Returns *true* if the current value of the **m_command** variable (see 12.3.4.2.5) `UVM_TLM_WRITE_COMMAND`.

12.3.4.2.18 set_write

```
virtual function void set_write()
```

Specifies the current value of the **m_command** variable (see 12.3.4.2.5) to `UVM_TLM_WRITE_COMMAND`.

12.3.4.2.19 get_address

```
virtual function bit [63:0] get_address()
```

Returns the value of the **m_address** variable (see 12.3.4.2.4).

12.3.4.2.20 set_address

```
virtual function void set_address(
   bit [63:0] addr
)
```

Specifies the value of the **m_address** variable (see 12.3.4.2.4).

12.3.4.2.21 get_data

```
virtual function void get_data (
   output byte unsigned p []
)
```

Returns the value of the **m_data** array (see 12.3.4.2.6).

12.3.4.2.22 set_data

```
virtual function void set_data (
   ref byte unsigned p []
)
```

Specifies the value of the **m_data** array (see 12.3.4.2.6).

12.3.4.2.23 get_data_length

```
virtual function int unsigned get_data_length()
```

Returns the current size of the **m_data** array (see 12.3.4.2.6).

12.3.4.2.24 set_data_length

```
virtual function void set_data_length(
   int unsigned length
)
```

Specifies the value of the **m_length** (see 12.3.4.2.7).

12.3.4.2.25 get_streaming_width

```
virtual function int unsigned get_streaming_width()
```

Returns the value of the **m_streaming_width** array (see 12.3.4.2.11).

12.3.4.2.26 set_streaming_width

```
virtual function void set_streaming_width(
   int unsigned width
)
```

Specifies the value of the **m_streaming_width** array (see 12.3.4.2.11).

12.3.4.2.27 get_byte_enable

```
virtual function void get_byte_enable(
   output byte unsigned p[]
)
```

Returns the value of the **m_byte_enable** array (see 12.3.4.2.9).

12.3.4.2.28 set_byte_enable

```
virtual function void set_byte_enable(
   ref byte unsigned p[]
)
```

Specifies the value of the **m_byte_enable** array (see 12.3.4.2.9).

12.3.4.2.29 get_byte_enable_length

```
virtual function int unsigned get_byte_enable_length()
```

Returns the current size of the **m_byte_enable** array (see 12.3.4.2.9).

12.3.4.2.30 set_byte_enable_length

```
virtual function void set_byte_enable_length(
   int unsigned length
)
```

Specifies the size **m_byte_enable_length** (see 12.3.4.2.10) of the **m_byte_enable** array (see 12.3.4.2.9), i.e., `m_byte_enable.size`.

12.3.4.2.31 set_dmi_allowed

```
virtual function void set_dmi_allowed(
   bit dmi
)
```

This is a DMI hint. It allows DMI access.

12.3.4.2.32 is_dmi_allowed

```
virtual function bit is_dmi_allowed()
```

This is a DMI hint. It queries to see if DMI access is allowed.

12.3.4.2.33 get_response_status

```
virtual function uvm_tlm_response_status_e get_response_status()
```

Returns the current value of the **m_response_status** variable (see 12.3.4.2.8).

12.3.4.2.34 set_response_status

```
virtual function void set_response_status(
   uvm_tlm_response_status_e status
)
```

Specifies the current value of the **m_response_status** variable (see 12.3.4.2.8).

12.3.4.2.35 is_response_ok

```
virtual function bit is_response_ok()
```

Returns *true* if the current value of the **m_response_status** variable (see 12.3.4.2.8) is `UVM_TLM_OK_RESPONSE`.

12.3.4.2.36 is_response_error

```
virtual function bit is_response_error()
```

Returns *true* if the current value of the **m_response_status** variable (see 12.3.4.2.8) is not `UVM_TLM_OK_RESPONSE`.

12.3.4.2.37 get_response_string

```
virtual function string get_response_string()
```

Returns the current value of the **m_response_status** variable (see 12.3.4.2.8) as a string.

12.3.4.2.38 Extension mechanism

uvm_tlm_generic_payload has the following extension mechanisms (see 12.3.4.2.39 to 12.3.4.2.43).

12.3.4.2.39 get_num_extensions

```
function int get_num_extensions()
```

Returns the current number of instance specific extensions.

12.3.4.2.40 get_extension

```
function uvm_tlm_extension_base get_extension(
   uvm_tlm_extension_base ext_handle
)
```

Returns the instance specific extension bound under the specified key. If no extension is bound under that key, *null* is returned.

12.3.4.2.41 set_extension

```
function uvm_tlm_extension_base set_extension(
   uvm_tlm_extension_base ext
)
```

Adds an instance-specific extension. Only one instance of any given extension type is allowed. If there is an existing extension instance of the type of *ext*, *ext* replaces it and its handle is returned. Otherwise, *null* is returned.

12.3.4.2.42 clear_extension

```
function void clear_extension(
   uvm_tlm_extension_base ext_handle
)
```

Removes the instance-specific extension bound under the specified key.

12.3.4.2.43 clear_extensions

```
function void clear_extensions()
```

Removes all instance-specific extensions.

12.3.4.3 uvm_tlm_gp

This typedef provides a short, more convenient name for the **uvm_tlm_generic_payload** type (see 12.3.4.2).

Class declaration

```
typedef uvm_tlm_generic_payload uvm_tlm_gp
```

12.3.4.4 uvm_tlm_extension_base

This is the non-parameterized base class for all generic payload extensions. The pure virtual function **get_type_handle** (see 12.3.4.4.4) returns a unique handle that represents the derived type, which is implemented in derived classes.

This class shall never be extended by user classes; such user classes shall extend from **uvm_tlm_extension** (see 12.3.4.5).

12.3.4.4.1 Class declaration

```
virtual class uvm_tlm_extension_base extends uvm_object
```

12.3.4.4.2 Methods

uvm_tlm_extension_base has the following methods (see 12.3.4.4.3 to 12.3.4.4.5).

12.3.4.4.3 new

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

The **new** constructor is only given as a pass-through mechanism to call **uvm_object::new**. This class is abstract and cannot be constructed itself.

12.3.4.4.4 get_type_handle

```
pure virtual function uvm_tlm_extension_base get_type_handle()
```

Intended to be an interface to polymorphically retrieve a handle that uniquely identifies the type of the subclass.

12.3.4.4.5 get_type_handle_name

```
pure virtual function string get_type_handle_name()
```

Intended to be an interface to polymorphically retrieve the name that uniquely identifies the type of the subclass.

12.3.4.5 uvm_tlm_extension

This is a UVM TLM 2 extension class. This class is parameterized with an arbitrary type that represents the type of the extension. An instance of the generic payload can contain one extension object of each type; it cannot contain two instances of the same extension type.

The extension type can be identified using the **ID** method (see 12.3.4.5.4).

To implement a generic payload extension, simply derive a new class from this class and specify the name of the derived class as the extension parameter.

12.3.4.5.1 Class declaration

```
class uvm_tlm_extension #(
   type T = int
) extends uvm_tlm_extension_base
```

12.3.4.5.2 Methods

uvm_tlm_extension has the following methods (see 12.3.4.5.3 to 12.3.4.5.4).

12.3.4.5.3 new

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

Creates a new extension object.

12.3.4.5.4 ID

```
static function uvm_tlm_extension #(T) ID()
```

Returns the unique ID of this UVM TLM 2 extension type. This method is used to identify the type of the extension to retrieve from a **uvm_tlm_generic_payload** instance (see 12.3.4.2), using the **uvm_tlm_generic_payload::get_extension** method (see 12.3.4.2.40).

12.3.5 Sockets

Sockets group together all the necessary core interfaces for transportation and binding. A socket is like a port or export; in fact it is derived from the same base class as port and export, namely **uvm_port_base #(IF)** (see 5.5). However, unlike a port or export, a socket provides both a forward and backward path. Thus asynchronous (pipelined) bidirectional communication can be enabled by connecting sockets together. A socket contains both a port and an export. Components that initiate transactions are called *initiators* and components that receive transactions sent by an initiator are called *targets*. Initiators have initiator sockets and targets have target sockets. Initiator sockets can connect to target sockets. Initiator sockets cannot be connected to other initiator sockets and target sockets cannot be connected to other target sockets.

Sockets come in several flavors: Each socket is either an initiator or a target, a pass-through, or a terminator. Furthermore, any particular socket implements either the blocking interfaces or the non-blocking interfaces. Terminator sockets are used on initiators and targets as well as interconnect components. Pass-through sockets are used to enable connections to cross hierarchical boundaries.

There are eight socket types: the cross of blocking and non-blocking, pass-through and termination, and target and initiator.

Sockets are specified based on what they are (`IS-A`) and what they contain (`HAS-A`). `IS-A` and `HAS-A` are types of object relationships. `IS-A` refers to the inheritance relationship and `HAS-A` refers to the ownership relationship. For example, the statement `D` is a `B` means `D` is derived from base `B`. Given that, the phrase object `A HAS-A B` then means `B` is a member of `A`.

12.3.5.1 uvm_tlm_b_target_socket

`IS-A` forward imp; has no backward path except via the payload contents.

The component instantiating this socket shall implement a **b_transport** method (see 12.3.2.2.3) with the following signature:

```
task b_transport(T t, uvm_tlm_time delay)
```

12.3.5.1.1 Class declaration

```
class uvm_tlm_b_target_socket #(
    type IMP = int,
    type T = uvm_tlm_generic_payload
) extends uvm_tlm_b_target_socket_base #(T)
```

12.3.5.1.2 Methods

uvm_tlm_b_target_socket has the following methods (see 12.3.5.1.3 to 12.3.5.1.4).

12.3.5.1.3 new

```
function new (
    string name,
    uvm_component parent,
    IMP imp = null
)
```

Constructs a new instance of this socket *imp*, a reference to the class implementing the **b_transport** method (see 12.3.2.2.3). If not specified, it is presumed to be the same as *parent*.

12.3.5.1.4 connect

```
function void connect(uvm_tlm_b_target_socket provider)
```

Connects this socket to the specified **uvm_tlm_b_initiator_socket** (see 12.3.5.2).

12.3.5.2 uvm_tlm_b_initiator_socket

`IS-A` forward port; has no backward path except via the payload contents.

12.3.5.2.1 Class declaration

```
class uvm_tlm_b_initiator_socket #(
   type T = uvm_tlm_generic_payload
) extends uvm_tlm_b_initiator_socket_base #(T)
```

12.3.5.2.2 Methods

uvm_tlm_b_initiator_socket has the following methods (see 12.3.5.2.3 to 12.3.5.2.4).

12.3.5.2.3 new

```
function new (
   string name,
   uvm_component parent,
)
```

Constructs a new instance of this socket.

12.3.5.2.4 connect

```
function void connect(uvm_tlm_b_initiator_socket provider)
```

Connects this socket to the specified **uvm_tlm_b_target_socket** (see 12.3.5.1).

12.3.5.3 uvm_tlm_nb_target_socket

`IS-A` forward imp; `HAS-A` backward port.

The component instantiating this socket shall implement a **nb_transport_fw** method (see 12.3.2.2.1) with the following signature:

```
function uvm_tlm_sync_e nb_transport_fw(T t, ref P p, input uvm_tlm_time
delay)
```

12.3.5.3.1 Class declaration

```
class uvm_tlm_nb_target_socket #(
   type IMP = int,
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_tlm_nb_target_socket_base#(T,P)
```

12.3.5.3.2 Methods

uvm_tlm_nb_target_socket has the following methods (see 12.3.5.3.3 to 12.3.5.3.4).

12.3.5.3.3 new

```
function new (
    string name,
    uvm_component parent,
    IMP imp = null
)
```

Constructs a new instance of this socket *imp*, a reference to the class implementing the **nb_transport_fw** method (see 12.3.2.2.1). If not specified, it is presumed to be the same as *parent*.

12.3.5.3.4 connect

```
function void connect(
    uvm_tlm_nb_target_socket provider
)
```

Connects this socket to the specified **uvm_tlm_nb_initiator_socket** (see 12.3.5.4).

12.3.5.4 uvm_tlm_nb_initiator_socket

`IS-A` forward port; `HAS-A` backward imp.

12.3.5.4.1 Class declaration

```
class uvm_tlm_nb_initiator_socket #(
    type IMP = int,
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e
) extends uvm_tlm_nb_initiator_socket_base#(T,P)
```

12.3.5.4.2 Methods

uvm_tlm_nb_initiator_socket has the following methods (see 12.3.5.4.3 to 12.3.5.4.4).

12.3.5.4.3 new

```
function new (
    string name,
    uvm_component parent,
    IMP imp = null
)
```

Constructs a new instance of this socket *imp* is a reference to the class implementing the **nb_transport_bw** method (see 12.3.4). If not specified, it is presumed to be the same as *parent*.

12.3.5.4.4 connect

```
function void connect(uvm_tlm_nb_initiator_socket provider)
```

Connects this socket to the specified **uvm_tlm_nb_target_socket** (see 12.3.5.3).

12.3.5.5 uvm_tlm_nb_passthrough_initiator_socket

IS-A forward port; HAS-A backward export.

Class declaration

```
class uvm_tlm_nb_passthrough_initiator_socket #(
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e
) extends uvm_tlm_nb_passthrough_initiator_socket_base#(T,P)
```

12.3.5.6 uvm_tlm_nb_passthrough_target_socket

IS-A forward export; HAS-A backward port.

12.3.5.6.1 Class declaration

```
class uvm_tlm_nb_passthrough_target_socket #(
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e
) extends uvm_tlm_nb_passthrough_target_socket_base#(T,P)
```

12.3.5.6.2 Methods

connect

```
function void connect(
    uvm_tlm_nb_initiator_socket provider
)
```

Connects this socket to the specified **uvm_tlm_nb_initiator_socket** (see 12.3.5.5).

12.3.5.7 uvm_tlm_b_passthrough_initiator_socket

IS-A forward port.

Class declaration

```
class uvm_tlm_b_passthrough_initiator_socket #(
    type T = uvm_tlm_generic_payload
) extends uvm_tlm_b_passthrough_initiator_socket_base#(T)
```

12.3.5.8 uvm_tlm_b_passthrough_target_socket

IS-A forward export.

Class declaration

```
class uvm_tlm_b_passthrough_target_socket #(
    type T = uvm_tlm_generic_payload
) extends uvm_tlm_b_passthrough_target_socket_base#(T)
```

12.3.6 Port classes

This subclause defines the UVM TLM 2 port classes.

12.3.6.1 uvm_tlm_b_transport_port

This class provides a blocking transport port, which can be bound to one export. There is no backward path for the blocking transport.

Class declaration

```
class uvm_tlm_b_transport_port #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

12.3.6.2 uvm_tlm_nb_transport_fw_port

This class provides a non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port, which can be bound to one export.

Class declaration

```
class uvm_tlm_nb_transport_fw_port #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.6.3 uvm_tlm_nb_transport_bw_port

This class provides a non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port, which can be bound to one export.

Class declaration

```
class uvm_tlm_nb_transport_bw_port #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.7 Export classes

This subclause defines the export classes for connecting UVM TLM 2 interfaces.

12.3.7.1 uvm_tlm_b_transport_export

This is a blocking transport export class.

Class declaration

```
class uvm_tlm_b_transport_export #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

12.3.7.2 uvm_tlm_nb_transport_fw_export

This is a non-blocking forward transport export class.

Class declaration

```
class uvm_tlm_nb_transport_fw_export #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.7.3 uvm_tlm_nb_transport_bw_export

This is a non-blocking backward transport export class.

Class declaration

```
class uvm_tlm_nb_transport_bw_export #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.8 Implementation (imp) classes imps

This subclause defines the implementation classes for connecting UVM TLM 2 interfaces.

UVM TLM 2 imps bind a UVM TLM 2 interface with the object that contains the interface implementation. In addition to the transaction type and the phase type, the imps are parameterized with the type of the object that provides the implementation. Typically, this is the type of the component where the imp resides. The constructor of the imp takes as an argument an object of type *IMP* and installs it as the implementation object. The imp constructor argument is usually "`this`".

The following subclauses show the IMP binding classes.

12.3.8.1 uvm_tlm_b_transport_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated, the implementation object is bound.

Class declaration

```
class uvm_tlm_b_transport_imp #(
   type T = uvm_tlm_generic_payload,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T))
```

12.3.8.2 uvm_tlm_nb_transport_fw_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated, the implementation object is bound.

Class declaration

```
class uvm_tlm_nb_transport_fw_imp #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.8.3 uvm_tlm_nb_transport_bw_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated, the implementation object is bound.

Class declaration

```
class uvm_tlm_nb_transport_bw_imp #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

12.3.9 uvm_tlm_time

```
typedef uvm_time uvm_tlm_time
```

The `uvm_tlm_time` type is the argument type used to represent *delays* in UVM TLM 2, such as in the **b_transport** (see 12.3.2.2.3), **nb_transport_fw** (see 12.3.2.2.1), and **nb_transport_bw** (see 12.3.2.2.1) methods.

13. Predefined component classes

Components form the foundation of UVM. They encapsulate 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 13.1).

13.1 uvm_component

The **uvm_component** class is the common base class for UVM components. In addition to the features inherited from **uvm_object** (see 5.3) and **uvm_report_object** (see 6.3), **uvm_component** provides the following interfaces:

a) *Hierarchy*—provides methods for searching and traversing the component hierarchy.

b) *Phasing*—defines a phased test flow that all components follow, with a group of standard phase methods and an API for custom phases and multiple independent phasing domains to mirror DUT behavior, e.g., power.

c) *Hierarchical reporting*—provides a convenience interface to the **uvm_report_handler** (see 6.4). All messages, warnings, and errors are processed through this interface.

d) *Transaction recording*—provides methods for recording the transactions produced or consumed by the component to a transaction database (application specific).

e) *Factory*—provides a convenience interface (see D.2.1) to the **uvm_factory** (see 8.3.1). The factory is used to create new components and other objects based on type-wide and instance-specific configuration.

uvm_component is automatically seeded during construction using UVM seeding, if enabled. See 5.3.3.3.

13.1.1 Class declaration

```
virtual class uvm_component extends uvm_report_object
```

13.1.2 Common methods

13.1.2.1 new

```
function new (
   string name,
   uvm_component parent
)
```

Initializes a new component with the given leaf instance *name* and handle to its *parent*.

The *name* shall be provided such that the full hierarchical name is unique and the leaf name is composed only from the characters **A** through **Z**, **a** through **z**, **0** through **9** or the special characters: **_ - [ ] ( ) { }**.

The component is inserted as a child of the *parent* object, if any. If *parent* already has a child by the given *name*, an error shall be generated.

If *parent* is *null*, the component shall become a child of the implicit top-level component (see F.7).

All classes derived from **uvm_component** shall call `super.new(`*`name`*`,`*`parent`*`)`.

13.1.2.2 print_enabled

```
bit print_enabled = 1
```

This bit determines if this component should automatically be printed as a child of its parent object.

By default, *print_enabled* shall be `1` and all children are printed. However, this bit allows a parent component to disable the printing of specific children.

13.1.3 Hierarchy interface

These methods provide user access to information about the component hierarchy, i.e., topology.

13.1.3.1 get_parent

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

Returns a handle to this component's parent or *null* if it has no parent.

13.1.3.2 get_full_name

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

Returns the full hierarchical name of this object, which is formed by concatenating the full hierarchical name of the parent, if any, with the leaf name of this object [as given by **uvm_object::get_name** (see 5.3.4.2)], separated by a period (**.**).

13.1.3.3 get_children

```
function void get_children(
   ref uvm_component children[$]
)
```

This function populates the end of the *children* array with the list of this component's children. *children* shall be a queue.

13.1.3.4 get_child, get_next_child, and get_first_child

```
function uvm_component get_child (
   string name
)

function int get_next_child (
   ref string name
)

function int get_first_child (
   ref string name
)
```

These methods are used to iterate through this component's children, if any.

a) **get_child**—Returns a reference to the child which has *name*. If no child exists with the given *name*, then *null* is returned.

b) **get_first_child**—Iteration method for the internal array of children components. If the array is non-empty, **get_first_child** sets *name* to the name of the first child in the array and returns `1`. If the array is empty, *name* is left unchanged and the method returns `0`.

c) **get_next_child**—Iteration method for the internal array of children components. If the array is non-empty, **get_next_child** sets *name* to the name of the next child in the array and returns `1`. If there are no more children in the array, *name* is left unchanged and the method returns `0`.

13.1.3.5 get_num_children

```
function int get_num_children()
```

Returns the number of this component's children.

13.1.3.6 has_child

```
function int has_child (
    string name
)
```

Returns `1` if this component has a child with the given *name*, `0` otherwise.

13.1.3.7 lookup

```
function uvm_component lookup (
    string name
)
```

Looks for a component with the given hierarchical *name* relative to this component. If the given *name* is preceded with a `.` (dot), the search begins relative to the top level (absolute lookup). The handle of the matching component is returned, if none, *null* is returned. The name shall not contain wild cards.

13.1.3.8 get_depth

```
function int unsigned get_depth()
```

Returns the component's depth from the root level. The implicit top-level component (see F.7) has a depth of `0`. The test and any other top-level components have a depth of `1`, and so on.

13.1.4 Phasing interface

These methods implement an interface that allows all components to step through a standard schedule of phases (see Clause 9) or a customized schedule, and also an API to allow independent phase domains that can jump like state machines to reflect behavior, e.g., power domains on the DUT in different portions of the testbench. The phase tasks and functions are the phase name plus the `_phase` suffix, e.g., the build phase function is `build_phase`.

All phase tasks have the property that forked tasks are killed when the phase ends and they do not influence the overall phase with the presence or absence of returning.

13.1.4.1 UVM common phases

13.1.4.1.1 build_phase

```
virtual function void build_phase(
   uvm_phase phase
)
```

The **uvm_build_phase** phase implementation method (see 9.8.1.1).

If automatic configuration is enabled (see 13.1.5.2), the component shall call **apply_config_settings** (see 13.1.5.1) when `super.build_phase(`*`phase`*`)` is called.

13.1.4.1.2 connect_phase

```
virtual function void connect_phase(
   uvm_phase phase
)
```

The **uvm_connect_phase** phase implementation method (see 9.8.1.2).

13.1.4.1.3 end_of_elaboration_phase

```
virtual function void end_of_elaboration_phase(
   uvm_phase phase
)
```

The **uvm_end_of_elaboration_phase** phase implementation method (see 9.8.1.3).

The list of connected imps within each port and export is populated and the port's minimum and maximum connection limits are enforced.

13.1.4.1.4 start_of_simulation_phase

```
virtual function void start_of_simulation_phase(
   uvm_phase phase
)
```

The **uvm_start_of_simulation_phase** phase implementation method (see 9.8.1.4).

13.1.4.1.5 run_phase

```
virtual task run_phase(
   uvm_phase phase
)
```

The **uvm_run_phase** phase implementation method (see 9.8.1.5).

Whether this task returns or not does not indicate the end or persistence of this phase. Thus, the phase automatically ends once all objections are dropped using `phase.drop_objection`.

13.1.4.1.6 extract_phase

```
virtual function void extract_phase(
   uvm_phase phase
)
```

The **uvm_extract_phase** phase implementation method (see 9.8.1.6).

13.1.4.1.7 check_phase

```
virtual function void check_phase(
   uvm_phase phase
)
```

The **uvm_check_phase** phase implementation method (see 9.8.1.7).

13.1.4.1.8 report_phase

```
virtual function void report_phase(
   uvm_phase phase
)
```

The **uvm_report_phase** phase implementation method (see 9.8.1.8).

13.1.4.1.9 final_phase

```
virtual function void final_phase(
   uvm_phase phase
)
```

The **uvm_final_phase** phase implementation method (see 9.8.1.9).

13.1.4.2 UVM run-time phases

Whether each of these tasks returns or not does not indicate the end or persistence of the particular phase. It is necessary to raise an objection using `phase.raise_objection` to cause a phase to persist. Once all components have dropped their respective objection, via `phase.drop_objection`, or if no component raises an objection, the phase is ended.

All processes associated with a task-based phase are killed when the phase ends, see 9.6.

13.1.4.2.1 pre_reset_phase

```
virtual task pre_reset_phase(
   uvm_phase phase
)
```

The **uvm_pre_reset_phase** phase implementation method (see 9.8.2.1).

13.1.4.2.2 reset_phase

```
virtual task reset_phase(
   uvm_phase phase
)
```

The **uvm_reset_phase** phase implementation method (see 9.8.2.2).

13.1.4.2.3 post_reset_phase

```
virtual task post_reset_phase(
   uvm_phase phase
)
```

The **uvm_post_reset_phase** phase implementation method (see 9.8.2.3).

13.1.4.2.4 pre_configure_phase

```
virtual task pre_configure_phase(
   uvm_phase phase
)
```

The **uvm_pre_configure_phase** phase implementation method (see 9.8.2.4).

13.1.4.2.5 configure_phase

```
virtual task configure_phase(
   uvm_phase phase
)
```

The **uvm_configure_phase** phase implementation method (see 9.8.2.5).

13.1.4.2.6 post_configure_phase

```
virtual task post_configure_phase(
   uvm_phase phase
)
```

The **uvm_post_configure_phase** phase implementation method (see 9.8.2.6).

13.1.4.2.7 pre_main_phase

```
virtual task pre_main_phase(
   uvm_phase phase
)
```

The **uvm_pre_main_phase** phase implementation method (see 9.8.2.7).

13.1.4.2.8 main_phase

```
virtual task main_phase(
   uvm_phase phase
)
```

The **uvm_main_phase** phase implementation method (see 9.8.2.8).

13.1.4.2.9 post_main_phase

```
virtual task post_main_phase(
   uvm_phase phase
)
```

The **uvm_post_main_phase** phase implementation method (see 9.8.2.9).

13.1.4.2.10 pre_shutdown_phase

```
virtual task pre_shutdown_phase(
   uvm_phase phase
)
```

The **uvm_pre_shutdown_phase** phase implementation method (see 9.8.2.10).

13.1.4.2.11 shutdown_phase

```
virtual task shutdown_phase(
   uvm_phase phase
)
```

The **uvm_shutdown_phase** phase implementation method (see 9.8.2.11).

13.1.4.2.12 post_shutdown_phase

```
virtual task post_shutdown_phase(
   uvm_phase phase
)
```

The **uvm_post_shutdown_phase** phase implementation method (see 9.8.2.12).

13.1.4.3 phase_* methods

Any threads spawned in these callbacks are not affected when the phase ends.

13.1.4.3.1 phase_started

```
virtual function void phase_started(
   uvm_phase phase
)
```

Invoked at the start of each phase. The *phase* argument specifies the phase being started.

13.1.4.3.2 phase_ready_to_end

```
virtual function void phase_ready_to_end(
   uvm_phase phase
)
```

Invoked when all objections to ending the given *phase* and all sibling phases have been dropped, thus indicating that *phase* is ready to begin a clean exit. Sibling phases are phases who share any adjacent successor nodes (see 9.3.1.6.10).

Components needing to consume delta cycles or advance time to perform a clean exit from the phase may raise the phase's objection, e.g., `phase.raise_objection(this,"Reason")`. It is the responsibility of this component to drop the objection once it is ready for this phase to end (and processes killed). If no objection to the given *phase* or sibling phases are raised, the phase state (see 9.3.1.1.3) shall proceed to `UVM_PHASE_ENDED`. If any objection is raised, when all objections to ending the given *phase* and siblings are dropped, another iteration of **phase_ready_to_end** is called. To prevent endless iterations due to coding error, **phase_ended** (see 13.1.4.3.3) is called after any iterations returned by **phase.get_max_ready_to_end_iterations** (see 9.3.1.3.4) regardless of whether a previous iteration lead to any objections being raised.

13.1.4.3.3 phase_ended

```
virtual function void phase_ended(
   uvm_phase phase
)
```

Invoked at the end of each phase. The *phase* argument specifies the phase ending.

13.1.4.4 *_domain methods

13.1.4.4.1 set_domain

```
function void set_domain(
   uvm_domain domain,
   int hier = 1
)
```

Applies a phase domain to this component and, if *hier* is non-zero, recursively to all its children. The default domain (before **set_domain** is called) is the `uvm` domain (see 13.1.4.4.2). The default value of *hier* shall be `1`.

Calls the virtual **define_domain** method (see 13.1.4.4.3), which derived components can override to augment or replace the domain definition of its base class.

13.1.4.4.2 get_domain

```
function uvm_domain get_domain()
```

Returns a handle to the phase domain specified for this component.

13.1.4.4.3 define_domain

```
virtual protected function void define_domain(
   uvm_domain domain
)
```

Builds phase schedules into the provided domain handle. The default implementation adds a copy of the *uvm* phasing schedule to the given *domain*, if one does not already exist, and only if the *domain* is currently empty.

This method is called by **set_domain** (see 13.1.4.4.1), which integrators may use to specify this component belongs in a domain apart from the default '`uvm`' domain.

Custom component base classes requiring a custom phasing schedule can augment or replace the domain definition they inherit by overriding their `define_domain`.

Alternatively, the integrator can attempt to define the schedule by setting up a new domain and setting it onto the component, and the component can override that schedule by overriding this method.

13.1.4.5 Suspending and resuming a component

These tasks can be used to suspend and resume a component.

13.1.4.5.1 suspend

```
virtual task suspend()
```

Suspends this component.

This method needs to be implemented by the user to suspend the component according to the protocol and functionality it implements. A suspended component can be subsequently resumed using **resume** (see 13.1.4.5.2).

13.1.4.5.2 resume

```
virtual task resume()
```

Resumes this component.

This method needs to be implemented by the user to resume a component that was previously suspended using **suspend** (see 13.1.4.5.1).

13.1.4.6 pre_abort

```
virtual function void pre_abort()
```

This callback is executed when the message system is executing a **UVM_EXIT** action (see F.2.2.2). The exit action causes an immediate termination of the simulation, but the **pre_abort** callback hook gives components an opportunity to provide additional information to the user before the termination happens.

The **pre_abort** callback hooks are called in a bottom-up fashion.

13.1.5 Configuration interface

Components can be designed to be user-configurable in terms of their topology (the type and number of children it has), mode of operation, and run-time parameters (knobs). The configuration interface accommodates this common need, allowing a component's composition and state to be modified without having to derive new classes or new class hierarchies for every configuration scenario.

13.1.5.1 apply_config_settings

```
virtual function void apply_config_settings (
   bit verbose = 0
)
```

Searches for all configuration settings matching this component's instance path.

For each resource with a scope matching the return of **get_full_name** (see 13.1.3.2) as follows:

a) **do_execute_op** (see 5.3.13.1) is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_SET` and *rhs* set to the resource.

b) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the resource shall be passed to **set_local** (see 5.3.12).

When the *verbose* bit is set to `1`, all settings are printed as they are applied. If the component's **print_config_matches** property is specified (see 13.1.5.3), **apply_config_settings** is automatically called with `verbose = 1`. **apply_config_settings** can also be overloaded to customize automated configuration. The default value of *verbose* shall be `0` or not set.

If automatic configuration is enabled (see 13.1.5.2), this function is called by **uvm_component::build_phase** (see 13.1.4.1.1).

13.1.5.2 use_automatic_config

```
virtual function bit use_automatic_config()
```

Returns `1` if the component should call **apply_config_settings** in the `build_phase` (see 13.1.4.1.1); otherwise, returns `0`.

By default, **use_automatic_config** returns `1`. If the user wishes to disable the automatic call to **apply_config_settings** (see 13.1.5.1), this method needs to be overloaded to return a `0`.

When an extended component extends **use_automatic_config** and returns a `0`, wherein the base class returned `1`, the extended component is assuming responsibility for any configuration that would have occurred within the **apply_config_settings** call in the base class.

13.1.5.3 Objection interface

These methods provide component level hooks into the **uvm_objection** mechanism (see 10.5.1).

13.1.5.4 raised

```
virtual function void raised (
    uvm_objection objection,
    uvm_object source_obj,
    string description,
    int count
)
```

Hook called by the default implementation of **uvm_objection::raised** (see 10.5.1.4.1).

13.1.5.5 dropped

```
virtual function void dropped (
    uvm_objection objection,
    uvm_object source_obj,
    string description,
    int count
)
```

Hook called by the default implementation of **uvm_objection::dropped** (see 10.5.1.4.2).

13.1.5.6 all_dropped

```
virtual function void all_dropped (
    uvm_objection objection,
    uvm_object source_obj,
    string description,
    int count
)
```

Hook called by the default implementation of **uvm_objection::all_dropped** (see 10.5.1.4.3).

13.1.6 Recording interface

These methods comprise the component-based transaction recording interface (see also Clause 7). They can be used to record the transactions that this component "sees," i.e., produces or consumes.

13.1.6.1 accept_tr

```
function void accept_tr (
    uvm_transaction tr,
    time accept_time = 0
)
```

This function marks the acceptance of a transaction, *tr*, by this component. Specifically, it performs the following actions:

— Calls the *tr*'s **uvm_transaction::accept_tr** method (see 5.4.2.2), passing to it the *accept_time* argument. The default value of *accept_time* shall be `0`.

— Calls this component's **do_accept_tr** method (see 13.1.6.2) to allow for any post-begin action in derived classes.

— Triggers the component's `accept_tr` event if it has added such an event to the event pool. Any processes waiting on this event shall resume in the next delta cycle.

13.1.6.2 do_accept_tr

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

The **accept_tr** method (see 13.1.6.1) calls this function to accommodate any user-defined post-accept action.

13.1.6.3 begin_tr

```
function int begin_tr (
    uvm_transaction tr,
    string stream_name = "main",
    string label = "",
    string desc = "",
    time begin_time = 0,
    int parent_handle = 0
)
```

This function marks the start of a transaction, *tr*, by this component. Specifically, it performs the following actions:

a) Calls *tr*'s **uvm_transaction::begin_tr** method (see 5.4.2.4), passing to it the *begin_time* argument. *begin_time* should be greater than or equal to the accept time. When `begin_time = 0`, the current simulation time is used. The default value of *begin_time* shall be `0`.

   If recording is enabled, a new database transaction is started on the component's transaction stream given by the *stream* argument. No transaction properties are recorded at this time.

b) Calls the component's **do_begin_tr** method (see 13.2) to allow for any post-begin action in derived classes.

c) Triggers the component's internal `begin_tr` event if one was added to the pool.

A handle to the transaction is returned. The meaning of this handle, as well as the interpretation of the arguments *stream_name*, *label*, and *desc* are application specific. The default value of *stream_name* shall be `"main"`. The default value of *parent_handle* shall be `0`.

13.1.6.4 do_begin_tr

```
virtual protected function void do_begin_tr (
    uvm_transaction tr,
    string stream_name,
    int tr_handle
)
```

The **begin_tr** (see 13.1.6.3) and **begin_child_tr** (see 13.1.6.4) methods call this function to accommodate any user-defined post-begin action.

13.1.6.5 end_tr

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

This function marks the end of a transaction, *tr*, by this component. Specifically, it performs the following actions:

a) Calls *tr*'s **uvm_transaction::end_tr** method (see 5.4.2.7), passing to it the *end_time* and *free_handle* arguments. *end_time* shall be greater than the begin time. When `end_time = 0`, the current simulation time is used.
b) Calls the component's **do_end_tr** method (see 13.2) to accommodate any post-end action in derived classes.
c) The transaction's properties are recorded to the database transaction on which it was started and then the transaction is ended. Only those properties handled by the transaction's **do_record** method (and optional **`uvm_*_field** macros) are recorded.
d) Triggers the component's internal `end_tr` event if such an event has been added.

An implementation shall **free** (see 16.4.4.3) the recorder associated with the transaction if *free_handle* is set to `1`. If *free_handle* is set to `0`, the implementation shall **close** (see 16.4.4.2) the recorder, but not call **free**. The default value of *free_handle* shall be `1`.

13.1.6.6 do_end_tr

```
virtual protected function void do_end_tr (
    uvm_transaction tr,
    int tr_handle
)
```

The **end_tr** (see 13.1.6.5) method calls this function to accommodate any user-defined post-end action.

13.1.6.7 record_error_tr

```
function int record_error_tr (
    string stream_name = "main",
    uvm_object info = null,
    string label = "error_tr",
    string desc = "",
    time error_time = 0,
    bit keep_active = 0
)
```

This function marks an error transaction by a component. Properties of the given `uvm_object`, *info*, are recorded to the transaction database, as implemented in its **uvm_object::do_record** (see 5.3.7.2) and **uvm_object::do_execute_op** (see 5.3.13.1) methods.

An *error_time* of `0` indicates to use the current simulation time. The *keep_active* bit determines if the handle should remain active; if `0`, then a zero-length error transaction is recorded. The default value of *error_time* shall be `0`. The default value of *keep_active* shall be `0`.

A handle to the transaction is returned. The interpretation of this handle, as well as the strings *stream_name*, *label*, and *desc*, are application specific. The default value of *stream_name* shall be `"main"`. The default value of *label* shall be `"error_tr"`.

13.1.6.8 record_event_tr

```
function int record_event_tr (
    string stream_name = "main",
    uvm_object info = null,
    string label = "event_tr",
    string desc = "",
    time event_time = 0,
    bit keep_active = 0
)
```

This function marks an event transaction by a component.

An *event_time* of `0` indicates to use the current simulation time. The *keep_active* bit determines if the handle may be used for other application-specific purposes (`0` means no; `1` means yes). The default value of *event_time* shall be `0`. The default value of *keep_active* shall be `0`.

A handle to the transaction is returned. The interpretation of this handle, as well as the strings *stream_name*, *label*, and *desc*, are application specific. The default value of *stream_name* shall be `"main"`. The default value of *label* shall be `"event_tr"`.

13.1.6.9 get_tr_stream

```
virtual function uvm_tr_stream get_tr_stream(
    string name,
    string stream_type_name = ""
)
```

Returns a tr stream with *this* component's full name as a scope.

name is the name for the stream. *stream_type_name* is the type name for the stream [the default is an *empty string* (`""`)].

Streams that are retrieved via this method are stored internally, such that later calls to **get_tr_stream** return the same stream reference.

The stream can be removed from the internal storage via a call to **free_tr_stream** (see 13.1.6.10).

13.1.6.10 free_tr_stream

```
virtual function void free_tr_stream(
    uvm_tr_stream stream
)
```

Frees any internal references caused by a call to this API. This method is an indication by the user that the implementation should remove any references it has to *stream*.

The next call to **get_tr_stream** (see 13.1.6.9) results in a newly created **uvm_tr_stream** (see 7.2).

13.1.6.11 set_tr_database

```
virtual function void set_tr_database(uvm_tr_database db)
```

Specifies the **uvm_tr_database** object (see 7.1) to use for **begin_tr** (see 13.1.6.3) and other methods in this subclause (see 13.2). The default object is **uvm_coreservice_t::get_default_tr_database** (see F.4.1.4.6).

13.1.6.12 get_tr_database

```
virtual function uvm_tr_database get_tr_database()
```

Returns the **uvm_tr_database** object set by **set_tr_database** or the default (see 13.1.6.11).

13.1.7 Other interfaces

uvm_component also provides the following convenience interfaces:

a) *Factory*—provides a convenience interface (see D.2.1) to the **uvm_factory** (see 8.3.1).
b) *Hierarchical reporting*—provides a convenience interface (see D.2.2) to **set_report_*** methods in the **uvm_report_object** base class (see 6.3).

13.2 uvm_test

This class is the virtual base class for any user-defined tests; using it provides the ability to select which test to execute via the `UVM_TESTNAME` command line (see G.2.1) or as an argument to the **uvm_root::run_test** task (see F.7.2.1).

Deriving from **uvm_test** allows distinguishing tests from other component types that inherit from **uvm_component** (see 13.1) directly. Such tests also automatically inherit features that may be added to **uvm_test** in the future.

13.2.1 Class declaration

```
virtual class uvm_test extends uvm_component
```

13.2.2 Methods

new

```
function new (
   string name,
   uvm_component parent
)
```

Initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

13.3 uvm_env

This is the base class for hierarchical containers of other components that together comprise a complete environment. The environment may initially consist of the entire testbench. Later, it can be reused as a sub-environment in even larger system-level environments.

13.3.1 Class declaration

```
virtual class uvm_env extends uvm_component
```

13.3.2 Methods

new

```
function new (
   string name = "env",
   uvm_component parent = null
)
```

Initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

13.4 uvm_agent

The **uvm_agent** virtual class should be used as the base class for the user-defined agents. Deriving from **uvm_agent** enables distinguishing agents from other component types also using its inheritance.

13.4.1 Class declaration

```
virtual class uvm_agent extends uvm_component
```

13.4.2 Methods

13.4.2.1 new

```
function new (
   string name,
   uvm_component parent
)
```

Initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

13.4.2.2 get_is_active

```
virtual function uvm_active_passive_enum get_is_active()
```

Returns the *is_active* status for this agent (see F.2.1.7).

The configuration parameter *is_active* is used to identify whether this agent should be acting in active or passive mode. The parameter should be set as a `uvm_active_passive_enum`, however it additionally supports `uvm_config_int` and `uvm_config_string` to allow for command line overrides.

Example

```
+uvm_config_int=<path.to.agent>,is_active,0
+uvm_config_string=<path.to.agent>,is_active,UVM_ACTIVE
uvm_config_db#(uvm_active_passive_enum)::(this, "<relative.path.to.agent>",
    "is_active", UVM_ACTIVE)
```

The default implementation shall return `UVM_PASSIVE`, unless overridden via the configuration database. An agent developer may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

13.5 uvm_monitor

This class should be used as the base class for user-defined monitors.

Deriving from **uvm_monitor** allows distinguishing monitors from other component types inheriting from **uvm_component** (see 13.1). Such monitors automatically inherit features that may be added to **uvm_monitor** in the future.

13.5.1 Class declaration

```
virtual class uvm_monitor extends uvm_component
```

13.5.2 Methods

new

```
function new (
    string name,
    uvm_component parent
)
```

Initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

13.6 uvm_scoreboard

This class should be used as the base class for user-defined scoreboards.

Deriving from **uvm_scoreboard** allows distinguishing scoreboards from other component types inheriting from **uvm_component** (see 13.1). Such scoreboards automatically inherit features that may be added to **uvm_scoreboard** in the future.

13.6.1 Class declaration

```
virtual class uvm_scoreboard extends uvm_component
```

13.6.2 Methods

Use the **new** method as detailed in 13.5.2.

13.7 uvm_driver #(REQ,RSP)

This is the base class for drivers that initiate requests for new transactions via a **uvm_seq_item_pull_port** (see 15.2.2.1). The ports are typically connected to the exports of an appropriate sequencer component.

This driver operates in a pull mode. Its ports are typically connected to the corresponding exports in a pull sequencer as follows:

```
driver.seq_item_port.connect(sequencer.seq_item_export)
driver.rsp_port.connect(sequencer.rsp_export)
```

The **rsp_port** (see 13.7.2.2) only needs connecting if the driver uses it to write responses to the analysis export in the sequencer.

13.7.1 Class declaration

```
class uvm_driver #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_component
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

13.7.2 Ports

13.7.2.1 seq_item_port

Derived driver classes should use this port to request items from the sequencer. They may also use it to send responses back.

13.7.2.2 rsp_port

This port provides an alternate way of sending responses back to the originating sequencer. Which port to use depends on which export the sequencer provides for connection.

13.7.3 Methods

Use the **new** method as detailed in 13.5.2.

13.8 uvm_push_driver #(REQ,RSP)

This is the base class for a driver that passively receives transactions, i.e., it does not initiate requests for transactions. This is also known as *push mode*. Its ports are typically connected to the corresponding ports in a push sequencer as follows:

```
push_sequencer.req_port.connect(push_driver.req_export)
push_driver.rsp_port.connect(push_sequencer.rsp_export)
```

The **rsp_port** (see 13.8.2.2) only needs connecting if the driver uses it to write responses to the analysis export in the sequencer.

13.8.1 Class declaration

```
class uvm_push_driver #(
    type REQ = uvm_sequence_item,
    type RSP = REQ
) extends uvm_component
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

13.8.2 Ports

13.8.2.1 req_export

This export provides the blocking `put` interface whose default implementation produces an error. Derived drivers shall override `put` with an appropriate implementation (and not call `super.put`). Ports connected to this export shall supply the driver with transactions.

13.8.2.2 rsp_port

This analysis port is used to send response transactions back to the originating sequencer.

13.8.3 Methods

Use the **new** method as detailed in 13.5.2.

13.9 uvm_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection “subscribes” this component to any transactions emitted by the connected analysis port.

Subtypes of this class shall define the `write` method to process the incoming transactions.

13.9.1 Class declaration

```
virtual class uvm_subscriber #(
    type T = int
) extends uvm_component
```

13.9.2 Ports

analysis_export

```
uvm_analysis_imp #(T,
    uvm_subscriber#(T))
    analysis_export
```

This export provides access to the write method, which derived subscribers shall implement.

13.9.3 Methods

13.9.3.1 new

Use the **new** method as detailed in 13.5.2.

13.9.3.2 write

```
pure virtual function void write(
   T t
)
```

This is a pure virtual method that shall be defined in each subclass. Access to this method by outside components should be done via the **analysis_export** (see 13.9.2).

14. Sequences classes

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.

With **uvm_sequence** objects (see 14.3), users can encapsulate DUT initialization code, bus-based stress tests, network protocol stacks—anything procedural—then have them all execute in specific or random order to more quickly reach corner cases and coverage goals.

14.1 uvm_sequence_item

The base class for user-defined sequence items and also the base class for the **uvm_sequence** class (see 14.3). The **uvm_sequence_item** class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

14.1.1 Class declaration

```
class uvm_sequence_item extends uvm_transaction
```

14.1.2 Common fields

14.1.2.1 new

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

The constructor method for **uvm_sequence_item**.

14.1.2.2 set_item_context

```
function void set_item_context(
    uvm_sequence_base parent_seq,
    uvm_sequencer_base sequencer = null
)
```

Specifies the sequence and sequencer execution context for a sequence item.

14.1.2.3 get_use_sequence_info and set_use_sequence_info

```
function bit get_use_sequence_info()

function void set_use_sequence_info(
    bit value
)
```

These methods are used to specify and return the status of the `use_sequence_info` bit. When `use_sequence_info` is set to `1`, the sequence information is printed, copied, and recorded. When `use_sequence_info` has the default value of `0`, the sequence information is not used.

14.1.2.4 set_id_info

```
function void set_id_info(
   uvm_sequence_item item
)
```

Copies the internal sequence id, as well as the transaction id (see 5.4.2.19), from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

14.1.2.5 get_sequencer

```
function uvm_sequencer_base get_sequencer()
```

Returns a reference to the item's sequencer, as set by **set_sequencer** (see 14.1.2.6).

14.1.2.6 set_sequencer

```
virtual function void set_sequencer(
   uvm_sequencer_base sequencer
)
```

Specifies the sequencer for this item to *sequencer*. This takes effect immediately, so it shall not be called while the sequence is actively communicating with the sequencer.

14.1.2.7 get_parent_sequence

```
function uvm_sequence_base get_parent_sequence()
```

Returns a reference to the parent sequence of any `sequence_item` on which this method was called.

14.1.2.8 set_parent_sequence

```
function void set_parent_sequence(
   uvm_sequence_base parent
)
```

Specifies the parent sequence of this item. This is used to identify the source sequence of a item.

14.1.2.9 get_depth

```
function int get_depth()
```

Returns the value set by the most recent **set_depth** call (see 14.1.2.10) or if **set_depth** has never been called, `1` + the number of recursive calls to **get_parent_sequence** (see 14.1.2.7) that can be done without returning *null*. A root sequence has a depth of `1`, its child has a depth of `2`, and its grandchild has a depth of `3`.

14.1.2.10 set_depth

```
function void set_depth(
   int value
)
```

The depth of any sequence is calculated automatically. However, this method may be used to specify the depth of a particular sequence. This method overrides the automatically calculated depth, even if it is incorrect.

14.1.2.11 is_item

```
virtual function bit is_item()
```

This function may be called on any sequence_item or `sequence`. It returns `1` for items and `0` for sequences (which derive from this class).

14.1.2.12 get_root_sequence_name

```
function string get_root_sequence_name()
```

Provides the name of the root sequence (the top-most parent sequence).

14.1.2.13 get_root_sequence

```
function uvm_sequence_base get_root_sequence()
```

Provides a reference to the root sequence (the top-most parent sequence).

14.1.2.14 get_sequence_path

```
function string get_sequence_path()
```

Provides a string of names of each sequence in the full hierarchical path. A dot (`.`) is used as the separator between each sequence.

14.1.3 Reporting interface

Sequence items and sequences use the sequencer that they are associated with for reporting messages. If no sequencer has been specified for the item/sequence using **set_sequencer** (see 14.1.2.6) [or indirectly via **uvm_sequence_base::start_item** (see 14.2.6.2) or **uvm_sequence_base::start** (see 14.2.3.1)], then the global reporter is used.

14.1.3.1 uvm_get_report_object

```
function uvm_report_object uvm_get_report_object()
```

Returns the sequencer if non-*null*; otherwise, returns the implicit top-level component (see F.4.1.4.1).

14.1.3.2 uvm_report_enabled

```
function int uvm_report_enabled(
   int verbosity,
   uvm_severity severity = UVM_INFO,
   string id = ""
)
```

Returns `1` if the configured verbosity for the object returned from **uvm_get_report_object** (see 14.1.3.1) for this *severity*/*id* is greater than or equal to *verbosity*; otherwise, returns `0`. The default value of *severity* shall be `UVM_INFO`.

See also 6.3.4.1 and F.3.2.2.

14.1.3.3 uvm_report, uvm_report_info, uvm_report_warning, uvm_report_error, and uvm_report_fatal

```
virtual function void uvm_report(
   uvm_severity severity,
   string id,
   string message,
   int verbosity = (severity ==
      uvm_severity'(UVM_ERROR)) ?
         UVM_NONE : (severity ==
      uvm_severity'(UVM_FATAL)) ?
         UVM_NONE : UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

virtual function void uvm_report_info(
   string id,
   string message,
   int verbosity = UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

virtual function void uvm_report_warning(
   string id,
   string message,
   int verbosity = UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

virtual function void uvm_report_error(
   string id,
   string message,
   int verbosity = UVM_NONE,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

virtual function void uvm_report_fatal(
   string id,
   string message,
   int verbosity = UVM_NONE,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)
```

These are the primary reporting methods in UVM, implemented in this class. If **uvm_report_enabled** (see 14.1.3.2) returns a `0`, these return with no action. Otherwise, they create a new report message populated according the argument values and pass that message to **uvm_process_report_message** (see 14.1.3.4).

14.1.3.4 uvm_process_report_message

```
virtual function void uvm_process_report_message(
   uvm_report_message report_message
)
```

This method takes a preformed **uvm_report_message** (see 6.2), populates it with the report object from **get_report_object** (see 6.2.3.1), if the **uvm_report_message** context is an *empty string* (**""**) then sets it to the return from **get_sequence_path** (see 14.1.2.14), and finally passes it to the designated report handler for the report object for processing; see 6.4.7.

14.2 uvm_sequence_base

The **uvm_sequence_base** class provides the interfaces needed to create streams of sequence items and/or other sequences.

A sequence is executed by calling its **start** method (see 14.2.3.1), either directly or via invocation of any of the **`uvm_do_*** macros (see B.3).

14.2.1 Class declaration

```
virtual class uvm_sequence_base extends uvm_sequence_item
```

14.2.2 Common methods

14.2.2.1 new

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

The constructor for **uvm_sequence_base**.

14.2.2.2 get_randomize_enabled

```
virtual function bit get_randomize_enabled
```

Returns the current value of the randomize enabled bit, as set by **set_randomize_enabled** (see 14.2.2.3). If **set_randomize_enabled** has not been called, then returns 1.

14.2.2.3 set_randomize_enabled

```
virtual function void set_randomize_enabled (
   bit enable
)
```

Sets the value of the randomize enabled bit. When set to `1`, the sequence shall be randomized automatically before being executed by the **`uvm_do*** and **`uvm_rand_send*** macros (see B.3), or as a default sequence. When set to `0`, the sequence shall not be automatically randomized.

14.2.2.4 get_sequence_state

```
function uvm_sequence_state_enum get_sequence_state()
```

Returns the sequence state as an enumerated value.

14.2.2.5 wait_for_sequence_state

```
task wait_for_sequence_state(
    int unsigned state_mask
)
```

Waits until the sequence reaches one of the given states. *state_mask* can be a bitwise `OR`'ing of any of the sequence states. If the sequence is already in one of the states, this method returns immediately.

14.2.3 Sequence execution

14.2.3.1 start

```
virtual task start (
    uvm_sequencer_base sequencer,
    uvm_sequence_base parent_sequence = null,
    int this_priority = -1,
    bit call_pre_post = 1
)
```

Executes this sequence, returning when the sequence has completed.

The *sequencer* argument specifies the sequencer on which to run this sequence.

If *parent_sequence* is *null*, and no parent sequence has been assigned via **set_parent_sequence** (see 14.1.2.8), then this sequence is a root sequence; otherwise, it is a child of *parent_sequence*. When *parent_sequence* is not *null*, the *parent_sequence*'s **pre_do** (see 14.2.3.4), **mid_do** (see 14.2.3.5), and **post_do** (see 14.2.3.7) methods are called during the execution of this sequence.

By default, the priority of a sequence is the priority of its parent sequence. If it is a *root sequence*, i.e., **get_parent_sequence** (see 14.1.2.7) returns *null*, its default priority is `100`. To change this, use a non-negative value of *this_priority*. Higher numbers indicate a higher priority.

If *call_pre_post* is set to `1` (the default), the **pre_body** (see 14.2.3.3) and **post_body** (see 14.2.3.8) tasks are called before and after the sequence **body** (see 14.2.3.6) is called.

14.2.3.2 pre_start

```
virtual task pre_start()
```

This task is a user-definable hook that is called before the optional execution of **pre_body** (see 14.2.3.3).

14.2.3.3 pre_body

```
virtual task pre_body()
```

This task is a user-definable hook. The value of *call_pre_post* as passed into **start** (see 14.2.3.1) is in control.

14.2.3.4 pre_do

```
virtual task pre_do(
   bit is_item
)
```

This task is a user-definable hook task.

Although **pre_do** is a task, consuming simulation cycles may result in unexpected behavior on the driver.

14.2.3.5 mid_do

```
virtual function void mid_do(
   uvm_sequence_item this_item
)
```

This function is a user-definable hook function that is called after the sequence item has been randomized and just before the item is sent to the driver.

14.2.3.6 body

```
virtual task body()
```

This is the user-defined task where the main sequence code resides.

14.2.3.7 post_do

```
virtual function void post_do(
   uvm_sequence_item this_item
)
```

This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either the **item_done** (see 15.2.1.2.3) or **put** (see 15.2.1.2.8) methods.

14.2.3.8 post_body

```
virtual task post_body()
```

This task is a user-definable hook. The value of *call_pre_post* as passed into **start** (see 14.2.3.1) is in control.

14.2.3.9 post_start

```
virtual task post_start()
```

This task is a user-definable hook that is called after the optional execution of **post_body** (see 14.2.3.8).

14.2.4 Run-time phasing

14.2.4.1 get_starting_phase

```
function uvm_phase get_starting_phase()
```

Returns the ‘starting phase’.

If non-*null*, the starting phase specifies the phase in which this sequence was started. The starting phase is set automatically when this sequence is started as the default sequence on a sequencer.

14.2.4.2 set_starting_phase

```
function void set_starting_phase(
   uvm_phase phase
)
```

Specifies the ‘starting phase’.

14.2.4.3 get_automatic_phase_objection

```
function bit get_automatic_phase_objection()
```

Returns (and locks) the value of the ‘automatically object to starting phase’ bit.

If `1`, the sequence automatically raises an objection to the starting phase (if the starting phase is not *null*) immediately prior to **pre_start** (see 14.2.3.2) being called. The objection is dropped after **post_start** (see 14.2.3.9) has executed or **kill** (see 14.2.5.11) has been called.

14.2.4.4 set_automatic_phase_objection

```
function void set_automatic_phase_objection(
   bit value
)
```

Specifies the ‘automatically object to starting phase’ bit.

The most common interaction with the starting phase within a sequence is to simply raise the phase’s objection prior to executing the sequence and drop the objection after ending the sequence [either naturally or via a call to **kill** (see 14.2.5.11)]. To simplify this interaction for the user, UVM provides the ability to perform this functionality automatically.

NOTE—Do not set the automatic phase objection bit to `1` if a sequence runs with a `forever` loop inside of the body, as the objection will never get dropped.

14.2.5 Sequence control

14.2.5.1 get_priority

```
function int get_priority()
```

This function returns the current priority of the sequence.

14.2.5.2 set_priority

```
function void set_priority (
   int value
)
```

The priority of a sequence may be changed at any point in time. *value* shall be >= `1`. When the priority of a sequence is changed, the new priority is used by the sequencer the next time that it arbitrates between sequences.

The default priority *value* is the value of the sequence's parent's priority. The default priority for root sequences is `100`. Higher values result in higher priorities.

14.2.5.3 is_relevant

```
virtual function bit is_relevant()
```

The default **is_relevant** implementation returns `1`, indicating that the sequence is always relevant.

Users may choose to override this with their own virtual function to indicate to the sequencer that the sequence is not currently relevant after a request has been made.

When the sequencer arbitrates, it will call **is_relevant** on each requesting, unblocked sequence to see if it is relevant. If a `0` is returned, then the sequence is not used.

If all requesting sequences are not relevant, the sequencer will call **wait_for_relevant** (see 14.2.5.4) on all sequences and re-arbitrate upon its return.

Any sequence that implements **is_relevant** shall also implement **wait_for_relevant** so the sequencer has a way to wait for a sequence to become relevant.

14.2.5.4 wait_for_relevant

```
virtual task wait_for_relevant()
```

This method is called by the sequencer when all available sequences are not relevant. When **wait_for_relevant** returns, the sequencer attempts to re-arbitrate.

An implementation in a derived sequence should ensure **is_relevant** is `1` (see 14.2.5.3). If a sequence defines **is_relevant**, then the sequence shall also supply a **wait_for_relevant** method.

14.2.5.5 lock

```
task lock(
    uvm_sequencer_base sequencer = null
)
```

Requests a lock on the specified sequencer. When *sequencer* is *null*, the lock is requested on the current default sequencer (see 14.1.2.5). If *sequencer* is *null* and **get_sequencer** returns *null*, an implementation shall generate a fatal message.

A lock request is arbitrated the same as any other request. A lock is granted after all previously arbitrated requests are completed and no other locks or grabs are blocking this sequence.

The **lock** call returns once the lock has been granted.

14.2.5.6 grab

```
task grab(
    uvm_sequencer_base sequencer = null
)
```

Requests a lock on the specified sequencer. If no argument is supplied, the lock is requested on the current default sequencer (see 14.1.2.5). If *sequencer* is *null* and **get_sequencer** returns *null*, an implementation shall generate a fatal message.

A grab request is put in front of the arbitration queue and is arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The **grab** call returns once the grab has been granted.

14.2.5.7 unlock

```
function void unlock(
uvm_sequencer_base sequencer = null
)
```

Removes any locks obtained by this sequence on the specified sequencer. When *sequencer* is *null*, the unlock is done on the current default sequencer (see 14.1.2.5). If *sequencer* is *null* and **get_sequencer** returns *null*, an implementation shall generate a fatal message.

NOTE—The **unlock** method removes any locks acquired from both **lock** (see 14.2.5.5) and **grab** (see 14.2.5.6).

14.2.5.8 ungrab

```
function void ungrab(
   uvm_sequencer_base sequencer = null
)
```

Convenience method for calling **unlock** (see 14.2.5.7).

14.2.5.9 is_blocked

```
function bit is_blocked()
```

Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab. A `1` is returned if the sequence is currently blocked. A `0` is returned if no lock or grab prevents this sequence from executing.

14.2.5.10 has_lock

```
function bit has_lock()
```

Returns `1` if this sequence has a lock, `0` otherwise.

Note that even if this sequence does not have a lock, a child sequence may have a lock, in which case the sequence is still blocked from issuing operations on the sequencer.

14.2.5.11 kill

```
function void kill()
```

This function kills the sequence and causes all current locks, grabs, or requests in the sequence's default sequencer to be removed. The sequence state changes to `UVM_STOPPED`, and the **post_body** (see 14.2.3.8) and **post_start** (see 14.2.3.9) callback methods are executed.

14.2.5.12 do_kill

```
function void do_kill()
```

This function is a user hook that is called whenever a sequence is terminated by using `sequence.kill` or `sequencer.stop_sequences` (which effectively calls `sequence.kill`).

14.2.6 Sequence item execution

14.2.6.1 create_item

```
protected function uvm_sequence_item create_item(
    uvm_object_wrapper type_var,
    uvm_sequencer_base l_sequencer,
    string name
)
```

This creates and initializes a sequence_item or sequence using the factory. The item or sequence's sequencer is set to *l_sequencer* via **uvm_sequence_item::set_sequencer** (see 14.1.2.6).

14.2.6.2 start_item

```
virtual task start_item (
    uvm_sequence_item item,
    int set_priority = -1,
    uvm_sequencer_base sequencer = null
)
```

This is a convenience method for initiating the execution of a sequence *item* request on a *sequencer* with a specified *priority*.

a) If *item* is *null*, or if *item* can be cast to **uvm_sequence_base** (see 14.2) and **is_item** (see 14.1.2.11) returns `0`, then the implementation shall generate an error message and return immediately.

b) The *priority* is determined by evaluating the *set_priority* argument. If the argument is greater than or equal to `0`, then *priority* is the *set_priority* value; otherwise, the *priority* is set to the return value of **get_priority** (see 14.2.5.1).

c) The *sequencer* is determined by evaluating the *sequencer* argument.
   1) If the *sequencer* argument is not *null*, then that *sequencer* value is used.
   2) If the *sequencer* argument is *null* and the *item*'s **get_sequencer** (see 14.1.2.5) return value is not *null*, then that return value is used.
   3) If the *sequencer* argument is *null*, the *item*'s **get_sequencer** return value is *null*, and this sequence's **get_sequencer** return value is not *null*, then that return value is used.
   4) Otherwise, the implementation shall generate an error message and return immediately.

d) The implementation shall perform the following steps in order:
   1) The **set_item_context** method (see 14.1.2.2) on *item* shall be called, and passed this sequence as a *parent_seq* and *sequencer* as determined in Step c).
   2) The **wait_for_grant** method (see 15.3.2.6) shall be called on the *sequencer*, with this sequence as *sequence_ptr* and the *priority* as determined in Step b) as *item_priority*.
   3) The **pre_do** method (see 14.2.3.4) on this sequence is called, with *is_item* set to `1`.

NOTE—While not strictly required, **start_item** is usually paired with a corresponding **finish_item** (see 14.2.6.3) call.

14.2.6.3 finish_item

```
virtual task finish_item (
    uvm_sequence_item item,
)
```

This is a convenience method for completing the execution of a sequence item.

a) If *item* is *null*, or if *item* can be cast to **uvm_sequence_base** (see 14.2) and **is_item** (see 14.1.2.11) returns `0`, then the implementation shall generate an error message and return immediately.

b) If the item's **get_sequencer** (see 14.1.2.5) return value is *null*, then the implementation shall generate an error message and return immediately.

c) The implementation shall perform the following steps in order:

   1) The **mid_do** method (see 14.2.3.5) on this sequence is called, with *item* as *this_item*.

   2) The **send_request** method (see 14.2.6.5) on this sequence is called, with *item* as *request* and *rerandomize* set to `0`.

   3) The **wait_for_item_done** method (see 14.2.6.6) on this sequence is called, with the return value of *item*'s **get_transaction_id** method (see 5.4.2.19) as *transaction_id*.

   4) The **post_do** method (see 14.2.3.7) on this sequence is called, with *item* as *this_item*.

NOTE—**finish_item** returning indicates the driver has signaled that the item is done, either explicitly via **item_done** (see 15.2.1.2.3) or implicitly via a call to **get** (see 15.2.1.2.6); however, it does not strictly indicate that the driver has completed all processing of the request.

14.2.6.4 wait_for_grant

```
virtual task wait_for_grant(
    int item_priority = -1,
    bit lock_request = 0
)
```

This task calls **sequencer.wait_for_grant** (see 15.3.2.6) on the current sequencer (see 14.1.2.5).

When this method returns, the sequencer has granted the sequence, and the sequence shall call **send_request** (see 14.2.6.5) without inserting any simulation delay other than delta cycles.

14.2.6.5 send_request

```
virtual function void send_request(
    uvm_sequence_item request,
    bit rerandomize = 0
)
```

This function may only be called after a **wait_for_grant** call (see 14.2.6.4). **send_request** calls **sequencer.send_request** (see 15.3.2.20) on the current sequencer (see 14.1.2.5).

14.2.6.6 wait_for_item_done

```
virtual task wait_for_item_done(
    int transaction_id = -1
)
```

A sequence may optionally call **wait_for_item_done**. This task blocks until the driver indicates that the item is done, either explicitly via **item_done** (see 15.2.1.2.3) or implicitly via a call to **get** (see 15.2.1.2.6).

If no *transaction_id* parameter is specified, or if *transaction_id* is `-1`, the call returns the next time that the driver calls **item_done** or **get**. When a specific *transaction_id* is specified, the call returns when the driver indicates completion of that specific item.

NOTE—If a specific *transaction_id* has been specified and the driver has already signaled **item_done** for that transaction, then the call will hang waiting for that specific *transaction_id*. Additionally, **wait_for_item_done** returning indicates the driver has signaled the item is done; however, it does not strictly indicate the driver has completed all processing of the request.

14.2.7 Response API

14.2.7.1 use_response_handler

```
function void use_response_handler(
   bit enable
)
```

When called with *enable* set to `1`, responses are sent to the response handler. Otherwise, responses need to be retrieved using **get_response** (see 14.3.3.3).

By default, responses from the driver are retrieved in the sequence by calling **get_response**.

14.2.7.2 get_use_response_handler

```
function bit get_use_response_handler()
```

Returns the state of the **use_response_handler** bit (see 14.2.7.1).

14.2.7.3 response_handler

```
virtual function void response_handler(
   uvm_sequence_item response
)
```

When the **use_response_handler** bit is set to `1` (see 14.2.7.1), this virtual function is called by the sequencer for each response that arrives for this sequence. No action is taken by this function unless it is extended.

14.2.7.4 get_response_queue_error_report_enabled

```
function bit get_response_queue_error_report_enabled()
```

When this bit is `1` (the default value), error reports are generated when the response queue overflows. When this bit is `0`, no such error reports are generated.

14.2.7.5 set_response_queue_error_report_enabled

```
function void set_response_queue_error_report_enabled(
   bit value
)
```

By default, if the response queue overflows, an error shall be generated. The response queue overflows when more responses are sent to this from the driver than **get_response** calls (see 14.3.3.3) are made. Setting *value* to `0` disables these errors, while setting it to `1` enables them.

14.2.7.6 get_response_queue_depth

```
function int get_response_queue_depth()
```

Returns the current depth setting for the response queue.

14.2.7.7 set_response_queue_depth

```
function void set_response_queue_depth(
   int value
)
```

The default maximum depth of the response queue is `8`. This method is used to change the maximum depth of the response queue. An implementation shall generate an error message if the maximum depth is set to a value that is less than the current number of responses in the queue.

Setting the depth of the response queue to `-1` indicates an arbitrarily deep response queue and no checking is done.

14.2.7.8 clear_response_queue

```
virtual function void clear_response_queue()
```

Empties the response queue for this sequence.

14.3 uvm_sequence #(REQ,RSP)

The **uvm_sequence** class provides the interfaces necessary to create streams of sequence items and/or other sequences.

14.3.1 Class declaration

```
virtual class uvm_sequence #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_sequence_base
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

14.3.2 Variables

14.3.2.1 req

```
REQ req
```

The sequence contains a field of the request type called *req*. The user can use this field or create another field to use. The default `do_print` prints this field.

14.3.2.2 rsp

```
RSP rsp
```

The sequence contains a field of the response type called *rsp*. The user can use this field, if desired, or create another field to use. The default `do_print` prints this field.

14.3.3 Methods

14.3.3.1 new

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

Creates and initializes a new sequence object.

14.3.3.2 get_current_item

```
function REQ get_current_item()
```

Returns the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method returns *null*.

The sequencer is executing an item from the time that **get_next_item** (see 15.2.1.2.1) or **peek** (see 12.2.10) is called until the time that **get** (see 15.2.1.2.6) or **item_done** (see 15.2.1.2.3) is called.

14.3.3.3 get_response

```
virtual task get_response(
    output RSP response,
    input int transaction_id = -1
)
```

By default, sequences retrieve responses by calling **get_response**. If no *transaction_id* is specified, this task returns the next response sent to this sequence. If no response is available in the response queue, the method blocks until a response is received.

If a *transaction_id* parameter is specified, the task blocks until a response with that *transaction_id* is received in the response queue. `-1` indicates wait for the next response.

The **get_response** method needs to be called soon enough to avoid an overflow of the response queue to prevent responses from being dropped. See also 14.2.7.6.

If a response is dropped in the response queue, an error shall be generated unless the error reporting is disabled via **set_response_queue_error_report_enabled** (see 14.2.7.5).

14.4 uvm_sequence_library

The **uvm_sequence_library** is a sequence that contains a list of registered sequence types. It can be configured to create and execute these sequences any number of times using one of several modes of operation (see F.2.4.3), including a user-defined mode.

When started (as any other sequence), the sequence library randomly selects and executes a sequence from its sequences queue. If in **UVM_SEQ_LIB_RAND** mode (see F.2.4.3), its **select_rand** property (see 14.4.4.7) is randomized and used as an index into sequences. When in **UVM_SEQ_LIB_RANDC** mode (see F.2.4.3), the **select_randc** property (see 14.4.4.8) is used. When in **UVM_SEQ_LIB_ITEM** mode (see F.2.4.3), only sequence items of the *REQ* type are generated and executed—no sequences are executed. Finally, when in **UVM_SEQ_LIB_USER** mode (see F.2.4.3), the **select_sequence** method (see 14.4.4.9) is called to obtain the index for selecting the next sequence to start. Users can override this method in subtypes to implement custom selection algorithms.

Creating a subtype of a sequence library requires invocation of the **`uvm_sequence_library_utils** macro (see B.3.2.2) in its declaration. The macro is needed to populate the sequence library with any sequences that were statically registered with it or any of its base classes.

14.4.1 Class declaration

```
class uvm_sequence_library#(
    type REQ = uvm_sequence_item,
    RSP = REQ
extends uvm_sequence #(REQ,RSP)
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

14.4.2 Example

```
class my_seq_lib extends uvm_sequence_library #(my_item)
    `uvm_object_utils(my_seq_lib)
    `uvm_sequence_library_utils(my_seq_lib)
    function new(string name="")
        super.new(name)
    endfunction
    ...
endclass
```

14.4.3 Common methods

new

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

Creates a new instance of this class.

14.4.4 Sequence selection

14.4.4.1 selection_mode

```
virtual function uvm_sequence_lib_mode get_selection_mode()

virtual function void set_selection_mode(
    uvm_sequence_lib_mode m
)
```

Specifies the mode used to select sequences for execution. If **set_selection_mode** has not been called since the sequence library was constructed, then **get_selection_mode** shall return `UVM_SEQ_LIB_RAND`.

14.4.4.2 min_random_count

```
virtual function int unsigned get_min_random_count()

virtual function void set_min_random_count(
    int unsigned count
)
```

Specifies the minimum number of sequences to execute. If **set_min_random_count** has not been called since the sequence library was constructed, then **get_min_random_count** shall return `10`.

14.4.4.3 max_random_count

```
virtual function int unsigned get_max_random_count()

virtual function void set_max_random_count(
   int unsigned count
)
```

Specifies the maximum number of sequences to execute. If **set_max_random_count** has not been called since the sequence library was constructed, then **get_max_random_count** shall return `10`.

14.4.4.4 get_num_sequences_executed

```
function int unsigned get_num_sequences_executed()
```

Returns the number of sequences executed, not including the currently executing sequence, if any.

14.4.4.5 sequences_executed

```
protected int unsigned sequences_executed
```

Indicates the number of sequences executed, not including the currently executing sequence, if any.

14.4.4.6 sequence_count

```
rand int unsigned sequence_count = 10
```

Specifies the number of sequences to execute when this sequence library is started. This value is constrained to be inside `{[min_random_count:max_random_count]}`. If in **UVM_SEQ_LIB_ITEM** mode (see F.2.4.3), specifies the number of sequence items to generate.

14.4.4.7 select_rand

```
rand int unsigned select_rand
```

This is the index variable that is randomized to select the next sequence to execute when in **UVM_SEQ_LIB_RAND** mode (see F.2.4.3). This variable is constrained to be a valid index into the array of registered sequences.

Extensions may place additional constraints on this variable.

14.4.4.8 select_randc

```
randc bit [15:0] select_randc
```

This is the index variable that is randomized to select the next sequence to execute when in **UVM_SEQ_LIB_RANDC** mode (see F.2.4.3). This variable is constrained to be a valid index into the array of registered sequences.

Extensions may place additional constraints on this variable.

14.4.4.9 select_sequence

```
virtual function int unsigned select_sequence(
   int unsigned max
)
```

Implementation hook that returns the index used to select the next sequence to execute when in **UVM_SEQ_LIB_USER** selection mode (see F.2.4.3). Overrides shall return a value between `0` and *max*, inclusive.

The default implementation returns `0`, incrementing on successive calls, wrapping back to `0` when reaching *max*. Extensions may use **get_sequence** (see 14.4.4.10) to assist in selecting an index.

14.4.4.10 get_sequence

```
virtual function uvm_object_wrapper get_sequence(
int unsigned idx
)
```

Returns the sequence registered with the sequence library at *idx*. An error message shall be generated if *idx* exceeds the current number of sequences registered with the sequence library and *null* shall be returned.

14.4.5 Sequence registration

14.4.5.1 add_typewide_sequence

```
static function void add typewide sequence(
   uvm_object_wrapper seq_type
)
```

Registers the provided sequence type with this sequence library type. The sequence type is available for selection by all instances of this class. Sequence types already registered are silently ignored. This method does not have any effect on sequence libraries that have already been constructed.

14.4.5.2 add_typewide_sequences

```
static function void add_typewide_sequences(
   uvm_object_wrapper seq_types[$]
)
```

Registers the provided sequence types with this sequence library type. The sequence types are available for selection by all instances of this class. Sequence types already registered are silently ignored. This method does not have any effect on sequence libraries that have already been constructed. *seq_types* shall be a queue.

14.4.5.3 add_sequence

```
function void add_sequence(
   uvm_object_wrapper seq_type
)
```

Registers the provided sequence type with this sequence library instance. Sequence types already registered are silently ignored.

14.4.5.4 add_sequences

```
virtual function void add_sequences(
   uvm_object_wrapper seq_types[$]
)
```

Registers the provided sequence types with this sequence library instance. Sequence types already registered are silently ignored. *seq_types* shall be a queue.

14.4.5.5 remove_sequence

```
virtual function void remove_sequence(
   uvm_object_wrapper seq_type
)
```

Removes the given sequence type from this sequence library instance. If the type was registered statically, the sequence queues of all instances of this sequence library are updated accordingly. A warning shallbe issued if the sequence is not registered.

14.4.5.6 get_sequences

```
virtual function void get_sequences(
   ref uvm_object_wrapper seq_types[$]
)
```

Appends to the provided *seq_types* array the list of registered *sequences*. *seq_types* shall be a queue.

15. Sequencer classes

15.1 Overview

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).

15.1.1 Sequencer variants

There are two sequencer variants available as follows:

a) **uvm_sequencer #(REQ,RSP)**—Requests for new sequence items are initiated by the driver (see 15.5). Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and delivers the next item to execute. This sequencer is typically connected to a user-extension of **uvm_driver #(REQ,RSP)** (see 13.7).

b) **uvm_push_sequencer #(REQ,RSP)**—Sequence items (from the currently running sequences) are pushed by the sequencer to the driver, which blocks item flow when it is not ready to accept new transactions (see 15.6). This sequencer is typically connected to a user-extension of **uvm_push_driver #(REQ,RSP)** (see 13.8).

Sequencer-driver communication follows a *pull* or *push* semantic, depending on which sequencer type is used. However, sequence-sequencer communication is always initiated by the user-defined sequence, i.e., it follows a *push* semantic.

See Clause 14 for an overview on sequences and sequence items.

15.1.2 Sequence item ports

The **uvm_sequencer #(REQ,RSP)** (see 15.5) and **uvm_driver #(REQ,RSP)** (see 13.7) pair uses a *sequence item pull port* (see 12.2.10) to achieve the special execution semantic needed by the sequencer-driver pair.

15.2 Sequencer interface

15.2.1 uvm_sqr_if_base #(T1,T2)

This class defines an interface for sequence drivers [**uvm_driver #(REQ,RSP)** (see 13.7)] to communicate with sequencers. The driver connects to the interface be via a port, and the sequencer implements it and provides it via an export.

This class provides timestamp properties, notification events, and transaction recording support. Its subtype, **uvm_sequence_item** (see 14.1), shall be used as the base class for all user-defined transaction types.

Note that **get_next_item/item_done** when called on a **uvm_seq_item_pull_port** (see 15.2.2.1) automatically triggers the event with the key `begin` or `end` in the event pool (see 5.4.2.14) via calls to **begin_tr** (see 13.1.6.3) and **end_tr** (see 13.1.6.5). While convenient, it is generally the responsibility of drivers to mark a transaction's progress during execution. To allow the driver or layering sequence to control sequence item timestamps, events, and recording, **uvm_sqr_if_base#(T1,T2)::disable_auto_item_recording** (see 15.2.1.2.10) needs to be called prior to the driver initiating its first request for an item.

Users can also use the transaction's event pool, **uvm_transaction::get_event_pool** (see 5.4.2.14), to define custom events for the driver to trigger and the sequences for waiting. Any in-between events, such as marking the beginning of the address and data phases of transaction execution, can be implemented via the event's pool.

In pipelined protocols, the driver can release a sequence [return from **finish_item** (see 14.2.6.3) or its `**uvm_do** macro (see B.3)] before the item has been completed. If the driver uses the **begin_tr**/**end_tr** API in **uvm_component** (see 13.1), the sequence can wait on the event at key `end` in the item's event pool (see 5.4.2.14) to block until the item was fully executed.

15.2.1.1 Class declaration

```
virtual class uvm_sqr_if_base #(
    type T1 = uvm_sequence_item,
        T2 = T1
)
```

The type of *T1* and *T2* shall be derived from **uvm_sequence_item** (see 14.1).

15.2.1.2 Methods

15.2.1.2.1 get_next_item

```
virtual task get_next_item(
    output T1 t
)
```

Retrieves the next available item from a sequence. The call shall block until an item is available. The following steps occur on this call:

a) Arbitrate among requesting, unlocked, and relevant sequences: choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
b) The chosen sequence returns from **uvm_sequence_base::wait_for_grant** (see 14.2.6.4).
c) The chosen sequence **uvm_sequence_base::pre_do** is called (see 14.2.3.4).
d) The chosen sequence item is randomized.
e) The chosen sequence **uvm_sequence_base::post_do** is called (see 14.2.3.7).
f) Return with a reference to the item.

Once **get_next_item** is called, **item_done** (see 15.2.1.2.3) needs to be called to indicate the completion of the request to the sequencer.

15.2.1.2.2 try_next_item

```
virtual task try_next_item(
    output T1 t
)
```

Retrieves the next available item from a sequence if one is available. Otherwise, the function returns immediately with request set to *null*. The following steps occur on this call:

a) Arbitrate among requesting, unlocked, and relevant sequences: choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return *null*.

b) The chosen sequence returns from **uvm_sequence_base::wait_for_grant** (see 14.2.6.4).

c) The chosen sequence **uvm_sequence_base::pre_do** is called (see 14.2.3.4).

d) The chosen sequence item is randomized.

e) The chosen sequence **uvm_sequence_base::post_do** is called (see 14.2.3.7).

f) Return with a reference to the item.

If **try_next_item** returns with a non-*null* request, **item_done** (see 15.2.1.2.3) needs to be called to indicate the completion of the request to the sequencer; this removes the request item from the sequencer FIFO.

15.2.1.2.3 item_done

```
virtual function void item_done(
    input T2 t = null
)
```

Indicates the request is completed to the sequencer. Any **uvm_sequence_base::wait_for_item_done** calls (see 14.2.6.6) made by a sequence for this item shall return.

If a response item is provided, it will be sent back to the requesting sequence. The response item shall have its sequence ID and transaction ID specified correctly, using the **uvm_sequence_item::set_id_info** method (see 14.1.2.4):

```
rsp.set_id_info(req)
```

Before **item_done** is called, any calls to **peek** (see 15.2.1.2.7) retrieve the current item that was obtained by **get_next_item** (see 15.2.1.2.1). After **item_done** is called, **peek** (see 15.2.1.2.7) causes the sequencer to arbitrate for a new item.

15.2.1.2.4 wait_for_sequences

```
virtual task wait_for_sequences()
```

Waits for a sequence to have a new item available. User-derived sequencers may override its **wait_for_sequences** implementation to perform some other application-specific implementation.

15.2.1.2.5 has_do_available

```
virtual function bit has_do_available()
```

Indicates whether a sequence item is available for immediate processing. Implementations shall return `1` if an item is available, `0` otherwise.

15.2.1.2.6 get

```
virtual task get(
    output T1 t
)
```

Retrieves the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call.

a) Arbitrate among requesting, unlocked, and relevant sequences: choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

b) The chosen sequence returns from **uvm_sequence_base::wait_for_grant** (see 14.2.6.4).

c) The chosen sequence **uvm_sequence_base::pre_do** is called (see 14.2.3.4).

d) The chosen sequence item is randomized.

e) The chosen sequence **uvm_sequence_base::post_do** is called (see 14.2.3.7).

f) Indicate **item_done** (see 15.2.1.2.3) to the sequencer.

g) Return with a reference to the item.

When **get** is called, **item_done** (see 15.2.1.2.3) may not be called. A new item can be obtained by calling **get** again, or a response shall be sent using either **put** (see 15.2.1.2.8) or **uvm_driver::rsp_port.write** (see 13.7.2.2).

15.2.1.2.7 peek

```
virtual task peek(
    output T1 t
)
```

Returns the current request item (see 15.3.3) if one is available. If no item is currently available, the following steps are performed in order:

a) Arbitrate among requesting, unlocked, and relevant sequences: choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

b) The chosen sequence returns from **uvm_sequence_base::wait_for_grant** (see 14.2.6.4).

c) The chosen sequence **uvm_sequence_base::pre_do** is called (see 14.2.3.4).

d) The chosen sequence item is randomized.

e) The chosen sequence **uvm_sequence_base::post_do** is called (see 14.2.3.7).

Once a request item has been retrieved, subsequent calls to **peek** return the same item until **get** (see 15.2.1.2.6) or **item_done** (see 15.2.1.2.3) is called.

15.2.1.2.8 put

```
virtual task put(
    output T2 t
)
```

Sends a response back to the sequence that issued the request. Before the response is `put`, it shall have its sequence ID and transaction ID specified to match the request. This can be done using the **uvm_sequence_item::set_id_info** call (see 14.1.2.4):

```
rsp.set_id_info(req)
```

While this is a task, it does not consume time (including delta cycles). The response is put into the sequence response queue or sent to the sequence response handler.

15.2.1.2.9 put_response

```
virtual function void put_response(
   output T2 t
)
```

Sends a response back to the sequence that issued the request. If the sequence has reached the UVM_STOPPED or UVM_FINISHED state (see F.2.4.2), then the implementation shall drop the response and issue a warning message. Before the response is put, it shall have its sequence ID and transaction ID specified to match the request. This can be done using the **uvm_sequence_item::set_id_info** call (see 14.1.2.4), e.g.,

```
rsp.set_id_info(req)
```

15.2.1.2.10 disable_auto_item_recording

```
virtual function void disable_auto_item_recording()
```

By default, item recording is performed automatically when **get_next_item** (see 15.2.1.2.1) and **item_done** (see 15.2.1.2.3) are called in the **uvm_sequencer#(REQ,RSP)** (see 15.5) or when the **uvm_push_sequencer#(REQ,RSP)** (see 15.6) puts an item on the **req_port** (see 15.6.217.5.2). However, this behavior might not be desired.

This automatic recording can be disabled by calling this function. Once disabled, automatic recording cannot be re-enabled. Automatic item recording can be globally turned off at compile time by defining UVM_DISABLE_RECORDING.

15.2.1.2.11 is_auto_item_recording_enabled

```
virtual function bit is_auto_item_recording_enabled()
```

Returns *TRUE* if automatic item recording is enabled for this port instance.

15.2.2 Sequence item pull ports

This defines the port, export, and imp port classes for communicating sequence items between **uvm_sequencer #(REQ,RSP)** (see 15.5) and **uvm_driver #(REQ,RSP)** (see 13.7).

15.2.2.1 uvm_seq_item_pull_port #(REQ,RSP)

UVM provides a port, export, and imp connector for use in sequencer-driver communication. All have standard port connector constructors.

Class declaration

```
class uvm_seq_item_pull_port #(
   type REQ = int,
   type RSP = REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.2.2.2 uvm_seq_item_pull_export #(REQ,RSP)

This export type is used in sequencer-driver communication. It has the standard constructor for exports.

Class declaration

```
class uvm_seq_item_pull_export #(
    type REQ = int,
    type RSP = REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.2.2.3 uvm_seq_item_pull_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication. It has the standard constructor for imp-type ports.

Class declaration

```
class uvm_seq_item_pull_imp #(
    type REQ = int,
    type RSP = REQ,
    type IMP = int
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.3 uvm_sequencer_base

Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

15.3.1 Class declaration

```
virtual class uvm_sequencer_base extends uvm_component
```

15.3.2 Methods

15.3.2.1 new

```
function new (
    string name,
    uvm_component parent
)
```

Creates and initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

15.3.2.2 is_child

```
function bit is_child (
    uvm_sequence_base parent,
    uvm_sequence_base child
)
```

Returns `1` if the *child* sequence is a child of the *parent* sequence, `0` otherwise.

15.3.2.3 user_priority_arbitration

```
virtual function int user_priority_arbitration(
    int avail_sequences[$]
)
```

When the sequencer arbitration mode is specified as `UVM_SEQ_ARB_USER` (using the **set_arbitration** method) (see 15.3.2.19), the sequencer calls this function each time it needs to arbitrate among sequences. *avail_sequences* shall be a queue.

Derived sequencers may override this method to perform a custom arbitration policy. The override shall return one of the entries from the *avail_sequences* queue, which contains the index of each sequence available for arbitration. The associated sequence for each index can be obtained via **get_arbitration_sequence** (see 15.3.2.4).

15.3.2.4 get_arbitration_sequence

```
virtual function uvm_sequence_base get_arbitration_sequence(
    int index
)
```

Returns the sequence available for the arbitration corresponding to *index*.

15.3.2.5 execute_item

```
virtual task execute_item(
    uvm_sequence_item item
)
```

Executes the given transaction item directly on this sequencer.

15.3.2.6 wait_for_grant

```
virtual task wait_for_grant(
    uvm_sequence_base sequence_ptr,
    int item_priority = -1,
    bit lock_request = 0
)
```

This task issues a request for the specified *sequence_ptr* sequence. If *item_priority* is not specified or is `-1`, the current sequence priority is used by the arbiter. This is the default condition and *item_priority* shall be `-1`. If a *lock_request* is made (`lock_request==1`), the sequencer issues a lock immediately before granting the sequence. [The lock may be granted without the sequence being granted if `is_relevant` (see 14.2.5.3) for the sequence is not asserted]. The default value of *lock_request* shall be `0`.

When this method returns, the sequencer has granted the sequence, and the sequence shall call **send_request** (see 15.3.2.20) without inserting any simulation delay other than delta cycles. The driver will be waiting for the next item to be sent via the **send_request** call.

15.3.2.7 wait_for_item_done

```
virtual task wait_for_item_done(
    uvm_sequence_base sequence_ptr,
    int transaction_id
)
```

A sequence may optionally call **wait_for_item_done**. This task blocks until the driver indicates that the item is done, either explicitly via **item_done** (see 15.2.1.2.3) or implicitly via a call to **get** (see 15.2.1.2.6) on a transaction issued by the specified *sequence_ptr* sequence. If no *transaction_id* parameter is specified, or the *transaction_id* is `-1`, the call returns the next time that the driver calls **item_done** or **get** on a transaction issued by the specified *sequence_ptr* sequence. When a specific *transaction_id* is used, the call only returns when the driver indicates it has completed that specific item.

NOTE—**wait_for_item_done** returning indicates the driver has signaled the item is done; however, it does not strictly indicate the driver has completed all processing of the request

15.3.2.8 is_blocked

```
function bit is_blocked(
    uvm_sequence_base sequence_ptr
)
```

Returns `1` if the sequence referred to by *sequence_ptr* is currently locked out of the sequencer. It returns `0` if the sequence is currently allowed to issue operations.

15.3.2.9 has_lock

```
function bit has_lock(
    uvm_sequence_base sequence_ptr
)
```

Returns `1` if the sequence referred to in by *sequence_ptr* currently has a lock on this sequencer, `0` otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer.

15.3.2.10 lock

```
virtual task lock(
    uvm_sequence_base sequence_ptr
)
```

Requests a lock for the sequence specified by *sequence_ptr*.

A lock request is arbitrated the same as any other request. A lock is granted after all previously granted requests are completed and no other locks or grabs are blocking this sequence.

The lock call returns once the lock has been granted.

15.3.2.11 grab

```
virtual task grab (
    uvm_sequence_base sequence_ptr
)
```

Requests a lock for the sequence specified by *sequence_ptr*.

A grab request is put in the front of the arbitration queue. It is arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call returns once the grab has been granted.

15.3.2.12 unlock

```
virtual task unlock(
   uvm_sequence_base sequence_ptr
)
```

Removes any locks and grabs obtained by the specified *sequence_ptr*.

15.3.2.13 ungrab

```
virtual task ungrab(
   uvm_sequence_base sequence_ptr
)
```

Calls **unlock** (see 15.3.2.12). Provided to give user code the symmetry of calling **grab** (see 15.3.2.11) and **ungrab**.

15.3.2.14 stop_sequences

```
virtual function void stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks, and responses that are currently queued. This essentially resets the sequencer to an idle state.

15.3.2.15 is_grabbed

```
virtual function bit is_grabbed()
```

Returns `1` if any sequence currently has a lock or grab on this sequencer, `0` otherwise.

15.3.2.16 current_grabber

```
virtual function uvm_sequence_base current_grabber()
```

Returns a reference to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, this returns the child that is currently allowed to perform operations on the sequencer.

15.3.2.17 has_do_available

```
virtual function bit has_do_available()
```

Returns `1` if any sequence running on this sequencer is ready to supply a transaction, `0` otherwise. A sequence is ready if it is not blocked [via **grab** (see 15.3.2.11) or **lock** (see 15.3.2.10)] and `is_relevant` (see 14.2.5.3) returns `1`.

15.3.2.18 get_arbitration

```
function UVM_SEQ_ARB_TYPE get_arbitration()
```

Returns the current arbitration mode for this sequencer. See 15.3.2.19 for a list of possible modes.

If the arbitration mode has not been set via a call to **set_arbitration** (see 15.3.2.19), then `UVM_SEQ_ARB_FIFO` shall be returned.

15.3.2.19 set_arbitration

```
function void set_arbitration(
    UVM_SEQ_ARB_TYPE val
)
```

Specifies the arbitration mode for the sequencer. *val* is one of

- `UVM_SEQ_ARB_FIFO`—Requests are granted in FIFO order (the default).
- `UVM_SEQ_ARB_WEIGHTED`—Requests are granted randomly by weight.
- `UVM_SEQ_ARB_RANDOM`—Requests are granted randomly.
- `UVM_SEQ_ARB_STRICT_FIFO`—Requests at highest priority granted in FIFO order.
- `UVM_SEQ_ARB_STRICT_RANDOM`—Requests at highest priority granted randomly.
- `UVM_SEQ_ARB_USER`—Arbitration is delegated to the user-defined function, `user_priority_arbitration`.

15.3.2.20 send_request

```
virtual function void send_request(
    uvm_sequence_base sequence_ptr,
    uvm_sequence_item t,
    bit rerandomize = 0
)
```

Derived classes shall implement this function to send a request item to the sequencer, which then forwards it to the driver. If the *rerandomize* bit is set to `1`, the item shall be randomized before being sent to the driver. The default value of *rerandomize* shall be `0`, which is not set.

This function may only be called after a **wait_for_grant** call (see 15.3.2.6).

15.3.2.21 set_max_zero_time_wait_relevant_count

```
virtual function void set_max_zero_time_wait_relevant_count(
    int new_val
)
```

Can be called at any time to change the maximum number of times `wait_for_relevant` can be called by the sequencer in zero time before an error is declared. The default maximum is `10`.

15.3.3 Requests

15.3.3.1 get_num_reqs_sent

```
function int get_num_reqs_sent()
```

Returns the number of requests that have been sent by this sequencer.

15.3.3.2 Last request buffer

15.3.3.2.1 set_num_last_reqs

```
function void set_num_last_reqs(
    int unsigned max
)
```

Specifies the size of the last requests buffer. The maximum buffer size is `1024`. If *max* is greater than `1024`, a warning shall be issued and the buffer is set to `1024`. The default value is `1`.

15.3.3.2.2 get_num_last_reqs

```
function int unsigned get_num_last_reqs()
```

Returns the size of the last requests buffer, as specified in **set_num_last_reqs** (see 15.3.3.2.1).

15.3.4 Responses

15.3.4.1 get_num_rsps_received

```
function int get_num_rsps_received()
```

Returns the number of responses received thus far by this sequencer.

15.3.4.2 Last response buffer

15.3.4.2.1 get_num_last_rsps

```
function int unsigned get_num_last_rsps()
```

Returns the maximum size of the last responses buffer, as specified in **set_num_last_rsps** (see 15.3.4.2.2).

15.3.4.2.2 set_num_last_rsps

```
function void set_num_last_rsps(
    int unsigned max
)
```

Specifies the size of the last response buffer. The maximum buffer size is `1024`. If *max* is greater than `1024`, a warning shall be issued and the buffer is set to `1024`. The default value is `1`.

15.3.5 Default sequence

A *default sequence* can be associated with a specified sequencer and **uvm_phase** (see 9.3.1).

A *default sequence* is specified via a **uvm_resource** (see Annex C for ways to set a resource or G.2.9 for the **uvm_set_default_sequence** command line option that sets the resource). The resource specifying a *default sequence* shall have a scope consisting of the concatenation `{"`*`path.to.sequencer`*`", "."," `*`phase_name`*`"}` (where *`path.to.sequencer`* stands for the sequencer's full name and *`phase_name`* stands for the name of the targeted phase), the name `"default_sequence"`, and the type **uvm_object_wrapper** (see 8.3.2) or **uvm_sequence_base** (see 14.2).

The *default sequence* for a sequencer/phase pair shall be selected as follows:

a) When the phase starts, **lookup_name** (see C.2.4.4.1) is called with the appropriate *scope* and *name* as described in the preceding paragraphs.
b) The result of **lookup_name** shall be filtered to remove any resources that are not of type **uvm_object_wrapper** (see 8.3.2) or **uvm_sequence_base** (see 14.2). The relative ordering of the remaining resources in the queue shall be maintained.
c) **get_highest_precedence** (see C.2.4.4.2) shall be called on the queue.

If the return value of **get_highest_precedence** is *null*, then no *default sequence* exists for the given sequencer/phase pair.

If **get_highest_precedence** (see C.2.4.4.2) returns a valid resource, however that resource contains a *null* value, then the *default sequence* for the given sequencer/phase pair has been explicitly disabled. How resources with non-*null* values are handled is dependent on the type of resource returned.

For resources of **uvm_object_wrapper** type (see 8.3.2), the object wrapper within the resource shall be passed to the **create_object_by_type** method (see 8.3.1.5) of the current factory (see F.4.1.4.2). The *parent_inst_path* shall be the full name of the sequencer and the *name* shall be the value returned by the wrapper's **get_type_name** method (see 8.3.2.2.3). If the object returned by **create_object_by_type** cannot be cast to a **uvm_sequence_base** (see 14.2), then an error message shall be generated and no *default sequence* shall be started for the given sequencer/phase pair. If the object can be cast into a **uvm_sequence_base**, then that sequence is the *default sequence* for this sequencer/phase pair.

For resources of **uvm_sequence_base** type, the sequence within the resource is the *default sequence* for this sequencer/phase pair.

The *default sequence* shall have its sequencer (see 14.1.2.6) and starting phase (see 14.2.4.2) set and be randomized [unless **get_randomize_enabled**returns `0` (see 14.2.2.2)], prior to being automatically started on the sequencer. If the *default sequence* is still running when the run-time phase ends, then it shall be killed.

15.4 Common sequencer API

This subclause describes the API implemented in both **uvm_sequencer#(REQ,RSP)** (see 15.5) and **uvm_push_sequencer#(REQ,RSP)** (see 15.6).

15.4.1 Method

get_current_item

```
function REQ get_current_item()
```

Returns the `request_item` currently being executed by the sequencer. If the sequencer is not currently executing an item, this method returns *null*.

The sequencer is executing an item from the time that **get_next_item** (see 15.2.1.2.1) or **peek** (see 12.2.10) is called until the time that **get** (see 15.2.1.2.6) or **item_done** (see 15.2.1.2.3) is called.

Note that a driver that only calls **get** does not show a current item, since the item is completed at the same time as it is requested.

15.4.2 Request

last_req

```
function REQ last_req(
   int unsigned n = 0
)
```

Returns the last request item by default, when *n* is `0`. If *n* is not `0`, it returns the n^{th}-before-last request item. If *n* is greater than the last request buffer size, the function returns *null*.

15.4.3 Responses

15.4.3.1 rsp_export

Drivers or monitors can connect to this port to send responses to the sequencer. Alternatively, a driver can send responses via its `seq_item_port`, e.g.,

```
seq_item_port.item_done(response)
seq_item_port.put(response)
rsp_port.write(response) <--- via this export
```

The **rsp_port** in the driver and/or monitor (see 13.7.2.2) needs to be connected to the **rsp_export** in this sequencer in order to send responses through the response analysis port.

15.4.3.2 last_rsp

```
function RSP last_rsp(
    int unsigned n = 0
)
```

Returns the last response item by default, when *n* is `0`. If *n* is not `0`, it returns the n^{th}-before-last response item. If *n* is greater than the last response buffer size, the function returns *null*.

15.5 uvm_sequencer #(REQ,RSP)

Requests for new sequence items are initiated by the driver. Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and delivers the next item to execute. This sequencer is typically connected to a user-extension of **uvm_driver #(REQ,RSP)** (see 13.7).

15.5.1 Class declaration

```
class uvm_sequencer #(
    type REQ = uvm_sequence_item,
    type RSP = REQ
) extends uvm_sequencer_base
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.5.2 Methods

15.5.2.1 Common sequencer API

uvm_sequencer #(REQ,RSP) implements all the API detailed in 15.4.

15.5.2.2 new

```
function new (
    string name,
    uvm_component parent
)
```

Creates and initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

15.5.2.3 seq_item_export

```
uvm_seq_item_pull_imp #(
    REQ,
    RSP,
    uvm_sequencer#(REQ, RSP)
) seq_item_export
```

This export provides access to this sequencer's implementation of the sequencer/driver interface (see 15.2.1).

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.6 uvm_push_sequencer #(REQ,RSP)

Sequence items (from the currently running sequences) are pushed by the sequencer to the driver. The driver blocks item flow when it is not ready to accept new transactions. This sequencer is typically connected to a user-extension of **uvm_push_driver #(REQ,RSP)** (see 13.8).

15.6.1 Class declaration

```
class uvm_push_sequencer #(
    type REQ = uvm_sequence_item,
    type RSP = REQ
) extends uvm_sequencer_base
```

The type of *REQ* and *RSP* shall be derived from **uvm_sequence_item** (see 14.1).

15.6.2 Ports

req_port

This is a `uvm_blocking_put_port #(REQ)` (see 12.2.2.3). The push sequencer requires access to a blocking put interface. A continuous stream of sequence items are sent out this port, based on the list of available sequences loaded into this sequencer.

15.6.3 Methods

15.6.3.1 Common sequence API

uvm_push_sequencer #(REQ,RSP) implements all the API detailed in 15.4.

15.6.3.2 new

```
function new (
    string name,
    uvm_component parent
)
```

Creates and initializes an instance of this class using the following constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any (see 13.1).

16. Policy classes

Each of UVM's policy classes perform a specific task for **uvm_object**-based objects (see 5.3): printing, comparing, recording, packing, and unpacking. They are implemented separately from **uvm_object** so that users can plug in different ways to print, compare, etc., without modifying the object class being utilized; the user can simply apply a different printer or compare "policy" to change how an object is printed or compared.

Each policy class includes several user-configurable parameters that control the operation. Users may also customize operations by deriving new policy subtypes from these base types. For example, UVM provides four different **uvm_printer**-based (see 16.2) policy classes, each of which print objects in a different format.

a) **uvm_printer** (see 16.2)—performs deep printing of **uvm_object**-based objects. UVM provides several subtypes to **uvm_printer** that print objects in a specific format: **uvm_table_printer** (see 16.2.10), **uvm_tree_printer** (see 16.2.11), and **uvm_line_printer** (see 16.2.12). Each such printer has many configuration options that govern what and how object members are printed.

b) **uvm_comparer** (see 16.3)—performs deep comparison of **uvm_object**-based objects. Users may configure what is compared and how miscompares are reported.

c) **uvm_recorder** (see 16.4.1)—performs the task of recording **uvm_object**-based objects to a transaction database. An implementation is application-specific.

d) **uvm_packer** (see 16.5)—used to pack and unpack **uvm_object**-based properties into arrays of type bit, byte, or int.

e) **uvm_copier** (see 16.6)—performs the task of copying fields of **uvm_object**-based objects.

16.1 uvm_policy

The abstract **uvm_policy** class provides a common base from which all UVM policy classes derive.

16.1.1 Class declaration

```
virtual class uvm_policy extends uvm_object
```

16.1.2 Methods

16.1.2.1 new

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

Creates a policy with the specified instance *name*. If *name* is not provided, then the policy instance is unnamed.

16.1.2.2 flush

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

The **flush** method resets the internal state of the policy, such that it can be reused.

16.1.2.3 Extensions

The policy extensions mechanism allows the user to pass additional information along with the policy class when executing a policy-based procedure. Objects may use these extensions to alter their interactions with the policy. For example: An object may use extensions to selectively filter some of its fields from being

processed. Policy extensions are not cleared via a call to the policy **flush** method (see 16.1.2.2), and need to be removed manually by using **clear_extension** (see 16.1.2.3.4) or **clear_extensions** (see 16.1.2.3.5).

16.1.2.3.1 extension_exists

```
virtual function bit extension_exists(
   uvm_object_wrapper ext_type
)
```

Returns `1` if an extension exists within the policy with type matching *ext_type*; otherwise, returns `0`.

16.1.2.3.2 set_extension

```
virtual function uvm_object set_extension(
   uvm_object extension
)
```

Sets the given *extension* inside of the policy, indexed using return value from *extension*'s **get_object_type** method (see 5.3.4.6). Only a single instance of an extension is stored per type. If there is an existing extension instance matching *extension*'s type, *extension* replaces the instance and the replaced instance handle is returned; otherwise, *null* is returned.

16.1.2.3.3 get_extension

```
virtual function uvm_object get_extension(
   uvm_object_wrapper ext_type
)
```

Returns the extension value stored within the policy with type matching *ext_type*. Returns *null* if no extension exists matching that type.

16.1.2.3.4 clear_extension

```
virtual function void clear_extension(
   uvm_object_wrapper ext_type
)
```

Removes the extension value stored within the policy matching type *ext_type*. If no extension exists matching type *ext_type*, the request is silently ignored.

16.1.2.3.5 clear_extensions

```
virtual function void clear_extensions()
```

Removes all extensions currently stored within the policy.

16.1.3 Active object

The active object methods are used to determine which object is actively being operated on by a policy. When a policy operates on an object, such as via **print_object** (see 16.2.3.1), **compare_object** (see 16.3.3.4), **record_object** (see 16.4.6.4), **pack_object** (see 16.5.4.2), **unpack_object** (16.5.4.4) or **copy_object** (see 16.6.4.1), it pushes the object onto the active object stack (see 16.1.3.1). When the policy completes the operation on the object, it pops the object off of the active object stack (see 16.1.3.2).

16.1.3.1 push_active_object

```
virtual function void push_active_object(
    uvm_object obj
)
```

Pushes *obj* on to the internal object stack for this policy, making it the current active object, as retrieved by **get_active_object** (see 16.1.3.3). An implementation shall generate an error message if *obj* is *null* and the request will be ignored.

Additionally, the policy shall push itself onto the active policy stack for *obj* using **push_active_policy** (see 5.3.14.1) when **push_active_object** is called.

16.1.3.2 pop_active_object

```
virtual function uvm_object pop_active_object()
```

Pops the current active object off of the internal object stack for this policy and returns the popped off value. If the internal object stack for this object is empty when **pop_active_object** is called, then *null* is returned and no error message is generated.

Additionally, the policy shall pop itself off of the active policy stack on *obj* using **pop_active_policy** (see 5.3.14.2) when **pop_active_object** is called.

16.1.3.3 get_active_object

```
virtual function uvm_object get_active_object()
```

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

16.1.3.4 get_active_object_depth

```
virtual function int unsigned get_active_object_depth()
```

Returns the current depth of the internal object stack for this policy.

16.1.4 recursion_state_e

An enum type that indicates whether a policy has operated on a given object or objects; it has the following values:

NEVER—The policy has never operated on the object(s).

STARTED —The policy has started operating on the object(s), but has not yet completed the operation.

FINISHED—The policy has completed operating on the object(s).

16.2 uvm_printer

The **uvm_printer** class provides an interface for printing **uvm_object**s (see 5.3) in various formats. Subtypes of **uvm_printer** implement different print formats or policies.

A user-defined printer format can be created or one of the following built-in printers can be used.

a) **uvm_printer**—provides base printer functionality; needs to be overridden.
b) **uvm_table_printer**—prints the object in a tabular form (see 16.2.10).
c) **uvm_tree_printer**—prints the object in a tree form (see 16.2.11).
d) **uvm_line_printer**—prints the information on a single line (see 16.2.12).

16.2.1 Class declaration

```
virtual class uvm_printer extends uvm_policy
```

16.2.2 Methods

16.2.2.1 new

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

Creates a new **uvm_printer** with the specified instance *name*. If *name* is not provided, the printer is unnamed.

16.2.2.2 set_default

```
static function void set_default (uvm_printer printer)
```

Helper method for setting the default printer policy instance via **uvm_coreservice_t::set_default_printer** (see F.4.1.4.12).

16.2.2.3 get_default

```
static function uvm_printer get_default()
```

Helper method for retrieving the default printer policy instance via **uvm_coreservice_t::get_default_printer** (see F.4.1.4.13).

16.2.3 Methods for printer usage

16.2.3.1 print_object

```
virtual function void print_object (
    string name,
    uvm_object value,
    byte scope_separator = "."
)
```

Prints an object.

name is the name to use when printing the object. Note that this may be different than the value returned by the object's **get_name** method (see 5.3.4.2).

value is the value of the object. *null* can be passed as a *value*.

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. The default value of *scope_separator* shall be **"."**.

Whether a non-*null value* is recursed depends on the settings for printer configuration (see 16.2.5). For objects that are being recursed, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).
b) The saved recursion state (see 16.1.4) for *value* and the current *recursion policy* (see 16.2.5.9) is set to `uvm_policy::STARTED`.
c) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_PRINT` and *policy* set to this printer.
d) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the printer passes itself the **do_print** method (see 5.3.6.3) on the object; otherwise, the method returns without calling **do_print**.
e) The saved recursion state (see 16.1.4) for *value* and the current *recursion policy* (see 16.2.5.9) is set to `uvm_policy::FINISHED`.
f) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

16.2.3.2 object_printed

```
virtual function uvm_policy::recursion_state_e object_printed(
    uvm_object value,
    uvm_recursion_policy_enum recursion
)
```

Returns the current recursion state (see 16.1.4) for *value* and *recursion* within the printer as defined by **print_object** (see 16.2.3.1). For objects that have never been passed to **print_object**, the return value shall be `uvm_policy::NEVER`.

16.2.3.3 print_generic

```
virtual function void print_generic (
    string name,
    string type_name,
    int size,
    string value,
    byte scope_separator = "."
)
```

Prints a field having the given *name*, *type_name*, *size*, *value*, and *scope_separator*.

name is the name of the field.

type_name is the variable type for the value being printed.

size is the size of the field.

value is the value of the field.

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (`.`) or an open bracket (`[`). The default value of *scope_separator* shall be `"."`.

16.2.3.4 print_generic_element

```
function void print_generic_element(
    string name,
    string type_name,
    string size,
    string value
)
```

An element is added as a child of the top element (see 16.2.7.2) with the given parameters.

This is a convenience mechanism, functionally identical to calling:

```
printer.push_element(name, type_name, size, value)
printer.pop_element()
```

16.2.3.5 print_array_header

```
virtual function void print_array_header(
    string name,
    int size,
    string arraytype = "array",
    byte scope_separator = "."
)
```

Prints the header of an array. This function is called before each individual element is printed. **print_array_footer** (see 16.2.3.7) is called to mark the completion of array printing. *arraytype* indicates what type of array is being printed. While it defaults to `"array"`, the user can set it to indicate queues or associative arrays.

The intent of *scope_separator* is to mark where the leaf name starts if the printer is configured to print only the leaf name of the identifier. The default value of *scope_separator* shall be `"."`.

16.2.3.6 print_array_range

```
virtual function void print_array_range (
    int min,
    int max
)
```

Prints a range using ellipses for values. This is used in honoring the partial printing of large arrays for **uvm_printer::set_begin_elements** and **uvm_printer:: set_end_elements** (see 16.2.6). *min* and *max* align to array indices. *min* should be the index of the first skipped value and *max* should be the index of the last skipped value.

This function should be called after printing the beginning of the array (as determined by **uvm_printer::get_begin_elements**) and before printing the end of the array (as determined by **uvm_printer::get_end_elements**).

16.2.3.7 print_array_footer

```
virtual function void print_array_footer (
    int size = 0
)
```

Prints the footer of an array. This function marks the end of an array print. Generally, there is no output associated with the array footer, but this method lets the printer know that the array printing is complete. The default value of *size* shall be `0`.

16.2.3.8 print_field

```
virtual function void print_field (
    string name,
    uvm_bitstream_t value,
    int size,
    uvm_radix_enum radix = UVM_NORADIX,
```

```
    byte scope_separator = ".",
    string type_name = ""
)
```

Prints an integral field.

name is the name of the field.

value is the value of the field.

size is the number of bits of the field [maximum is defined by `UVM_MAX_STREAMBITS (see B.6.2)].

radix is the radix to use for printing. The printer knob for radix is used if no radix is specified. The default value for *radix* shall be UVM_NORADIX (see F.2.1.5).

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (.) or an open bracket ([). The default value of *scope_separator* shall be ".".

type_name is the variable type for the value being printed; its default value is an *empty string* ("").

16.2.3.9 print_field_int

```
virtual function void print_field_int (
    string name,
    uvm_integral_t value,
    int size,
    uvm_radix_enum radix = UVM_NORADIX,
    byte scope_separator = ".",
    string type_name = ""
)
```

Prints an integral field (up to 64 bits).

name is the name of the field.

value is the value of the field.

size is the number of bits of the field.

radix is the radix to use for printing. The printer knob for *radix* is used if no *radix* is specified. The default value for *radix* shall be UVM_NORADIX (see F.2.1.5).

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (.) or an open bracket ([). The default value of *scope_separator* shall be ".".

type_name is the variable type for the value being printed; its default value is an *empty string* ("").

16.2.3.10 print_string

```
virtual function void print_string (
    string name,
    string value,
    byte scope_separator = "."
)
```

Prints a string field.

name is the name of the field.

value is the value of the field.

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (`.`) or an open bracket (`[`). The default value of *scope_separator* shall be `"."`.

16.2.3.11 print_time

```
virtual function void print_time (
    string name,
    time value,
    byte scope_separator = "."
)
```

Prints a time value.

name is the name of the field.

value is the value to print.

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (`.`) or an open bracket (`[`). The default value of *scope_separator* shall be `"."`.

16.2.3.12 print_real

```
virtual function void print_real (
    string name,
    real value,
    byte scope_separator = "."
)
```

Prints a real field.

name is the name of the field.

value is the value of the field.

scope_separator is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are a dot (`.`) or an open bracket (`[`). The default value of *scope_separator* shall be `"."`.

16.2.4 Methods for printer subtyping

16.2.4.1 emit

```
virtual function string emit()
```

Emits a string representing the contents of an object in a format defined by an extension of this object.

16.2.4.2 flush

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

The **flush** method resets the internal state of the printer. This includes clearing the internal element stack (see 16.2.7), and setting both the bottom element (see 16.2.7.1) and the top element (see 16.2.7.2) to *null*.

An implementation of the printer is allowed to reuse elements after a flush. Any outstanding handles to elements accessed prior to the flush via **get_bottom_element** (see 16.2.7.1) or **get_top_element** (see 16.2.7.2) shall be considered unstable after the flush.

16.2.5 Methods for printer configuration

The following methods define the printer settings available to all printer subtypes for use with the **print_*** methods (see 16.2.3) and **emit** (see 16.2.4.1).

16.2.5.1 Name enabled

```
virtual function void set_name_enabled (bit enabled)
virtual function bit get_name_enabled()
```

Controls whether field names shall be printed during **emit** (see 16.2.4.1). A value of `1` indicates field names shall be printed; a value of `0` indicates field names shall be omitted.

If **set_name_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_name_enabled** shall return `1`.

16.2.5.2 Type name enabled

```
virtual function void set_type_name_enabled (bit enabled)
virtual function bit get_type_name_enabled()
```

Controls whether field type names shall be printed during **emit** (see 16.2.4.1). A value of `1` indicates field type names shall be printed; a value of `0` indicates field type names shall be omitted.

If **set_type_name_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_type_name_enabled** shall return `1`.

16.2.5.3 Size enabled

```
virtual function void set_size_enabled (bit enabled)
virtual function bit get_size_enabled()
```

Controls whether field sizes shall be printed during **emit** (see 16.2.4.1). A value of `1` indicates field sizes shall be printed; a value of `0` indicates field sizes shall be omitted.

If **set_size_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_size_enabled** shall return `1`.

16.2.5.4 ID enabled

```
virtual function void set_id_enabled (bit enabled)
virtual function bit get_id_enabled()
```

Controls whether a unique reference ID shall be printed for object fields during **print_object** (see 16.2.3.1). A value of `1` indicates a unique reference ID shall be printed; a value of `0` indicates the unique reference ID shall be omitted.

If **set_id_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_id_enabled** shall return `1`.

16.2.5.5 Radix enabled

```
virtual function void set_radix_enabled (bit enabled)
virtual function bit get_radix_enabled()
```

Controls whether a *radix* string (see 16.2.5.6) shall be prepended to integral value fields **print_field** (see 16.2.3.8) or **print_field_int** (see 16.2.3.9). A value of `1` indicates the *radix* string shall be prepended; a value of `0` indicates the *radix* string shall be omitted.

If **set_radix_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_radix_enabled** shall return `1`.

16.2.5.6 Radix strings

```
virtual function void set_radix_string (uvm_radix_enum radix, string prefix)
virtual function string get_radix_string (uvm_radix_enum radix)
```

Controls the *prefix* strings used by **print_field** (see 16.2.3.8) or **print_field_int** (see 16.2.3.9) when **get_radix_enabled** (see 16.2.5.5) returns `1`.

If **set_radix_string** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then the *radix* for **get_radix_string** shall have a return value as shown in Table 2.

Table 2—radix return values

radix	Return value
`UVM_DEC`	`'d`
`UVM_BIN`	`'b`
`UVM_OCT`	`'o`
`UVM_UNSIGNED`	`'d`
`UVM_HEX`	`'h`

The return value for any *radix* value not shown in Table 2 is *undefined*.

16.2.5.7 Default radix

```
virtual function void set_default_radix (uvm_radix_enum radix)
virtual function uvm_radix_enum get_default_radix()
```

Controls the default *radix* used by **print_field** (see 16.2.3.8) or **print_field_int** (see 16.2.3.9) when **get_radix_enabled** (see 16.2.5.5) returns `1` and *radix* equals `UVM_NORADIX`.

If **set_default_radix** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_default_radix** shall return `UVM_HEX`.

16.2.5.8 Root enabled

```
virtual function void set_root_enabled (bit enabled)
virtual function bit get_root_enabled()
```

Controls whether **uvm_object::get_full_name** (see 5.3.4.3) or **uvm_object::get_name** (see 5.3.4.2) is used as the field name for the initial object being printed. A value of `1` indicates **uvm_object::get_full_name** shall be used; a value of `0` indicates **uvm_object::get_name** shall be used.

If **set_root_enabled** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_root_enabled** shall return `1`.

16.2.5.9 Recursion policy

```
virtual function void set_recursion_policy (uvm_recursion_policy_enum policy)
virtual function uvm_recursion_policy_enum get_recursion_policy()
```

Controls the recursion *policy* to use for object values supplied to **print_object** (see 16.2.3.1).

`UVM_DEEP`—Prints all fields of the target, doing a "deep" print (any object fields are printed using a DEEP recursion).

`UVM_SHALLOW`—Prints all fields of the target using a "shallow" print (any object fields are printed as REFERENCES).

`UVM_REFERENCE`—Prints the target as a reference.

A value of `UVM_DEFAULT_POLICY` shall be treated as `UVM_DEEP`.

If **set_recursion_policy** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_recursion_policy** shall return `UVM_DEFAULT_POLICY`.

16.2.5.10 Maximum depth

```
virtual function void set_max_depth (int depth)
virtual function int get_max_depth()
```

Controls the maximum recursion *depth* for objects printed via **print_object** (see 16.2.3.1). A maximum depth less than `0` indicates all objects shall be recursed using the current recursion policy (see 16.2.5.9); otherwise, for objects whose scope depth (see 16.1.3.4) exceeds the current maximum recursion depth, the printer shall print the object as if the *recursion policy* was `UVM_REFERENCE`.

If **set_max_depth** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_max_depth** shall return `-1`.

16.2.5.11 File

```
virtual function void set_file (UVM_FILE fl)
virtual function UVM_FILE get_file()
```

Controls *fl* [the current `UVM_FILE` (see F.2.8)], which specifies where the output of **uvm_object::print** (see 5.3.6.1) shall be directed.

If **set_file** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_file** shall return `UVM_STDOUT` (see F.2.9).

16.2.5.12 Line prefix

```
virtual function void set_line_prefix (string prefix)
virtual function string get_line_prefix()
```

Controls the string to use as a *prefix* to all lines of text generated by the printer during **emit** (see 16.2.4.1).

If **set_line_prefix** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_line_prefix** shall return an *empty string* (**""**).

16.2.6 Methods for object print control

The following methods define values that may be used in **do_print** (see 5.3.6.3) or **do_execute_op** (see 5.3.13.1) to control how fields are printed within an object.

Array elements

```
virtual function void set_begin_elements (int elements = 5)
virtual function void set_end_elements (int elements = 5)
virtual function int get_begin_elements()
virtual function int get_end_elements()
```

These options can control the number of elements at the beginning and end of an array to print. A value less than `0` for either *begin_elements* or *end_elements* indicates all elements of the array shall be printed. If both values are `0` or greater, then the object may omit array elements that are both:

— greater than or equal to *begin_elements* from the beginning of the array;

— greater than or equal to *end_elements* from the end of the array.

When omitting array elements in this fashion, **print_array_range** (see 16.2.3.6) shall be used to represent the skipped elements.

If **set_begin_elements** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_begin_elements** shall return `5`. If **set_end_elements** has not been called since the printer was constructed or since the last call to **flush**, then **get_end_elements** shall return `5`.

16.2.7 Element stack

Within the printer, a stack of **uvm_printer_element** (see 16.2.8) is maintained. Each element stores the *name*, *type_name*, *size*, and *value* to be printed as strings. The element stack allows for a separation between the content of the data being printed and the structure of the eventual output string. For example, an integral value that is printed looks the same regardless of whether it is printed in a tree structure or as a row in a tabular structure. The bottom of the stack represents the outer most layer of encapsulation being printed, while the top of the stack represents the layer of encapsulation currently being used.

When printing an individual field via **print_generic** (see 16.2.3.3), **print_generic_element** (see 16.2.3.4), **print_array_range** (see 16.2.3.6), **print_field** (see 16.2.3.8), **print_field_int** (see 16.2.3.9), **print_string** (see 16.2.3.10), **print_time** (see 16.2.3.11), and **print_real** (see 16.2.3.12), the printer shall push a new element onto the stack (see 16.2.7.3) and immediately pop the element from the stack (see 16.2.7.4).

When printing structured data via **print_object** (see 16.2.3.1), **print_array_header** (see 16.2.3.5), and **print_array_footer** (see 16.2.3.7), the printer shall push a new element onto the stack and eventually pop the element from the stack when the structure is done being printed, i.e., immediately before **print_object** returning or when **print_array_footer** is called.

16.2.7.1 get_bottom_element

```
protected virtual function uvm_printer_element get_bottom_element()
```

Returns the bottom element of the internal stack. This element represents the outermost layer of encapsulation being printed.

16.2.7.2 get_top_element

```
protected virtual function uvm_printer_element get_top_element()
```

Returns the top element of the internal stack. This element represents the layer of encapsulation currently being used.

16.2.7.3 push_element

```
virtual function void push_element(
    string name,
    string type_name,
    string size,
    string value=""
)
```

Pushes an element with the provided *name*, *type_name*, *size*, and *value* onto the top of the internal element stack, becoming the new return value for **get_top_element** (see 16.2.7.2). If the bottom element (see 16.2.7.1) is currently *null*, then the pushed element becomes the new bottom. If the top element (see 16.2.7.2) was not previously *null*, then the pushed element is a child of the previous top.

16.2.7.4 pop_element

```
virtual function void pop_element()
```

Pops the top element (see 16.2.7.2) off of the internal element stack, thereby restoring the next element on the stack to top.

If the top element on the stack is also the bottom element (see 16.2.7.1), then this request is silently ignored.

16.2.8 uvm_printer_element

This class is used by the **uvm_printer** (see 16.2) to represent the structure being printed in string form. The **uvm_printer::emit** method (see 16.2.4) is responsible for parsing the elements to produce properly formatted output.

16.2.8.1 Class declaration

```
class uvm_printer_element extends uvm_object
```

16.2.8.2 Methods

16.2.8.2.1 new

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

Initializes a new **uvm_printer_element** with the specified *name*. The default value of *name* shall be an *empty string* ("").

16.2.8.2.2 set

```
virtual function void set(
    string element_name = "",
    element_type_name = "",
    element_size = "",
```

```
    element_value = ""
)
```

Convenience method for setting the **element_name** (see 16.2.8.2.3), **element_type_name** (see 16.2.8.2.4), **element_size** (see 16.2.8.2.5), and **element_value** (see 16.2.8.2.6). The default value of all arguments shall be an *empty string* (**""**).

16.2.8.2.3 element_name

```
virtual function void set_element_name (string element_name)
  virtual function string get_element_name()
```

Controls the name of the element to be printed. The **get_element_name** method shall return *element_name*, as defined by the most recent call to **set** (see 16.2.8.2.2) or **set_element_name**. If **set** and **set_element_name** have not been called since this printer element was constructed, then **get_element_name** shall return an *empty string* (**""**).

16.2.8.2.4 element_type_name

```
virtual function void set_element_type_name (string element_type_name)
  virtual function string get_element_type_name()
```

Controls the type name associated with the element being printed. The **get_element_type_name** method shall return *element_type_name*, as defined by the most recent call to **set** (see 16.2.8.2.2) or **set_element_type_name**. If **set** and **set_element_type_name** have not been called since this printer element was constructed, then **get_element_type_name** shall return an *empty string* (**""**).

16.2.8.2.5 element_size

```
virtual function void set_element_size (string element_size)
  virtual function string get_element_size()
```

Controls the size of the element to be printed. The **get_element_size** method shall return *element_size*, as defined by the most recent call to **set** (see 16.2.8.2.2) or **set_element_size**. If **set** and **set_element_size** have not been called since this printer element was constructed, then **get_element_size** shall return an *empty string* (**""**).

16.2.8.2.6 element_value

```
virtual function void set_element_value (string element_value)
  virtual function string get_element_value()
```

Controls the value of the element to be printed. The **get_element_value** method shall return *element_value*, as defined by the most recent call to **set** (see 16.2.8.2.2) or **set_element_value**. If **set** and **set_element_value** have not been called since this printer element was constructed, then **get_element_value** shall return an *empty string* (**""**).

16.2.9 uvm_printer_element_proxy

This is a structural proxy class (see F.5.2) for the **uvm_printer_element** (see 16.2.8). It can be used to determine the children of an element.

uvm_printer_element (see 16.2.8) can be used to represent hierarchical elements, such as classes, arrays, structs, etc. Each field/value within the printed structure can be stored as additional children elements within

the parent, e.g., an array of depth `2` could be represented using three elements; the parent element representing the array itself, and then one child element per value within the array.

The representation of parent/child relationships within **uvm_printer_element** (see 16.2.8) is specific to an implementation of the **uvm_printer** (see 16.2), however the **uvm_printer_element_proxy** is provided as a standard mechanism for traversing the structure.

16.2.9.1 Class declaration

```
class uvm_printer_element_proxy extends uvm_structure_proxy#
    (uvm_printer_element)
```

16.2.9.2 Methods

16.2.9.2.1 new

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

Initializes a new **uvm_printer_element_proxy** with the specified *name*. The default value of *name* shall be an *empty string* (`""`).

16.2.9.2.2 get_immediate_children

```
virtual function void get_immediate_children(uvm_printer_element s,
                                             ref uvm_printer_element children[$])
```

This is an extension of the **uvm_structure_proxy**'s **get_immediate_children** method (see F.5.2). This method pushes the children elements of *s* to the back of the *children* queue. Any previously existing values within the *children* queue remain untouched.

16.2.10 uvm_table_printer

The table printer prints output in a tabular format.

16.2.10.1 Class declaration

```
class uvm_table_printer extends uvm_printer
```

16.2.10.2 Methods

16.2.10.2.1 new

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

Creates a new instance of **uvm_table_printer** with the specified instance *name*. If *name* is not provided, the printer is unnamed.

16.2.10.2.2 get_type_name

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

Returns the string `"uvm_table_printer"`.

16.2.10.2.3 set_default

```
static function void set_default (uvm_table_printer printer)
```

Overrides the default table printer instance *printer*, as retrieved by **get_default** (see 16.2.10.2.4).

16.2.10.2.4 get_default

```
static function uvm_table_printer get_default()
```

Retrieves the default table printer instance, as set by **set_default** (see 16.2.10.2.3). If **set_default** has not been called prior to the first **get_default** call, then the implementation shall instance a **uvm_table_printer** and pass that instance to **set_default** automatically.

16.2.10.3 Methods for printer configuration

The following method defines the printer settings available to table printers:

Indentation

```
virtual function void set_indent (int indent)
virtual function int get_indent()
```

Returns the number of spaces to use for indentation (*indent*) when printing the children of a **uvm_printer_element** (see 16.2.8).

If **set_indent** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_indent** shall return `2`.

16.2.11 uvm_tree_printer

The tree printer prints output in a tree format.

16.2.11.1 Class declaration

```
class uvm_tree_printer extends uvm_printer
```

16.2.11.2 Methods

16.2.11.2.1 new

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

Creates a new instance of **uvm_tree_printer** with the specified instance *name*. If *name* is not provided, the printer is unnamed.

16.2.11.2.2 get_type_name

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

Returns the string `"uvm_tree_printer"`.

16.2.11.2.3 set_default

```
static function void set_default (uvm_tree_printer printer)
```

Overrides the default tree *printer* instance, as retrieved by **get_default** (see 16.2.11.2.4).

16.2.11.2.4 get_default

```
static function uvm_tree_printer get_default()
```

Retrieves the default tree printer instance, as set by **set_default** (see 16.2.11.2.3). If **set_default** has not been called prior to the first **get_default** call, then the implementation shall instance a **uvm_tree_printer** and pass that instance to **set_default** automatically.

16.2.11.3 Methods for printer configuration

The following methods define the printer settings available to tree printers.

16.2.11.3.1 Indentation

```
virtual function void set_indent (int indent)
virtual function int get_indent()
```

Returns the number of spaces to use for indentation (*indent*) when printing the children of a **uvm_printer_element** (see 16.2.8).

If **set_indent** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_indent** shall return `2`.

16.2.11.3.2 Separators

```
virtual function void set_separators (string separators)
virtual function string get_separators()
```

Controls the *separators* used for printing the children of a **uvm_printer_element** (see 16.2.8). The first character of the string represents the opening separator and the second character represents the closing separator; all other characters are ignored.

If **set_separators** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_separators** shall return the curly brackets as string ("`{}`").

16.2.12 uvm_line_printer

The line printer prints output in a line format.

16.2.12.1 Class declaration

```
class uvm_line_printer extends uvm_printer
```

16.2.12.2 Methods

16.2.12.2.1 new

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

Creates a new instance of **uvm_line_printer** with the specified instance *name*. If *name* is not provided, the printer is unnamed.

16.2.12.2.2 get_type_name

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

Returns the string "uvm_line_printer".

16.2.12.2.3 set_default

```
static function void set_default (uvm_line_printer printer)
```

Overrides the default line *printer* instance, as retrieved by **get_default** (see 16.2.12.2.4).

16.2.12.2.4 get_default

```
static function uvm_line_printer get_default()
```

Retrieves the default line printer instance, as set by **set_default** (see 16.2.10.2.3). If **set_default** has not been called prior to the first **get_default** call, then the implementation shall instance a **uvm_line_printer** and pass that instance to **set_default** automatically.

16.2.12.3 Methods for printer configuration

The following method defines the printer settings available to line printers:

Separators

```
virtual function void set_separators (string separators)
virtual function string get_separators()
```

Controls the *separators* used for printing the children of a **uvm_printer_element** (see 16.2.8). The first character of the string represents the opening separator and the second character represents the closing separator; all other characters are ignored.

If **set_separators** has not been called since the printer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_separators** shall return the curly brackets as string ("{}").

16.3 uvm_comparer

The **uvm_comparer** class provides a policy object for doing comparisons. The policies determine how miscompares are treated and counted. Results of a comparison are stored in the comparer object. The **uvm_object::compare** (see 5.3.9.1) and **uvm_object::do_compare** (see 5.3.9.2) methods are passed a **uvm_comparer** policy object.

16.3.1 Class declaration

```
class uvm_comparer extends uvm_policy
```

16.3.2 Methods

16.3.2.1 new

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

Creates a new **uvm_comparer** with the specified instance *name*. If *name* is not provided, the object is unnamed.

16.3.2.2 flush

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

The **flush** method resets the internal state of the comparer. This includes setting the value returned by **get_result** (see 16.3.3.9) to `0` and setting the value returned by **get_miscompares** (see 16.3.3.8) to an *empty string* (`""`).

16.3.2.3 get_type_name

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

Returns the string `"uvm_comparer"`.

16.3.2.4 set_default

```
static function void set_default (uvm_comparer comparer)
```

Helper method for setting the default *comparer* policy instance via **uvm_coreservice_t::set_default_comparer** (see F.4.1.4.16).

16.3.2.5 get_default

```
static function uvm_comparer get_default()
```

Helper method for retrieving the default comparer policy instance via **uvm_coreservice_t::get_default_comparer** (see F.4.1.4.17).

16.3.3 Methods for comparer usage

16.3.3.1 compare_field

```
virtual function bit compare_field (
    string name,
    uvm_bitstream_t lhs,
    uvm_bitstream_t rhs,
    int size,
    uvm_radix_enum radix = UVM_NORADIX
)
```

Compares two integral values.

The *name* variable is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* variables are the two values used for comparison.

The *size* variable indicates the number of bits to compare; *size* shall be less than or equal to `UVM_MAX_STREAMBITS (see B.6.2).

The *radix* variable is used for the purposes of formatting the stored miscompare string.

16.3.3.2 compare_field_int

```
virtual function bit compare_field_int (
    string name,
    uvm_integral_t lhs,
    uvm_integral_t rhs,
    int size,
    uvm_radix_enum radix = UVM_NORADIX
)
```

Compares two **uvm_integral_t** values (see F.2.1.4).

The *name* variable is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* variables are the two values used for comparison.

The *size* variable indicates the number of bits to compare; size shall be less than or equal to 64.

The *radix* variable is used for the purposes of formatting the stored miscompare string.

16.3.3.3 compare_field_real

```
virtual function bit compare_field_real (
    string name,
    real lhs,
    real rhs
)
```

Compares two real values.

The *name* variable is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* variables are the two values used for comparison.

16.3.3.4 compare_object

```
virtual function bit compare_object (
    string name,
    uvm_object lhs,
    uvm_object rhs
)
```

Compares two class objects using the *recursion policy* (see 16.3.4.1) to determine whether the comparison should be deep, shallow, or reference.

The *name* input is used for purposes of storing and printing a miscompare.

The *lhs* and *rhs* objects are the two objects used for comparison.

For objects that are being compared, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).

b) The saved recursion state (see 16.1.4) for *lhs*, *rhs*, and the current *recursion policy* (see 16.3.4.1) is set to `uvm_policy::STARTED`.

c) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_COMPARE` and *policy* set to this comparer.

d) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the comparer passes itself and the *rhs* to the **do_-compare** method (see 5.3.9.2) on *lhs*.

e) The saved recursion state for *lhs*, *rhs*, and the current *recursion policy* is set to `uvm_policy::FINISHED`, and the saved return value for *lhs*, *rhs*, and the current *recursion policy* is set as follows:

   1) If the value of the **get_result** counter (see 16.3.3.9) increased during c) or d), then the value to return is set to `0`.

   2) If the **do_compare** call in d) returned `0`, then the valueto return is set to `0`.

   3) If the value of the **get_result** counter did not increase during c) or d), and the **do_compare** call in (d) returned `1`, then the value set to return is set to `1`.

f) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

g) **compare_object** returns the valuedetermined in e).

16.3.3.5 object_compared

```
virtual function uvm_policy::recursion_state_e object_compared(
    uvm_object lhs,
    uvm_object rhs,
    uvm_recursion_policy_enum recursion,
    output bit ret_val
)
```

Returns the current recursion state (see 16.1.4) for *lhs*, *rhs*, and *recursion* within the comparer as defined by **compare_object** (see 16.3.3.4). For objects that have never been passed to **compare_object**, the return value shall be `uvm_policy::NEVER`.

If the recursion state is `uvm_policy::FINISHED`, then *ret_val* is the return value of the comparison as defined by **compare_object**. If the recursion state is a value other than `uvm_policy::FINISHED`, then the value of *ret_val* is `0`.

The values passed to *lhs* and *rhs* need to be passed to **object_compared** using the same ordering as **compare_object**.

16.3.3.6 compare_string

```
virtual function bit compare_string (
    string name,
    string lhs,
    string rhs
)
```

Compares two string variables.

The *name* variable is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* variables are the two values used for comparison.

16.3.3.7 print_msg

```
function void print_msg (
    string msg
)
```

Causes the error count to be incremented and the message, *msg*, to be appended to the **get_miscompares** string (see 16.3.3.8). (A newline character is used to separate messages.)

If the message count is less than **set_show_max** (see 16.3.5.1), the message is printed to the standard output using the current verbosity and severity settings.

16.3.3.8 get_miscompares

```
virtual function string get_miscompares()
```

Returns the set of miscompares, if any, that have occurred since the comparer was constructed or since the last call to **flush** (see 16.2.4.2).

If no miscompares have occurred, then **get_miscompares** shall return an *empty string* (`""`).

16.3.3.9 get_result

```
virtual function int unsigned get_result()
```

Returns the number of miscompares that have occurred since the comparer was constructed or since the last call to **flush** (see 16.2.4.2).

16.3.4 Methods for comparer configuration

16.3.4.1 Recursion policy

```
virtual function void set_recursion_policy (uvm_recursion_policy_enum policy)
virtual function uvm_recursion_policy_enum get_recursion_policy()
```

Controls the recursion policy to use for object values supplied to **compare_object** (see 16.3.3.4).

> `UVM_DEEP`—Compares all fields of the object, doing a “deep” compare (any object fields are compared using a DEEP recursion).
>
> `UVM_SHALLOW`—Compares all fields of the object using a “shallow” compare (any object fields are compared as REFERENCES).
>
> `UVM_REFERENCE`—Compares the object as a reference.

A value of `UVM_DEFAULT_POLICY` shall be treated as `UVM_DEEP`.

If **set_recursion_policy** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_recursion_policy** shall return `UVM_DEFAULT_POLICY`.

16.3.4.2 Type checking

```
virtual function void set_check_type (bit enabled)
virtual function bit get_check_type()
```

Controls (via *enabled*) whether the **compare_object** method (see 16.3.3.4) compares the object types, given by **uvm_object::get_type** (see 5.3.4.5).

If **set_check_type** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_check_type** shall return `1`.

16.3.5 Methods for comparer reporting control

16.3.5.1 Max miscompare messages

```
virtual function void set_show_max (int unsigned show_max)
virtual function int unsigned get_show_max()
```

Controls the maximum allowed number of miscompare messages (*show_max*) generated by the comparer during a compare operation.

If **set_show_max** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_show_max** shall return `1`.

16.3.5.2 Verbosity

```
virtual function void set_verbosity (int unsigned verbosity)
virtual function int unsigned get_verbosity()
```

Controls the *verbosity* value to be used by the comparer when generating messages.

If **set_verbosity** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2),then **get_verbosity** shall return `UVM_LOW`.

16.3.5.3 Severity

```
virtual function void set_severity (uvm_severity severity)
virtual function uvm_severity get_severity()
```

Controls the *severity* value to be used by the comparer when generating messages.

If **set_severity** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_severity** shall return `UVM_INFO`.

16.3.6 Methods for object compare control

The following methods define values that may be used in **do_compare** (see 5.3.9.2) or **do_execute_op** (see 5.3.13.1) to control how fields are printed within an object:

Return threshold

```
virtual function void set_threshold (int unsigned threshold)
virtual function int unsigned get_threshold()
```

Controls the return *threshold* value.

A threshold value greater than `0` indicates that the **do_compare** (see 5.3.9.2) or **do_execute_op** (see 5.3.13.1) method may return as quickly as possible after the *result* (see 16.3.3.9) reaches the threshold value, potentially skipping additional field comparisons. A return value of `0` indicates all fields should be compared, even if miscompares have already been detected. This allows for a “fast failure” mode, which can detect a miscompare faster at the sacrifice of additional debugging information.

If **set_threshold** has not been called since the comparer was constructed or since the last call to **flush** (see 16.2.4.2), then **get_threshold** shall return `1`.

16.4 uvm_recorder

The **uvm_recorder** class serves two purposes.

— Firstly, it is an abstract representation of a record within a **uvm_tr_stream** (see 7.2).
— Secondly, it is a policy object for recording fields into that record within the *stream*.

16.4.1 Class declaration

```
virtual class uvm_recorder extends uvm_object
```

16.4.2 Methods for recorder configuration

16.4.2.1 recursion_policy

```
virtual function void set_recursion_policy (uvm_recursion_policy_enum policy)
virtual function uvm_recursion_policy_enum get_recursion_policy()
```

Controls the recursion policy to use for object values supplied to **record_object** (see 16.4.6.4).

> UVM_DEEP—Records all fields of the target, doing a "deep" record (any object fields are recorded using a DEEP recursion).
>
> UVM_SHALLOW—Records all fields of the target using a "shallow" record (any object fields are recorded as REFERENCES).
>
> UVM_REFERENCE—Records the target as a reference.

A value of UVM_DEFAULT_POLICY shall be treated as UVM_DEEP.

If **set_recursion_policy** has not been called since the recorder was constructed or since the last call to **flush** (see 16.2.4.2), then **get_recursion_policy** shall return UVM_DEFAULT_POLICY.

16.4.2.2 ID enabled

```
virtual function void set_id_enabled (bit enabled)
virtual function bit get_id_enabled()
```

Controls (via *enabled*) whether a unique reference ID shall be printed for object fields during **record_object** (see 16.4.6.4). A value of 1 indicates a unique reference ID shall be recorded; a value of 0 indicates the unique reference ID shall be omitted.

If **set_id_enabled** has not been called since the recorder was constructed or since the last call to **flush** (see 16.2.4.2), then **get_id_enabled** shall return 1.

16.4.2.3 Default radix

```
virtual function void set_default_radix (uvm_radix_enum radix)
virtual function uvm_radix_enum get_default_radix()
```

Controls the default *radix* used by **record_field** (see 16.4.6.1) or **record_field_int** (see 16.4.6.2) when **get_radix_enabled** (see 16.2.5.5) returns 1 and *radix* equals UVM_NORADIX.

If **set_default_radix** has not been called since the recorder was constructed or since the last call to **flush** (see 16.2.4.2), then **get_default_radix** shall return UVM_HEX.

16.4.3 Introspection API

get_stream

```
function uvm_tr_stream get_stream()
```

Returns a reference to the stream that created this record.

A warning shall be issued if **get_stream** is called prior to the record being initialized via **do_open** (see 16.4.7.1).

16.4.4 Transaction recorder API

Once a recorder has been opened via **uvm_tr_stream::open_recorder** (see 7.2.5.1), the user can then **close** (see 16.4.4.2) the recorder.

Since many database implementations cross a language boundary, an additional step of freeing the recorder is required.

A link can be established within the database any time between when **uvm_tr_stream::open_recorder** (see 7.2.5.1) and then **free** (see 16.4.4.3) are called, however it shall be an error to establish a link after freeing the recorder.

16.4.4.1 flush

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

The **flush** method resets the internal state of the recorder. If the recorder is currently open (see 16.4.4.4), then the implementation shall call **free** (see 16.4.4.3) with a *close_time* value of `0`.

16.4.4.2 close

```
function void close(
    time close_time = 0
)
```

Closes this recorder.

Closing a recorder marks the end of the transaction in the stream; it has the following parameter:

close_time—Optional time to record as the closing time of this transaction. The default value of *close_time* shall be `0`.

This method triggers a **do_close** call (see 16.4.7.2).

16.4.4.3 free

```
function void free(
    time close_time = 0
)
```

Frees this recorder.

Freeing a recorder indicates the stream and database can release any references to the recorder; it has the following parameter:

close_time—Optional time to record as the closing time of this transaction. The default value of *close_time* shall be `0`.

If a recorder has not yet been closed [via a call to **close** (see 16.4.4.2)], **close** is automatically called and passed the *close_time*. If the recorder has already been closed, the *close_time* is ignored.

This method triggers a **do_free** call (see 16.4.7.3).

16.4.4.4 is_open

```
function bit is_open()
```

Returns *True* if this **uvm_recorder** was opened on its stream, but has not yet been closed.

16.4.4.5 get_open_time

```
function time get_open_time()
```

Returns the *open_time*. See also 7.2.5.1.

16.4.4.6 is_closed

```
function bit is_closed()
```

Returns *True* if this **uvm_recorder** was closed on its stream, but has not yet been freed.

16.4.4.7 get_close_time

```
function time get_close_time()
```

Returns the *close_time* (see 16.4.4.2).

16.4.5 Handles

16.4.5.1 get_handle

```
function int get_handle()
```

Returns a unique ID for this recorder.

A value of `0` indicates the recorder has been freed and no longer has a valid ID.

16.4.5.2 get_recorder_from_handle

```
static function uvm_recorder get_recorder_from_handle(
    int id
)
```

This static accessor returns a recorder reference for a given unique *id*.

If no recorder exists with the given *id* or if the recorder with that *id* has been freed, then *null* is returned.

This method can be used to access the recorder associated with a call to **uvm_transaction::begin_tr** (see 5.4.2.4) or **uvm_component::begin_tr** (see 13.1.6.3).

16.4.6 Attribute recording

16.4.6.1 record_field

```
function void record_field(
    string name,
    uvm_bitstream_t value,
    int size,
    uvm_radix_enum radix = UVM_NORADIX
)
```

Records an integral field [less than or equal to the value defined by `UVM_MAX_STREAMBITS (see B.6.2) bits]; it has the following parameters:

name—Name of the field.

value—Value of the field to record.

size—Number of bits of the field that apply.

radix—The **uvm_radix_enum** (see F.2.1.5) to use. No *radix* information is provided, the printer/recorder can use its default *radix*. The default value of *radix* shall be UVM_NORADIX.

This method triggers a **do_record_field** call (see 16.4.7.4).

16.4.6.2 record_field_int

```
function void record_field_int(
    string name,
    uvm_integral_t value,
    int size,
    uvm_radix_enum radix = UVM_NORADIX
)
```

Records an integral field (less than or equal to 64 bits); it has the following parameters:

name—Name of the field.

value—Value of the field to record.

size—Number of bits of the field that apply.

radix—The **uvm_radix_enum** (see F.2.1.5) to use. No *radix* information is provided, the printer/recorder can use its default *radix*. The default value of *radix* shall be UVM_NORADIX.

This optimized version of **record_field** (see 16.4.6.1) is useful for sizes up to 64 bits.

This method triggers a **do_record_field_int** call (see 16.4.7.5).

16.4.6.3 record_field_real

```
function void record_field_real(
    string name,
    real value
)
```

Records a real field; it has the following parameters:

name—Name of the field.

value—Value of the field to record.

This method triggers a **do_record_field_real** call (see 16.4.7.6).

16.4.6.4 record_object

```
function void record_object(
    string name,
    uvm_object value
)
```

Records an object field; it has the following parameters:

name—is the name to use when recording the object. Note that this may be different then the value returned by the object's **get_name** method (see 5.3.4.2).

value—is the value of the object to be recorded. *null* can be passed as a *value*.

This method triggers a **do_record_object** call (see 16.4.7.7).

Whether a non-*null* value is recursed depends on a variety of knobs, such as **recursion_policy** (see 16.4.2.1). For objects that are being recursed, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).
b) The **do_record_object** method is called (see 16.4.7.7).
c) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

16.4.6.5 record_string

```
function void record_string(
    string name,
    string value
)
```

Records a string field; it has the following parameters:

name—Name of the field.

value—Value of the field.

This method triggers a **do_record_string** call (see 16.4.7.9).

16.4.6.6 record_time

```
function void record_time(
    string name,
    time value
)
```

Records a time field; it has the following parameters:

name—Name of the field.

value—Value of the field.

This method triggers a **do_record_time** call (see 16.4.7.10).

16.4.6.7 record_generic

```
function void record_generic(
    string name,
    string value,
    string type_name = ""
)
```

Records a *name*/*value* pair, where *value* has been converted to a string; it has the following parameters:

name—Name of the field.

value—Value of the field.

type_name—Type name of the field (optional).

This method triggers a **do_record_generic** call (see 16.4.7.11).

16.4.6.8 use_record_attribute

```
virtual function bit use_record_attribute()
```

Indicates that this recorder does (or does not) support usage of the **`uvm_record_attribute** macro (see B.2.3.1).

The default return value is `0` (not supported). **uvm_recorder** can be extended (set its value to `1`) to support **`uvm_record_attribute**.

16.4.6.9 get_record_attribute_handle

```
virtual function int get_record_attribute_handle()
```

This provides a tool-specific handle that is compatible with **`uvm_record_attribute** (see B.2.3.1).

By default, this method returns the same value as **get_handle**. Applications can override this method to provide tool-specific handles to passed to the **`uvm_record_attribute** macro.

16.4.7 Implementation agnostic API

16.4.7.1 do_open

```
protected virtual function void do_open(
    uvm_tr_stream stream,
    time open_time,
    string type_name
)
```

This is a callback triggered via **uvm_tr_stream::open_recorder** (see 7.2.5.1); it has the following parameters:

stream – The stream on which the recorder was opened.

open_time—The time to record as the opening of this transaction.

type_name—The type name for the transaction.

The **do_open** callback can be used to initialize any internal state within the recorder, as well as providing a location to record any initial information.

16.4.7.2 do_close

```
protected virtual function void do_close(
    time close_time
)
```

This is a callback triggered via **close** (see 16.4.4.2); it has the following parameter:

close_time—The time to record as the closing time of this transaction.

The **do_close** callback can be used to specify the internal state within the recorder, as well as providing a location to record any closing information.

16.4.7.3 do_free

```
protected virtual function void do_free()
```

This is a callback triggered via **free** (see 16.4.4.3).

The **do_free** callback can be used to release the internal state within the recorder, as well as providing a location to record any "freeing" information.

16.4.7.4 do_record_field

```
pure virtual protected function void do_record_field(
    string name,
    uvm_bitstream_t value,
    int size,
    uvm_radix_enum radix
)
```

Intended to record an integral field [less than or equal to the value defined by `UVM_MAX_STREAMBITS (see B.6.2) bits].

Derived classes need to provide an implementation of this API (see 16.4.6.1).

16.4.7.5 do_record_field_int

```
pure virtual protected function void do_record_field_int(
    string name,
    uvm_integral_t value,
    int size,
    uvm_radix_enum radix
)
```

Intended to record an integral field (less than or equal to 64 bits).

Derived classes need to provide an implementation of this API (see 16.4.6.2).

16.4.7.6 do_record_field_real

```
pure virtual protected function void do_record_field_real(
    string name,
    real value
)
```

Intended to record a real field.

Derived classes need to provide an implementation of this API (see 16.4.6.3).

16.4.7.7 do_record_object

```
pure virtual protected function void do_record_object(
    string name,
    uvm_object value
)
```

Implementation hook for record_object (see 16.4.6.4).

The default implementation executes the following steps in order:

a) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_RECORD` and *policy* set to this recorder.

b) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the recorder passes itself the **do_record** method (see 5.3.7.2) on the object; otherwise, the method returns without calling **do_record_object**.

16.4.7.8 object_recorded

```
virtual function uvm_policy::recursion_state_e object_recorded(
    uvm_object value,
    uvm_recursion_policy_enum recursion
)
```

Returns the current recursion state (see 16.1.4) for *value* and *recursion* within the recorder as defined by **record_object** (see 16.4.6.4). For objects that have never been passed to **record_object**, the return value shall be `uvm_policy::NEVER`.

16.4.7.9 do_record_string

```
pure virtual protected function void do_record_string(
    string name,
    string value
)
```

Intended to record a string field.

Derived classes need to provide an implementation of this API (see 16.4.6.5).

16.4.7.10 do_record_time

```
pure virtual protected function void do_record_time(
    string name,
    time value
)
```

Intended to record a time field.

Derived classes need to provide an implementation of this API (see 16.4.6.6).

16.4.7.11 do_record_generic

```
pure virtual protected function void do_record_generic(
    string name,
    string value,
    string type_name
)
```

Intended to record a *name*/*value* pair, where *value* has been converted to a string.

Derived classes need to provide an implementation of this API (see 16.4.6.7).

16.5 uvm_packer

The **uvm_packer** class provides a policy object for packing and unpacking **uvm_object**s (see 5.3). The policies determine how packing and unpacking shall be done.

16.5.1 Class declaration

```
class uvm_packer extends uvm_policy
```

16.5.2 Methods

16.5.2.1 new

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

Creates a new **uvm_packer** with the specified instance *name*. If *name* is not provided, the object is unnamed.

16.5.2.2 flush

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

The **flush** method resets the internal state of the packer. This includes clearing any data that has been previously packed via a call to one of the *packing* methods (see 16.5.4).

16.5.2.3 get_type_name

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

Returns the string "`uvm_packer`".

16.5.2.4 set_default

```
static function void set_default (uvm_packer packer)
```

Helper method for setting the default *packer* policy instance via **uvm_coreservice_t::set_default_packer** (see F.4.1.4.14).

16.5.2.5 get_default

```
static function uvm_packer get_default()
```

Helper method for retrieving the default packer policy instance via **uvm_coreservice_t::get_default_packer** (see F.4.1.4.15).

16.5.3 Methods for packer subtyping

16.5.3.1 State assignment

```
virtual function void set_packed_bits(
   ref bit stream[]
)

virtual function void set_packed_bytes(
   ref byte stream[]
)

virtual function void set_packed_ints(
   ref int stream[]
)

virtual function void set_packed_longints(
   ref longint stream[]
)
```

The *state assignment* methods set the internal state of the packer, such that the **unpack** methods can be used to retrieve previously packed data. The *stream* argument is a `bit`, `byte`, `int`, or `longint` array of unspecified length and format. Calling the *state assignment* methods with a *stream* that was not obtained from an identically typed *state retrieval* method (see 16.5.3.2) of a compatible packer implementation is undefined. Packers are considered compatible if their *state retrieval* methods return identical streams after packing identical fields.

16.5.3.2 State retrieval

```
virtual function void get_packed_bits(
   ref bit stream[]
)

virtual function void get_packed_bytes(
   ref byte stream[]
)

virtual function void get_packed_ints(
   ref int stream[]
)

virtual function void get_packed_longints(
   ref longint stream[]
)
```

The *state retrieval* methods copy the internal state of the packer to the *stream* argument, which is a `bit`, `byte`, `int`, or `longint` array of unspecified length and format. The length and contents of the *stream* are implementation dependent.

16.5.3.3 get_packed_size

```
virtual function int get_packed_size()
```

Returns the current number of bits of packed data stored within the packer.

16.5.4 Packing and unpacking

16.5.4.1 pack_bits

```
virtual function void pack_bits(
    ref bit value[],
    input int size = -1
)
```

Packs bits (*value*) from an unpacked array of bits. This method allows for fields of arbitrary length to be passed in using the SystemVerilog `stream` operator.

An optional *size* parameter is provided, which defaults to `-1`. If set to any value greater than `-1` (including `0`), the packer uses *size* as the number of bits to pack; otherwise, the packer simply packs the entire stream.

It shall be an error if *size* exceeds the size of the source array.

16.5.4.2 pack_object

```
virtual function void pack_object (
    uvm_object value
)
```

Packs an object *value*.

For objects that are being packed, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).
b) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_PACK` and *policy* set to this packer.
c) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the packer passes itself to the **do_pack** method (see 5.3.10.2) on *value*.
d) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

16.5.4.3 is_null

```
virtual function bit is_null()
```

This method is used during unpack operations to determine if the object at the current location in the pack data is *null* (whether to allocate a new object or not). If the object is *null*, the return value is a `1`; otherwise, it is `0`.

While **is_null** can be used to determine if an object in the packed data is *null*, it does not change the internal state of the *packer*. As such, regardless of the return value of **is_null**, **unpack_object** (see 16.5.4.4), needs to be called to move the *packer*'s internal state to the next field.

16.5.4.4 unpack_object

```
virtual function void unpack_object (
    uvm_object value
)
```

Unpacks an object and stores the result into *value*, which shall be an allocated object that has enough space for the data being unpacked or is *null*.

Use **is_null** (see 16.5.4.3) to determine if the object shall be set to *null* before calling this method. It shall be an error to pass a *value* argument that is incompatible with the return value of **is_null**, e.g., passing a *null* value when **is_null** returns `0` or passing a non-*null* value when **is_null** returns `1`. When this occurs, the *packer* shall generate an error message and the resulting behavior of any further **unpack_***calls on the *packer* is undefined.

For non-*null* objects which are being unpacked, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).

b) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_UNPACK` and *policy* set to this packer.

c) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the packer passes itself to the **do_unpack** method (see 5.3.11.2) on *value*.

d) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

16.5.4.5 pack_string

```
virtual function void pack_string (
    string value
)
```

Packs a string *value*.

16.5.4.6 pack_time

```
virtual function void pack_time (
    time value
)
```

Packs a time *value*.

16.5.4.7 pack_real

```
virtual function void pack_real (
    real value
)
```

Packs a real *value*.

16.5.4.8 pack_field

```
virtual function void pack_field (
    uvm_bitstream_t value,
    int size
)
```

Packs an integral value into the packed array. *size* is the number of bits of *value* to pack. An error message shall be generated if the *size* is larger than `` `UVM_MAX_STREAMBITS `` (see B.6.2).

16.5.4.9 pack_field_int

```
virtual function void pack_field_int (
    uvm_integral_t value,
    int size
)
```

Packs an integral value into the pack array. *size* is the number of bits to pack. An error message shall be generated if the *size* is larger that 64 bits.

NOTE—This optimized version of **pack_field** (see 16.5.4.8) is useful for *size*s up to `64` bits.

16.5.4.10 pack_bytes

```
virtual function void pack_bytes(
    ref byte value[],
    input int size = -1
)
```

Packs bits from an unpacked array of bytes into the pack array. *size* represents the number of bits to pack from the array. Setting *size* to `-1` indicates that the entire array shall be packed. The default value of *size* shall be `-1`.

An implementation shall generate an error message if the size is less than `-1` or greater than the total number of bits within the array.

See 16.5.4.1 for additional information.

16.5.4.11 pack_ints

```
virtual function void pack_ints(
    ref int value[],
    input int size = -1
)
```

Packs bits from an unpacked array of ints into the pack array. *size* represents the number of bits to pack from the array. Setting *size* to `-1` indicates that the entire array shall be packed. The default value of *size* shall be `-1`.

An implementation shall generate an error message if the size is less than `-1` or greater than the total number of bits within the array.

See 16.5.4.1 for additional information.

16.5.4.12 unpack_ints

```
virtual function void unpack_ints(
    ref int value[],
    input int size = -1
)
```

Unpacks bits from the pack array into an unpacked array of `int`s.

The unpacked array is unpacked from the internal pack array. This method allows for fields of arbitrary length to be passed in without expanding into a predefined integral type first.

An optional *size* parameter is provided, which defaults to `-1`. If set to any value greater than `-1` (including `0`), the packer uses *size* as the number of bits to unpack; otherwise, the packer simply unpacks the entire stream.

It shall be an error to specify a *size* that exceeds the size of the source array.

16.5.4.13 unpack_string

```
virtual function string unpack_string()
```

Unpacks a string.

16.5.4.14 unpack_time

```
virtual function time unpack_time()
```

Unpacks a time variable.

16.5.4.15 unpack_real

```
virtual function real unpack_real()
```

Unpacks a real variable.

16.5.4.16 unpack_field

```
virtual function uvm_bitstream_t unpack_field (
    int size
)
```

Unpacks bits from the pack array and returns the bitstream that was unpacked. *size* is the number of bits to unpack; the maximum is the value defined by `` `UVM_MAX_STREAMBITS `` (see B.6.2) bits.

16.5.4.17 unpack_field_int

```
virtual function uvm_integral_t unpack_field_int (
    int size
)
```

Unpacks bits from the pack array and returns the bitstream that was unpacked.

size is the number of bits to unpack; the maximum is `64` bits. This is a more efficient variant than **unpack_field** (see 16.5.4.16) when unpacking into smaller vectors.

16.5.4.18 unpack_bits

```
virtual function void unpack_bits(
    ref bit value[],
    input int size = -1
)
```

Unpacks bits from the pack array into an unpacked array of bits.

An optional *size* parameter is provided, which defaults to `-1`. If set to any value greater than `-1` (including `0`), the packer uses *size* as the number of bits to unpack; otherwise, the packer simply unpacks the entire stream.

16.5.4.19 unpack_bytes

```
virtual function void unpack_bytes(
    ref byte value[],
    input int size = -1
)
```

Unpacks bits from the pack array into an unpacked array of bytes.

An optional *size* parameter is provided, which defaults to `-1`. If set to any value greater than `-1` (including `0`), the packer uses *size* as the number of bits to unpack; otherwise, the packer simply unpacks the entire stream.

16.6 uvm_copier

The **uvm_copier** class provides a policy object for copying **uvm_object**s (see 5.3). The policies determine how copying should be done.

16.6.1 Class declaration

```
class uvm_copier extends uvm_policy
```

16.6.2 Methods

16.6.2.1 new

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

Creates a new **uvm_copier** with the specified instance *name*. If *name* is not provided, the object is unnamed.

16.6.2.2 get_type_name

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

Returns the string "`uvm_copier`".

16.6.2.3 set_default

```
static function void set_default (
    uvm_copier copier
)
```

Helper method for setting the default copier policy instance via **uvm_coreservice_t::set_default_copier** (see F.4.1.4.18).

16.6.2.4 get_default

```
static function uvm_copier get_default()
```

Helper method for retrieving the default copier policy instance via **uvm_coreservice_t::get_default_copier** (see F.4.1.4.19).

16.6.3 Methods for object copy control

The following methods define values that may be used in **do_copy** (see 5.3.8.2) or **do_execute_op** (see 5.3.13.1) to control how fields are copied within an object:

Recursion policy

```
virtual function void set_recursion_policy (uvm_recursion_policy_enum policy)
virtual function uvm_recursion_policy_enum get_recursion_policy()
```

Controls the recursion policy to use for object fields during **do_copy** (see 5.3.8.2) or **do_execute_op** (see 5.3.13.1).

> `UVM_DEEP`—Copy all fields of *rhs* to *lhs*, doing a "deep" copy [any object fields are copied using a DEEP recursion, i.e., `copier.copy_object(tgt, src)`].
>
> `UVM_SHALLOW`—Copy all fields of *rhs* to *lhs*, using a "shallow" copy (any object fields are copied as REFERENCES, i.e,. `tgt = src`).
>
> `UVM_REFERENCE`—Copy the *rhs* to *lhs* as a reference.

A value of `UVM_DEFAULT_POLICY` shall be treated as `UVM_DEEP`.

If **set_recursion_policy** has not been called since the copier was constructed or since the last call to **flush** (see 16.2.4.2), then **get_recursion_policy** shall return `UVM_DEFAULT_POLICY`.

16.6.4 Methods for copier usage

16.6.4.1 copy_object

```
virtual function void copy_object (
    uvm_object lhs,
    uvm_object rhs
)
```

Copies the fields of *rhs* to *lhs* using the *recursion policy* (see 16.6.3) to determine whether the copy should be deep or shallow. Objects that are meant to be copied by reference shall use an assignment operation.

Unlike other policies, the *copier* relies on **do_copy** (see 5.3.8.2) and **do_execute_op** (see 5.3.13.1) to process copies via direct assignment when the *recursion policy* is set to `UVM_REFERENCE`. The copier shall generate an error message if **copy_object** is called when the *recursion policy* is set to `UVM_REFERENCE`, and the result of the **copy_object** operation is undefined.

For objects that are being copied, the following steps occur in order:

a) The object is pushed onto the active object stack via **push_active_object** (see 16.1.3.1).
b) The saved recursion state (see 16.6.4.2) for *lhs*, *rhs*, and the current *recursion policy* (see 16.6.3) is set to `uvm_policy::STARTED`.
c) The **do_execute_op** method (see 5.3.13.1) on the object is passed a **uvm_field_op** (see 5.3.13.2) with *op_type* `UVM_COPY` and *policy* set to this copier.
d) If **user_hook_enabled** (see 5.3.13.2.9) returns `1`, the copier passes itself and the *rhs* to the **do_copy** method (see 5.3.8.2) on *lhs*; otherwise, the method returns without calling **do_copy**.

e) The saved recursion state (see 16.6.4.2) for *lhs*, *rhs*, and the current *recursion policy* is set to `uvm_policy::FINISHED`.

f) The object is popped off of the active object stack via **pop_active_object** (see 16.1.3.2).

16.6.4.2 object_copied

```
virtual function uvm_policy::recursion_state_e object_copied(
    uvm_object lhs,
    uvm_object rhs,
    uvm_recursion_policy_enum recursion,
)
```

Returns the current recursion state (see 16.1.4) for *lhs*, *rhs*, and *recursion* within the copier as defined by **copy_object** (see 16.6.4.1). For objects that have never been passed to **copy_object**, the return value shall be `uvm_policy::NEVER`.

The values passed to *lhs* and *rhs* need to be passed to **object_copied** using the same ordering as **copy_object**.

17. Register layer

17.1 Overview

The UVM register layer defines base classes that, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification.

17.2 Global declarations

This subclause defines the globally available types, enums, and utility classes.

17.2.1 Types

17.2.1.1 uvm_reg_data_t

2-state data value with **`UVM_REG_DATA_WIDTH** bits (see B.6.5).

17.2.1.2 uvm_reg_data_logic_t

4-state data value with **`UVM_REG_DATA_WIDTH** bits (see B.6.5).

17.2.1.3 uvm_reg_addr_t

2-state address value with **`UVM_REG_ADDR_WIDTH** bits (see B.6.4).

17.2.1.4 uvm_reg_addr_logic_t

4-state address value with **`UVM_REG_ADDR_WIDTH** bits (see B.6.4).

17.2.1.5 uvm_reg_byte_en_t

2-state `byte_enable` value with **`UVM_REG_BYTENABLE_WIDTH** bits (see B.6.6).

17.2.1.6 uvm_reg_cvr_t

Coverage model value specified with **`UVM_REG_CVR_WIDTH** bits (see B.6.7).

Symbolic values for individual coverage models are defined by the **uvm_coverage_model_e** type (see 17.2.2.9).

The bits in the setting are assigned as follows:

Bits

`0-7`	Reserved.
`8-15`	Coverage models defined by applications, implemented in a register model generator.
`16-23`	User-defined coverage models.
`24-`\`UVM-REG_CVR_WIDTH-1`	Reserved.

17.2.1.7 uvm_hdl_path_slice

Slice of an HDL (hardware description language) path.

This is a struct that specifies the HDL variable corresponding to all of or a portion of a register; it has the following parameters:

path—Path to the HDL variable.

offset—Offset of the least significant bit (LSB) in the register that this variable implements.

size—Number of bits [toward the most significant bit (MSB)] that this variable implements.

If the HDL variable implements all of the register, *offset* and *size* are specified as `-1`, e.g.:
`r1.add_hdl_path('{ '{"r1", -1, -1} }).`

17.2.2 Enumerations

17.2.2.1 uvm_status_e

Return status for register operations; it has the following enumerated types:

`UVM_IS_OK`—Operation completed successfully.

`UVM_NOT_OK`—Operation completed with error.

`UVM_HAS_X`—Operation completed successfully, but had unknown bits.

17.2.2.2 uvm_door_e

Door used for register operation; it has the following enumerated types:

`UVM_FRONTDOOR`—Use the front door.

`UVM_BACKDOOR`—Use the back door.

`UVM_PREDICT`—Operation derived from observations by a bus monitor via the **uvm_reg_predictor** class (see 19.3).

`UVM_DEFAULT_DOOR`—Operation specified by the context.

17.2.2.3 uvm_check_e

Use read-only or read-and-check; it has the following enumerated types:

`UVM_NO_CHECK`—Read only.

`UVM_CHECK`—Read and check.

17.2.2.4 uvm_endianness_e

Specifies byte ordering; it has the following enumerated types:

`UVM_NO_ENDIAN`—Byte ordering not applicable.

`UVM_LITTLE_ENDIAN`—Least-significant bytes first in consecutive addresses.

`UVM_BIG_ENDIAN`—Most-significant bytes first in consecutive addresses.

`UVM_LITTLE_FIFO`—Least-significant bytes first at the same address.

`UVM_BIG_FIFO`—Most-significant bytes first at the same address.

17.2.2.5 uvm_elem_kind_e

Type of element being read or written; it has the following enumerated types:

UVM_REG—Register.
UVM_FIELD—Field.
UVM_MEM—Memory location.

17.2.2.6 uvm_access_e

Type of operation begin performed; it has the following enumerated types:

UVM_READ—Read operation.
UVM_WRITE—Write operation.

17.2.2.7 uvm_hier_e

Whether to provide the requested information from a hierarchical context; it has the following enumerated types:

UVM_NO_HIER—Provide info from the local context.
UVM_HIER—Provide info based on the hierarchical context.

17.2.2.8 uvm_predict_e

How the mirror is to be updated; it has the following enumerated types:

UVM_PREDICT_DIRECT—Predicted value is as is.
UVM_PREDICT_READ—Predict based on the specified value having been read.
UVM_PREDICT_WRITE—Predict based on the specified value having been written.

17.2.2.9 uvm_coverage_model_e

Coverage models available or desired; it has the following enumerated types:

UVM_NO_COVERAGE—None.
UVM_CVR_REG_BITS—Individual register bits.
UVM_CVR_ADDR_MAP—Individual register and memory addresses.
UVM_CVR_FIELD_VALS—Field values.
UVM_CVR_ALL—All coverage models.

Multiple models may be specified by bitwise ORing individual model identifiers.

17.2.2.10 uvm_reg_mem_tests_e

Selects which predefined test sequence to execute; it has the following parameters:

UVM_DO_REG_HW_RESET—Run **uvm_reg_hw_reset_seq** (see E.1).
UVM_DO_REG_BIT_BASH—Run **uvm_reg_bit_bash_seq** (see E.2.2).
UVM_DO_REG_ACCESS—Run **uvm_reg_access_seq** (see E.3.2).
UVM_DO_MEM_ACCESS—Run **uvm_mem_access_seq** (see E.5.2).
UVM_DO_SHARED_ACCESS—Run **uvm_reg_mem_shared_access_seq** (see E.4.3).
UVM_DO_MEM_WALK—Run **uvm_mem_walk_seq** (see E.6.2).
UVM_DO_ALL_REG_MEM_TESTS—Run all of the above.

Multiple test sequences may be selected by bitwise `OR`ing their respective symbolic values.

Test sequences, when selected, are executed in the order in which they are specified here.

18. Register model

A register model is typically composed of a hierarchy of blocks. Blocks contain registers, register files, memories, and address maps.

18.1 uvm_reg_block

This is the block abstraction base class.

A block represents a design hierarchy. It can contain registers, register files, memories, and sub-blocks. A block has one or more address maps, each corresponding to a physical interface on the block.

18.1.1 Class declaration

```
class uvm_reg_block extends uvm_object
```

18.1.2 Methods

18.1.2.1 new

```
function new(
   string name = "",
   int has_coverage = UVM_NO_COVERAGE
)
```

This creates an instance of a block abstraction class with the specified *name*.

has_coverage specifies which functional coverage models are present in the extension of the block abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the **uvm_coverage_model_e** type (see 17.2.2.9). The default value of *has_coverage* shall be `UVM_NO_COVERAGE`.

18.1.2.2 configure

```
function void configure(
   uvm_reg_block parent = null,
   string hdl_path = ""
)
```

This is an instance-specific configuration; it specifies the *parent* block of this block. A block without *parent* is a root block.

If the block file corresponds to a hierarchical register transfer level (RTL) structure, its contribution to the HDL path is specified as the *hdl_path*. Otherwise, the block does not correspond to a hierarchical RTL structure (i.e., it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers or memories.

18.1.2.3 create_map

```
virtual function uvm_reg_map create_map(
       string name,
       uvm_reg_addr_t base_addr,
       int unsigned n_bytes,
       uvm_endianness_e endian,
       bit byte_addressing = 1
)
```

This creates an address map with the specified *name*, then configures it with the following properties:

a) *base_addr*—the base address for the map. All registers, memories, and sub-blocks within the map will be at offsets to this address.

b) *n_bytes*—the byte-width of the bus on which this map is used.

c) *endian*—the endian format. See **uvm_endianness_e** for possible values (see 17.2.2.4).

d) *byte_addressing*—specifies whether consecutive addresses are `1` byte apart (`TRUE`; the default) or `n_bytes` apart (`FALSE`).

18.1.2.4 set_default_map

```
function void set_default_map (
   uvm_reg_map map
)
```

This makes the specified address map the default mapping for this block. The address map needs to be a map of this address block.

18.1.2.5 lock_model

```
virtual function void lock_model()
```

This recursively locks an entire register model and builds the address maps to enable the **uvm_reg_map::get_reg_by_offset** (see 18.2.4.17) and **uvm_reg_map::get_mem_by_offset** (see 18.2.4.18) methods.

When locked, no structural changes, such as adding registers or memories, can be made. Hence, it is important that all sub-blocks, maps, and registers have been created before the **lock_model** is called.

18.1.2.6 unlock_model

```
virtual function void unlock_model()
```

Unlocks the register model, bringing the register mode to the state before **lock_model** (see 18.1.2.5), such that structural changes are allowed again.

This invalidates all precomputed information derived in a previous call to **lock_model**, as well as any information that has been cached since the last call to **lock_model**.

18.1.2.7 set_lock

```
virtual function void set_lock(
   bit v
)
```

Sets the lock mode to *v* for the current register block and all its sub-blocks.

18.1.2.8 wait_for_lock

```
task wait_for_lock()
```

Blocks until **lock_model** (see 18.1.2.5) completes.

18.1.2.9 is_locked

```
function bit is_locked()
```

Returns `TRUE` if the model is locked.

18.1.2.10 unregister

```
virtual function void unregister(
   uvm_reg_map m
)
```

Removes all knowledge of map *m* from the current block and, therefore, all registers, memories, and virtual registers contained in *m* from the current block.

18.1.3 Introspection

18.1.3.1 get_parent

```
virtual function uvm_reg_block get_parent()
```

Returns the parent block.

When this a top-level block, returns *null*.

18.1.3.2 get_root_blocks

```
static function void get_root_blocks(
   ref uvm_reg_block blks[$]
)
```

This returns an array of all root blocks in the simulation. *blks* shall be a queue.

18.1.3.3 find_blocks

```
static function int find_blocks(
   input string name,
   ref uvm_reg_block blks[$],
   input uvm_reg_block root = null,
   input uvm_object accessor = null
)
```

Finds the blocks whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block; otherwise, they are absolute. *blks* shall be a queue.

This returns the number of blocks found.

18.1.3.4 find_block

```
static function uvm_reg_block find_block(
   input string name,
   input uvm reg block root = null,
   input uvm_object accessor = null
)
```

Finds the first block whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block; otherwise, they are absolute.

This returns the first block found or *null* otherwise. A warning shall be issued if more than one block is found.

18.1.3.5 get_blocks

```
virtual function void get_blocks (
    ref uvm_reg_block blks[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the sub-blocks instantiated in these blocks. If *hier* is `UVM_HIER`, recursively includes any sub-blocks. The default value of *hier* shall be `UVM_HIER`. *blks* shall be a queue.

18.1.3.6 get_maps

```
virtual function void get_maps (
    ref uvm_reg_map maps[$]
)
```

Returns the address maps instantiated in this block. *maps* shall be a queue.

18.1.3.7 get_registers

```
virtual function void get_registers (
    ref uvm_reg regs[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the registers instantiated in this block. If *hier* is `TRUE`, recursively includes the registers in the sub-blocks. The default value of *hier* shall be `UVM_HIER`. *regs* shall be a queue.

Note that registers may be located in different and/or multiple address maps. To find the registers in a specific address map, use the **uvm_reg_map::get_registers** method (see 18.2.4.11).

18.1.3.8 get_fields

```
virtual function void get_fields (
    ref uvm_reg_field fields[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the fields in the registers instantiated in this block. If *hier* is `TRUE`, recursively includes the fields of the registers in the sub-blocks. The default value of *hier* shall be `UVM_HIER`. *fields* shall be a queue.

18.1.3.9 get_memories

```
virtual function void get_memories (
    ref uvm_mem mems[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the memories instantiated in this block. If *hier* is `TRUE`, recursively includes the memories in the sub-blocks. The default value of *hier* shall be `UVM_HIER`. *mems* shall be a queue.

Note that memories may be located in different and/or multiple address maps. To find the memories in a specific address map, use the **uvm_reg_map::get_memories** method (see 18.2.4.13).

18.1.3.10 get_virtual_registers

```
virtual function void get_virtual_registers(
   ref uvm_vreg regs[$],
   input uvm_hier_e hier = UVM_HIER
)
```

Returns the virtual registers instantiated in this block. If *hier* is `TRUE`, recursively includes the virtual registers in the sub-blocks. The default value of *hier* shall be `UVM_HIER`. *regs* shall be a queue.

18.1.3.11 get_virtual_fields

```
virtual function void get_virtual_fields (
   ref uvm_vreg_field fields[$],
   input uvm_hier_e hier = UVM_HIER
)
```

Returns the virtual fields from the virtual registers instantiated in this block. If *hier* is `TRUE`, recursively includes the virtual fields in the virtual registers in the sub-blocks. The default value of *hier* shall be `UVM_HIER`. *fields* shall be a queue.

18.1.3.12 get_block_by_name

```
virtual function uvm_reg_block get_block_by_name (
   string name
)
```

Finds a sub-block with the specified simple name.

The *name* is the simple name of the block, not a hierarchical name relative to this block. If no block with that name is found in this block, the sub-blocks are searched for a block of that name and the first one to be found is returned.

If no blocks are found, returns *null*.

18.1.3.13 get_map_by_name

```
virtual function uvm_reg_map get_map_by_name (
   string name
)
```

Finds an address map with the specified simple name.

The *name* is the simple name of the address map, not a hierarchical name relative to this block. If no map with that name is found in this block, the sub-blocks are searched for a map of that name and the first one to be found is returned.

If no address maps are found, returns *null*.

18.1.3.14 get_reg_by_name

```
virtual function uvm_reg get_reg_by_name (
    string name
)
```

Finds a register with the specified simple name.

The *name* is the simple name of the register, not a hierarchical name relative to this block. If no register with that name is found in this block, the sub-blocks are searched for a register of that name and the first one to be found is returned.

If no registers are found, returns *null*.

18.1.3.15 get_field_by_name

```
virtual function uvm_reg_field get_field_by_name (
    string name
)
```

Finds a field with the specified simple name.

The *name* is the simple name of the field, not a hierarchical name relative to this block. If no field with that name is found in this block, the sub-blocks are searched for a field of that name and the first one to be found is returned.

If no fields are found, returns *null*.

18.1.3.16 get_mem_by_name

```
virtual function uvm_mem get_mem_by_name (
    string name
)
```

Finds a memory with the specified simple name.

The *name* is the simple name of the memory, not a hierarchical name relative to this block. If no memory with that name is found in this block, the sub-blocks are searched for a memory of that name and the first one to be found is returned.

If no memories are found, returns *null*.

18.1.3.17 get_vreg_by_name

```
virtual function uvm_vreg get_vreg_by_name (
    string name
)
```

Finds a virtual register with the specified simple name.

The *name* is the simple name of the virtual register, not a hierarchical name relative to this block. If no virtual register with that name is found in this block, the sub-blocks are searched for a virtual register of that name and the first one to be found is returned.

If no virtual registers are found, returns *null*.

18.1.3.18 get_vfield_by_name

```
virtual function uvm_vreg_field get_vfield_by_name (
    string name
)
```

Finds a virtual field with the specified simple name.

The *name* is the simple name of the virtual field, not a hierarchical name relative to this block. If no virtual field with that name is found in this block, the sub-blocks are searched for a virtual field of that name and the first one to be found is returned.

If no virtual fields are found, returns *null*.

18.1.4 Coverage

18.1.4.1 build_coverage

```
protected function uvm_reg_cvr_t build_coverage(
    uvm_reg_cvr_t models
)
```

Checks if all of the specified coverage model needs to be built in this instance of the block abstraction class, as specified by calls to **uvm_reg::include_coverage** (see 18.4.7.1).

models are specified by adding the symbolic value of individual coverage models as defined in 17.2.2.9. This returns the sum of all coverage models to be built in the block model.

18.1.4.2 add_coverage

```
virtual protected function void add_coverage(
    uvm_reg_cvr_t models
)
```

Specifies that additional coverage models are available.

Adds the specified coverage model to the coverage models available in this class. *models* are specified by adding the symbolic value of individual coverage model as defined in 17.2.2.9.

This method shall be only called in the constructor of subsequently derived classes.

18.1.4.3 has_coverage

```
virtual function bit has_coverage(
    uvm_reg_cvr_t models
)
```

Checks if this block has coverage model(s).

This returns `TRUE` if the block abstraction class contains a coverage model for all of the models specified. *models* are specified by adding the symbolic value of individual coverage model as defined in 17.2.2.9.

18.1.4.4 get_coverage

```
virtual function bit get_coverage(
    uvm_reg_cvr_t is_on = UVM_CVR_ALL
)
```

Checks if coverage measurement is on.

This returns `TRUE` if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. The default value of *is_on* shall be `UVM_CVR_ALL`.

See 18.1.4.5 for more details.

18.1.4.5 set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(
    uvm_reg_cvr_t is_on
)
```

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this block and all blocks, registers, fields, and memories within it. The functional coverage measurement is turned on for every coverage model specified using **uvm_coverage_model_e** symbolic identifiers (see 17.2.2.9). Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off.

This returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the various abstraction classes, then enabled during construction. See 18.1.4.3 to identify the available functional coverage models.

18.1.4.6 sample

```
protected virtual function void sample(
    uvm_reg_addr_t offset,
    bit is_read,
    uvm_reg_map map
)
```

This is a functional coverage sampling method; it is invoked by the block abstraction class whenever an address within one of its address maps is successfully read or written. The specified *offset* is the offset within the block, not an absolute address.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

18.1.4.7 sample_values

```
virtual function void sample_values()
```

This is a functional coverage sampling method for field values; it is invoked by the user or by the **uvm_reg_block::sample_values** method of the parent block to trigger the sampling of the current field

values in the block-level functional coverage model. It also recursively invokes the **uvm_reg_block::sample_values** and **uvm_reg::sample_values** methods (see 18.4.7.8) in the blocks and registers within this block.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model. If this method is extended, it shall call `super.sample_values`.

18.1.5 Access

18.1.5.1 get_default_door

```
virtual function uvm_door_e get_default_door()
```

This returns the default door for this block (`UVM_FRONTDOOR` or `UVM_BACKDOOR`), see 17.2.2.2.

18.1.5.2 set_default_door

```
virtual function void set_default_door(
   uvm_door_e door
)
```

This sets the default door for this block.

18.1.5.3 reset

```
virtual function void reset(
   string kind = "HARD"
)
```

This clears all access semaphores and sets the mirror value of all registers in the block and sub-blocks to the reset value corresponding to the specified reset event. See 18.5.5.4 for more details.

This does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror. The default value of *kind* shall be `"HARD"`.

18.1.5.4 needs_update

```
virtual function bit needs_update()
```

Checks if DUT registers need to be written.

If a mirror value has been modified in the abstraction model without actually updating the actual register [either through randomization or via the **uvm_reg::set** method (see 18.4.4.2)], the mirror and state of the registers are outdated. Then, any corresponding registers in the DUT need to be updated.

This method returns `TRUE` if the state of at least one register in the block or sub-blocks needs to be updated to match the mirrored values. The mirror values, or actual content of registers, are not modified. For additional information, see 18.1.5.5.

18.1.5.5 update

```
virtual task update(
   output uvm_status_e status,
   input uvm_door_e path = UVM_DEFAULT_DOOR,
```

```
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This updates all of the design registers in this block and its sub-blocks with the mirrored value using the minimum number of write operations. The accesses may be performed using front-door or back-door operations (see 18.4.4.13).

This method performs the reverse operation of **uvm_reg_block::mirror** (see 18.1.5.6).

18.1.5.6 mirror

```
virtual task mirror(
    output uvm_status_e status,
    input uvm_check_e check = UVM_NO_CHECK,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads all of the registers in this block and its sub-blocks and updates their mirror values to match the corresponding values in the design. If *check* is set to `UVM_CHECK` (see 17.2.2.3), an error message shall be generated when the current mirrored value does not match the actual value in the design.

The accesses may be performed using front-door or back-door operations (see 18.4.4.14).

This method performs the reverse operation of **uvm_reg_block::update** (see 18.1.5.6).

18.1.5.7 Others

For other **uvm_reg_block** convenience access methods, see D.3.

18.1.6 Back door

18.1.6.1 get_backdoor

```
function uvm_reg_backdoor get_backdoor(
    bit inherited = 1
)
```

This returns the user-defined back door for all registers in this block and all sub-blocks—unless it is overridden by a back door specified in a lower-level block or in the register itself.

If *inherited* is `TRUE`, and no back door has been specified for this block, and a parent block has been specified, then this returns the value of the **get_backdoor** (inherited) from the parent block. The default value of *inherited* shall be `1`, which is `TRUE`.

18.1.6.2 set_backdoor

```
function void set_backdoor (
   uvm_reg_backdoor bkdr,
   string fname = "",
   int lineno = 0
)
```

This defines the back-door mechanism for all registers instantiated in this block and sub-blocks, unless overridden by a definition in a lower-level block or register. The default value of *lineno* shall be `0`.

18.1.6.3 clear_hdl_path

```
function void clear_hdl_path (
   string kind = "RTL"
)
```

This removes any previously specified HDL path to the block instance for the specified design abstraction *kind*. The default value of *kind* shall be `"RTL"`.

18.1.6.4 add_hdl_path

```
function void add_hdl_path (
   string path,
   string kind = "RTL"
)
```

This adds the specified HDL *path* to the block instance for the specified design abstraction *kind*. This method may be called more than once for the same design abstraction if the block is physically duplicated in the design abstraction. The default value of *kind* shall be `"RTL"`.

18.1.6.5 has_hdl_path

```
function bit has_hdl_path (
   string kind = ""
)
```

This returns `TRUE` if the block instance has a HDL path defined for the specified design abstraction *kind*. If no design abstraction is specified, it uses the default design abstraction specified for this block or the nearest block ancestor with a specified default design abstraction.

18.1.6.6 get_hdl_path

```
function void get_hdl_path (
   ref string paths[$],
   input string kind = ""
)
```

This returns the HDL path(s) defined for the specified design abstraction, *kind*, in the block instance. It returns only the component of the HDL paths that corresponds to the block, not a full hierarchical path. *paths* shall be a queue.

If no design abstraction is specified, the default design abstraction for this block is used.

18.1.6.7 get_full_hdl_path

```
function void get_full_hdl_path (
   ref string paths[$],
   input string kind = "",
   string separator = "."
)
```

This returns the full hierarchical HDL path(s) defined for the specified design abstraction, *kind*, in the block instance. There may be more than one path returned when any of the parent components have more than one path defined for the same design abstraction, even if only one path was defined for the block instance. *paths* shall be a queue. The default *separator* of *kind* shall be `"."`.

If no design abstraction is specified for the current block, then it is determined via the default design abstraction (see 18.1.6.8).

18.1.6.8 get_default_hdl_path

```
function string get_default_hdl_path()
```

This returns the default design abstraction for this block instance. If a default design abstraction has not been explicitly specified for this block instance, it returns the default design abstraction for the nearest block ancestor. This returns the string `"RTL"` if no default design abstraction has been specified.

18.1.6.9 set_default_hdl_path

```
function void set_default_hdl_path (
   string kind
)
```

Specifies the default design abstraction, *kind*, for this block instance.

18.1.6.10 set_hdl_path_root

```
function void set_hdl_path_root (
   string path,
   string kind = "RTL"
)
```

This sets the specified *path* as the absolute HDL path to the block instance for the specified design abstraction *kind*. This absolute root path is prepended to all hierarchical paths under this block. The HDL path of any ancestor block is ignored. This method overrides any incremental path for the same design abstraction specified using **add_hdl_path** (see 18.1.6.4). The default value of *kind* shall be `"RTL"`.

18.1.6.11 is_hdl_path_root

```
function bit is_hdl_path_root (
   string kind = ""
)
```

This returns `TRUE` if an absolute HDL path to the block instance for the specified design abstraction, *kind*, has been defined. If no design abstraction is specified, the default design abstraction for this block is used.

18.2 uvm_reg_map

This class represents an address map. An address map is a collection of registers and memories accessible via a specific physical interface. Address maps can be composed into higher-level address maps.

Address maps are created using the **uvm_reg_block::create_map** method (see 18.1.2.3).

18.2.1 Class declaration

```
class uvm_reg_map extends uvm_object
```

18.2.2 Common methods

backdoor

```
static function uvm_reg_map backdoor()
```

Returns the back-door pseudo-map singleton.

This pseudo-map is used to specify or configure the back door instead of a real address map.

18.2.3 Methods

18.2.3.1 new

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

Creates a new instance.

18.2.3.2 configure

```
virtual function void configure(
    uvm_reg_block parent,
    uvm_reg_addr_t base_addr,
    int unsigned n_bytes,
    uvm_endianness_e endian,
    bit byte_addressing = 1
)
```

This is an instance-specific configuration.

Configures this map with the following properties:

a) *parent*—the block in which this map is created and applied.
b) *base_addr*—the base address for this map. All registers, memories, and sub-blocks are at offsets to this address.
c) *n_bytes*—the byte-width of the bus on which this map is used.
d) *endian*—the endian format. See 17.2.2.4 for possible values.
e) *byte_addressing*—specifies whether the address increment is on a per-byte basis. For example, consecutive memory locations with `n_bytes=4` (a 32-bit bus) are `4` increments apart: `0`, `4`, `8`, and so on. The default value of *byte_addressing* shall be `1`, which is `TRUE`.

18.2.3.3 add_reg

```
virtual function void add_reg (
    uvm_reg rg,
    uvm_reg_addr_t offset,
    string rights = "RW",
    bit unmapped = 0,
    uvm_reg_frontdoor frontdoor = null
)
```

Adds the specified register instance *rg* to this address map.

The register is located at the specified address *offset* from this map's configured base address.

The *rights* specify the register's accessibility via this map. Valid values are **"RW"**, **"RO"**, and **"WO"**. Whether a register field can be read or written depends on both the field's configured access policy (see 18.4.4) and the register's rights in the map being used to access the field. The default value of *rights* shall be `"RW"`.

The number of consecutive physical addresses occupied by the register depends on the width of the register and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is *True*, the register does not occupy any physical addresses and the base address is ignored. Unmapped registers require a user-defined *front door* to be specified. The default value of *unmapped* shall be `0`, which is `FALSE`.

A register may be added to multiple address maps if it is accessible from multiple physical interfaces. A register may only be added to an address map whose parent block is the same as the register's parent block.

18.2.3.4 add_mem

```
virtual function void add_mem (
    uvm_mem mem,
    uvm_reg_addr_t offset,
    string rights = "RW",
    bit unmapped = 0,
    uvm_reg_frontdoor frontdoor = null
)
```

Adds the specified memory instance *mem* to this address map.

The memory is located at the specified base address, *offset*, and has the specified access *rights* (valid values are **"RW"**, **"RO"**, and **"WO"**).The default value of *rights* shall be `"RW"`.

The number of consecutive physical addresses occupied by the memory depends on the width and size of the memory and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is *True*, the memory does not occupy any physical addresses and the base address is ignored. Unmapped memories require a user-defined *front door* to be specified. The default value of *unmapped* shall be `0`, which is `FALSE`.

A memory may be added to multiple address maps if it is accessible from multiple physical interfaces. A memory may only be added to an address map whose parent block is the same as the memory's parent block.

18.2.3.5 add_submap

```
virtual function void add_submap (
    uvm_reg_map child_map,
    uvm_reg_addr_t offset
)
```

Adds the specified address map instance to this address map. The address map is located at the specified base address. The number of consecutive physical addresses occupied by the submap depends on the number of bytes in the physical interface that corresponds to the submap, the number of addresses used in the submap, and the number of bytes in the physical interface corresponding to this address map.

An address map may be added to multiple address maps if it is accessible from multiple physical interfaces. An address map may only be added to an address map in the grandparent block of the address submap.

18.2.3.6 set_sequencer

```
virtual function void set_sequencer (
    uvm_sequencer_base sequencer,
    uvm_reg_adapter adapter = null
)
```

Specifies the sequencer and adapter associated with this map. This method needs to be called before starting any sequences based on **uvm_reg_sequence** (see 19.4.1).

18.2.3.7 get_submap_offset

```
virtual function uvm_reg_addr_t get_submap_offset (
    uvm_reg_map submap
)
```

Returns the offset of the given *submap*. If the *submap* does not exist, or if the handle to the *submap* is *null*, this generates an error and returns `-1`.

18.2.3.8 set_submap_offset

```
virtual function void set_submap_offset (
    uvm_reg_map submap,
    uvm_reg_addr_t offset
)
```

Specifies the offset of the given *submap* as *offset*.

18.2.3.9 set_base_addr

```
virtual function void set_base_addr (
    uvm_reg_addr_t offset
)
```

Specifies the base address of this map.

18.2.3.10 reset

```
virtual function void reset(
    string kind = "SOFT"
)
```

Resets the mirror for all registers in this address map.

Sets the mirror value of all registers in this address map and all of its submaps to the reset value corresponding to the specified reset event. See 18.5.5.4 for more details.

This does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror.

Note that, unlike the other **reset** method (see 18.5.5.4), the default reset event for this method is **"SOFT"**.

18.2.3.11 unregister

```
virtual function void unregister()
```

Disassociates all content (registers, memories, virtual registers, submaps, etc.) from this map.

18.2.3.12 clone_and_update

```
virtual function clone_and_update(
   string rights
)
```

Clones the current map into a new map instance. In contrast to the source map, the new map has the rights for the content set to *rights*.

18.2.4 Introspection

18.2.4.1 get_root_map

```
virtual function uvm_reg_map get_root_map()
```

Returns the externally visible address map.

Returns the top-most address map where this address map is instantiated, which corresponds to the externally visible address map that can be accessed by the verification environment.

18.2.4.2 get_parent

```
virtual function uvm_reg_block get_parent()
```

This returns the block that is the parent of this address map.

18.2.4.3 get_parent_map

```
virtual function uvm_reg_map get_parent_map()
```

This returns the address map in which this address map is mapped. This returns *null* if this is a top-level address map.

18.2.4.4 get_base_addr

```
virtual function uvm_reg_addr_t get_base_addr (
   uvm_hier_e hier = UVM_HIER
)
```

Returns the base offset address for this map. If this map is the root map, the base address is that set with the *base_addr* argument to **uvm_reg_block::create_map** (see 18.1.2.3). If this map is a submap of a higher-level map, the base address is submap offset added to the parent map base address. See 18.2.3.8. The default value of *hier* shall be UVM_HIER.

18.2.4.5 get_n_bytes

```
virtual function int unsigned get_n_bytes (
   uvm_hier_e hier = UVM_HIER
)
```

Returns the width in bytes of the bus associated with this map. If *hier* is UVM_HIER, this retrieves the effective bus width relative to the system level. The effective bus width is the narrowest bus width from this map to the top-level root map. Each bus access is limited to this bus width. The default value of *hier* shall be UVM_HIER.

18.2.4.6 get_addr_unit_bytes

```
virtual function int unsigned get_addr_unit_bytes()
```

Returns the number of bytes in the smallest addressable unit in the map. Returns 1 if the address map was configured using byte-level addressing. Returns **get_n_bytes** otherwise (see 18.2.4.5).

18.2.4.7 get_endian

```
virtual function uvm_endianness_e get_endian (
   uvm_hier_e hier = UVM_HIER
)
```

Returns the endianness (see 17.2.2.4) of the bus associated with this map. If *hier* is set to UVM_HIER (see 17.2.2.7), returns the system-level endianness.

18.2.4.8 get_sequencer

```
virtual function uvm_sequencer_base get_sequencer (
   uvm_hier_e hier = UVM_HIER
)
```

Returns the sequencer for the bus associated with this map. If *hier* is set to UVM_HIER, this returns the sequencer for the bus at the system-level. The default value of *hier* shall be UVM_HIER. See 18.2.3.6.

18.2.4.9 get_adapter

```
virtual function uvm_reg_adapter get_adapter (
   uvm_hier_e hier = UVM_HIER
)
```

Returns the bus adapter for the bus associated with this map. If *hier* is set to UVM_HIER, this returns the adapter for the bus used at the system level. The default value of *hier* shall be UVM_HIER. See 18.2.3.6.

18.2.4.10 get_submaps

```
virtual function void get_submaps (
   ref uvm_reg_map maps[$],
   input uvm_hier_e hier = UVM_HIER
)
```

Returns the address sub-maps instantiated in this address map. If *hier* is UVM_HIER, this recursively includes the address maps in the sub-maps. *maps* shall be a queue. The default value of *hier* shall be UVM_HIER.

18.2.4.11 get_registers

```
virtual function void get_registers (
    ref uvm_reg regs[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the registers instantiated in this address map. If *hier* is UVM_HIER, this recursively includes the registers in the sub-maps. *regs* shall be a queue. The default value of *hier* shall be UVM_HIER.

18.2.4.12 get_fields

```
virtual function void get_fields (
    ref uvm_reg_field fields[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the fields in the registers instantiated in this address map. If *hier* is UVM_HIER, this recursively includes the fields of the registers in the sub-maps. The default value of *hier* shall be UVM_HIER. *fields* shall be a queue.

18.2.4.13 get_memories

```
extern virtual function void get_memories (
    ref uvm_mem mems[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the memories instantiated in this address map. If *hier* is UVM_HIER, recursively includes the memories in the sub-maps. The default value of *hier* shall be UVM_HIER. *mems* shall be a queue.

18.2.4.14 get_virtual_registers

```
virtual function void get_virtual_registers (
    ref uvm_vreg regs[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the virtual registers instantiated in this address map. If *hier* is UVM_HIER, this recursively includes the virtual registers in the sub-maps. The default value of *hier* shall be UVM_HIER. *regs* shall be a queue.

18.2.4.15 get_virtual_fields

```
virtual function void get_virtual_fields (
    ref uvm_vreg_field fields[$],
    input uvm_hier_e hier = UVM_HIER
)
```

Returns the virtual fields instantiated in this address map. If *hier* is UVM_HIER, this recursively includes the virtual fields in the sub-maps. The default value of *hier* shall be UVM_HIER. *fields* shall be a queue.

18.2.4.16 get_physical_addresses

```
virtual function int get_physical_addresses(
   uvm_reg_addr_t base_addr,
   uvm_reg_addr_t mem_offset,
   int unsigned n_bytes,
   ref uvm_reg_addr_t addr[]
)
```

This translates a local address into external addresses.

This identifies the sequence of addresses that need to be accessed physically to access the specified number of bytes at the specified address within this address map. This returns the number of bytes of valid data in each access.

The return in *addr* is a list of address in little endian order, with the granularity of the top-level address map. A register is specified using a base address with a *mem_offset* of `0`. A location within a memory is specified using the base address of the memory and the index of the location within that memory.

18.2.4.17 get_reg_by_offset

```
virtual function uvm_reg get_reg_by_offset(
   uvm_reg_addr_t offset,
   bit read = 1
)
```

Returns the register mapped at *offset*.

This identifies the register located at the specified *offset* within this address map for the specified type of access. This returns *null* if no such register is found. The default value of *read* shall be `1`.

The model needs to be locked using **uvm_reg_block::lock_model** (see 18.1.2.5) to enable this functionality.

18.2.4.18 get_mem_by_offset

```
virtual function uvm_mem get_mem_by_offset(
   uvm_reg_addr_t offset
)
```

Returns the memory mapped at *offset*.

This identifies the memory located at the specified *offset* within this address map. The *offset* may refer to any memory location in that memory. This returns *null* if no such memory is found.

The model needs to be locked using **uvm_reg_block::lock_model** (see 18.1.2.5) to enable this functionality.

18.2.5 Bus access

18.2.5.1 get_auto_predict

```
virtual function bit get_auto_predict()
```

Returns the auto-predict mode setting for this map.

18.2.5.2 set_auto_predict

```
virtual function void set_auto_predict(
   bit on = 1
)
```

Specifies the auto-predict mode for this map.

When *on* is `1`, the register model automatically updates its mirror (what it thinks should be in the DUT) immediately after any bus read or write operation via this map. Before a **uvm_reg::write** (see 18.4.4.9) or **uvm_reg::read** operation (see 18.4.4.10) returns, the register's **uvm_reg::predict** method (see 18.4.4.15) is called to update the mirrored value in the register. The default value of *on* shall be `1`, which is `TRUE`.

When *on* is *False* (the default), bus reads and writes via this map do not automatically update the mirror. For real-time updates to the mirror in this mode, connect a **uvm_reg_predictor** instance (see 19.3) to the bus monitor. The predictor takes observed bus transactions from the bus monitor, looks up the associated **uvm_reg** register (see 18.4) given the address, then calls that register's **uvm_reg::predict** method (see 18.4.4.15). While more complex, this mode captures all register read/write activity, including any not directly descendant from calls to **uvm_reg::write** (see 18.4.4.9) and **uvm_reg::read** (see 18.4.4.10).

18.2.5.3 set_check_on_read

```
virtual function void set_check_on_read(
   bit on = 1
)
```

Specifies the check-on-read mode for this map and all of its submaps.

When *on* is `1`, the register model automatically checks any value read back from a register or field against the current value in its mirror and report any discrepancy. This effectively combines the functionality of the **uvm_reg::read** (see 18.4.4.10) and **uvm_reg::mirror(UVM_CHECK)** (see 18.4.4.14) methods. This mode is useful when the register model is used passively. The default value of *on* shall be `1`, which is `TRUE`.

When *on* is *False* (the default), no check is made against the mirrored value.

At the end of the read operation, the mirror value is updated based on the value that was read regardless of this mode setting.

18.2.5.4 get_transaction_order_policy

```
virtual function uvm_reg_transaction_order_policy
   get_transaction_order_policy()
```

Returns the transaction order policy.

18.2.5.5 set_transaction_order_policy

```
virtual function void set_transaction_order_policy(
   uvm_reg_transaction_order_policy pol
)
```

Specifies the transaction order policy.

18.3 uvm_reg_file

The register file abstraction base class.

A register file is a collection of register files and registers used to create regular repeated structures.

18.3.1 Class declaration

```
class uvm_reg_file extends uvm_object
```

18.3.2 Methods

18.3.2.1 new

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

Creates a new instance of a register file abstraction class with the specified *name*.

18.3.2.2 configure

```
function void configure (
   uvm_reg_block blk_parent,
   uvm_reg_file regfile_parent,
   string hdl_path = ""
)
```

Configures a register file instance.

This specifies the parent block and register file of the register file instance. If the register file is instantiated in a block, *regfile_parent* is specified as *null*. If the register file is instantiated in a register file, *blk_parent* shall be the block parent of that register file and *regfile_parent* is specified as that register file.

If the register file corresponds to a hierarchical RTL structure, its contribution to the HDL path is specified as the *hdl_path*. Otherwise, the register file does not correspond to a hierarchical RTL structure (i.e., it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers.

18.3.3 Introspection

18.3.3.1 get_parent

```
virtual function uvm_reg_block get_parent()
```

Returns the parent block.

18.3.3.2 get_regfile

```
virtual function uvm_reg_file get_regfile()
```

Returns the parent register file.

This returns *null* if this register file is instantiated in a block.

18.3.4 Back door

18.3.4.1 clear_hdl_path

```
function void clear_hdl_path (
   string kind = "RTL"
)
```

This removes any previously specified HDL paths to the register file instance for the specified design abstraction. The default value of *kind* shall be `"RTL"`.

18.3.4.2 add_hdl_path

```
function void add_hdl_path (
   string path,
   string kind = "RTL"
)
```

This adds the specified HDL path to the register file instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the register file is physically duplicated in the design abstraction. The default value of *kind* shall be `"RTL"`.

18.3.4.3 has_hdl_path

```
function bit has_hdl_path (
   string kind = ""
)
```

This returns *True* if the register file instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, it uses the default design abstraction specified for the nearest enclosing register file or block.

18.3.4.4 get_hdl_path

```
function void get_hdl_path (
   ref string paths[$],
   input string kind = ""
)
```

This returns the HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, it uses the default design abstraction specified for the nearest enclosing register file or block. Only the component of the HDL paths that corresponds to the register file is returned, not a full hierarchical path. *paths* shall be a queue.

18.3.4.5 get_full_hdl_path

```
function void get_full_hdl_path (
   ref string paths[$],
   input string kind = ""
)
```

This returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, it uses the default design abstraction specified for the nearest enclosing register file or block. If any of the parent components have more than one path defined for the

same design abstraction, there may be more than one path returned, even if only one path was defined for the register file instance. *paths* shall be a queue.

18.3.4.6 get_default_hdl_path

```
function string get_default_hdl_path()
```

This returns the default design abstraction for this register file instance. If a default design abstraction has not been explicitly specified for this register file instance, it returns the default design abstraction for the nearest register file or block ancestor. The default is "RTL".

18.3.4.7 set_default_hdl_path

```
function void set_default_hdl_path (
    string kind
)
```

Specifies the default design abstraction for this register file instance.

18.4 uvm_reg

The register abstraction base class.

A register represents a set of fields that are accessible as a single entity. A register may be mapped to one or more address maps, each with different access rights and policy.

18.4.1 Class declaration

```
class uvm_reg extends uvm_object
```

18.4.2 Methods

18.4.2.1 new

Creates a new instance and type-specific configuration.

```
function new (
    string name = "",
    int unsigned n_bits,
    int has_coverage
)
```

This creates an instance of a register abstraction class with the specified name.

n_bits specifies the total number of bits in the register. Not all bits need to be implemented.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the **uvm_coverage_model_e** type (see 17.2.2.9).

18.4.2.2 configure

```
function void configure (
    uvm_reg_block blk_parent,
```

```
    uvm_reg_file regfile_parent = null,
    string hdl_path = ""
)
```

This is an instance-specific configuration. It specifies the parent block of this register. This may also set a parent register file for this register.

If the register is implemented in a single HDL variable, its name is specified as the *hdl_path*. Otherwise, if the register is implemented as a concatenation of variables (usually one per field), then the HDL path shall be specified using the **add_hdl_path** (see 18.4.6.4) or **add_hdl_path_slice** (see 18.4.6.5) methods.

18.4.2.3 set_offset

```
virtual function void set_offset (
    uvm_reg_map map,
    uvm_reg_addr_t offset,
    bit unmapped = 0
)
```

This modifies the *offset* of the register.

The *offset* of a register within an address map is set using the **uvm_reg_map::add_reg** method (see 18.2.3.3). This method is used to modify that offset dynamically.

Modifying the offset of a register shall make the register model diverge from the specification that was used to create it. The default value of *unmapped* shall be `0`.

18.4.2.4 uvm_reg_transaction_order_policy

uvm_reg_transaction_order_policy has the following *Methods*.

order

```
pure virtual function void order(
    ref uvm_reg_bus_op q[$]
)
```

The **order** function may reorder the sequence of bus transactions produced by a single **uvm_reg** transaction (read/write) (see 18.4.4). This can be used in scenarios when the register width differs from the bus width and one register access results in a series of bus transactions. *q* shall be a queue.

The first item (`0`) of the queue is the first bus transaction; the last item (`$`) is the final transaction.

18.4.2.5 unregister

```
virtual function void unregister(
    uvm_reg_map map
)
```

Removes the association that the current register instance resides in *map*.

18.4.3 Introspection

18.4.3.1 get_parent

```
virtual function uvm_reg_block get_parent()
```

Returns the parent block.

18.4.3.2 get_regfile

```
virtual function uvm_reg_file get_regfile()
```

Returns the parent register file.

This returns *null* if this register is instantiated in a block.

18.4.3.3 get_n_maps

```
virtual function int get_n_maps()
```

Returns the number of address maps where this register is mapped.

18.4.3.4 is_in_map

```
function bit is_in_map (
   uvm_reg_map map
)
```

Returns `1` if this register is in the specified address *map*.

18.4.3.5 get_maps

```
virtual function void get_maps (
   ref uvm_reg_map maps[$]
)
```

Returns all of the address *maps* where this register is mapped. *maps* shall be a queue.

18.4.3.6 get_rights

```
virtual function string get_rights (
   uvm_reg_map map = null
)
```

Returns the accessibility (**"RW"**, **"RO"**, or **"WO"**) of this register in the given *map*.

If *map* is *null* and the register is mapped in only one address map, that address map is used. If *map* is *null* and the register is mapped in more than one address map, the default address map of the parent block is used.

Whether a register field can be read or written depends on both the field's configured access policy (see 18.5.4.6) and the register's accessibility rights in the map being used to access the field.

If an address map is specified and the register is not mapped in the specified address map, an error message shall be generated and `"RW"` is returned.

18.4.3.7 get_n_bits

```
virtual function int unsigned get_n_bits()
```

Returns the width, in bits, of this register.

18.4.3.8 get_n_bytes

```
virtual function int unsigned get_n_bytes()
```

Returns the width, in bytes, of this register. Rounds up to next whole byte if the register is not a multiple of 8.

18.4.3.9 get_max_size

```
static function int unsigned get_max_size()
```

Returns the maximum width, in bits, of all registers.

18.4.3.10 get_fields

```
virtual function void get_fields (
    ref uvm_reg_field fields[$]
)
```

Fills the specified array with the abstraction class for all of the fields contained in this register. Fields are ordered from least-significant position to most-significant position within the register. *fields* shall be a queue.

18.4.3.11 get_field_by_name

```
virtual function uvm_reg_field get_field_by_name(
    string name
)
```

Returns the named field in this register.

Finds a field with the specified name in this register and returns its abstraction class. If no fields are found, this returns *null*.

18.4.3.12 get_offset

```
virtual function uvm_reg_addr_t get_offset (
    uvm_reg_map map = null
)
```

Returns the offset of this register in an address *map*.

If *map* is *null* and the register is mapped in only one address map, that address map is used. If *map* is *null* and the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, a warning message shall be issued and `-1` is returned.

18.4.3.13 get_address

```
virtual function uvm_reg_addr_t get_address (
    uvm_reg_map map = null
)
```

Returns the base external physical address of this register if accessed through the specified address *map*.

If *map* is *null* and the register is mapped in only one address map, that address map is used. If *map* is *null* and the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, a warning message shall be issued and `-1` is returned.

18.4.3.14 get_addresses

```
virtual function int get_addresses (
    uvm_reg_map map = null,
    ref uvm_reg_addr_t addr[]
)
```

Identifies the external physical address(es) of this register.

This computes all of the external physical addresses that need to be accessed to completely read or write this register. The addressees are specified in little endian order. This returns the number of bytes transferred on each access.

If *map* is *null* and the register is mapped in only one address map, that address map is used. If *map* is *null* and the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, a warning message shall be issued and `-1` is returned.

18.4.4 Access

18.4.4.1 get

```
virtual function uvm_reg_data_t get(
    string fname = "",
    int lineno = 0
)
```

Returns the desired value of the fields in the register. This does not actually read the value of the register in the design, only the desired value in the abstraction class. Unless set to a different value using **uvm_reg::set** (see 18.4.4.2), the desired value and the mirrored value are identical. The default value of *lineno* shall be `0`.

Use the **uvm_reg::read** (see 18.4.4.10) or **uvm_reg::peek** (see 18.4.4.12) methods to retrieve the actual register value.

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and presumed to reside in the bits implementing these fields. Although a physical read operation would return something different for these fields, the returned value is the actual content.

18.4.4.2 set

```
virtual function void set (
    uvm_reg_data_t value,
    string fname = "",
    int lineno = 0
)
```

Specifies the desired *value* of the fields in the register. This does not actually set the value of the register in the design, only the desired value in its corresponding abstraction class in the register model. Use the

uvm_reg::update method (see 18.4.4.13) to update the actual register with the mirrored value or the **uvm_reg::write** method (see 18.4.4.9) to specify the actual register and its mirrored value. The default value of *lineno* shall be `0`.

Unless this method is used, the desired value is equal to the mirrored value.

Refer to 18.5.5.2 for more details on the effect of setting mirror values on fields with different access policies.

To modify the mirrored field values to a specific value, and thus use the mirrored values as a scoreboard for the register values in the DUT, use the **uvm_reg::predict** method (see 18.4.4.15).

18.4.4.3 get_mirrored_value

```
virtual function uvm_reg_data_t get_mirrored_value(
    string fname = "",
    int lineno = 0
)
```

Returns the mirrored value of the fields in the register. This does not actually read the value of the register in the design. The default value of *lineno* shall be `0`.

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and presumed to reside in the bits implementing these fields.

Although a physical read operation would return something different for these fields, the returned value is the actual content.

18.4.4.4 needs_update

```
virtual function bit needs_update()
```

Returns `1` if any of the fields need updating.

See 18.5.5.8 for details. Use **uvm_reg::update** (see 18.4.4.13) to actually update the DUT register.

18.4.4.5 reset

```
virtual function void reset(
    string kind = "HARD"
)
```

Resets the desired/mirrored value for this register.

This sets the desired and mirror value of the fields in this register to the reset value for the specified reset *kind*. The default value of *kind* shall be `"HARD"`. See 18.5.5.4 for more details.

It also resets the semaphore that prevents concurrent access to the register. This semaphore needs to be explicitly reset if a thread accessing this register array was killed before the access was completed.

18.4.4.6 get_reset

```
virtual function uvm_reg_data_t get_reset(
    string kind = "HARD"
)
```

Returns the specified reset value for this register; this returns the reset value for this register for the specified reset *kind*. The default value of *kind* shall be "HARD".

18.4.4.7 has_reset

```
virtual function bit has_reset(
    string kind = "HARD",
    bit delete = 0
)
```

Checks if any field in the register has a reset value specified for the specified reset *kind*. The default value of *kind* shall be "HARD".

If *delete* is TRUE, this removes the reset value, if any. The default value of *delete* shall be 0, which is FALSE.

18.4.4.8 set_reset

```
virtual function void set_reset(
    uvm_reg_data_t value,
    string kind = "HARD"
)
```

Specifies or modifies the reset *value* for all the fields in the register corresponding to the cause specified by *kind*. The default value of *kind* shall be "HARD".

18.4.4.9 write

```
virtual task write(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This initiates a write (using *value* for data) to the register in the design that corresponds to this abstraction class instance.

The write may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the register will remain unchanged.

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.4.4.10 read

```
virtual task read(
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads and returns the current *value* from the register in the design that corresponds to this abstraction class instance.

The read may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of reading the register through a physical access is mimicked. For example, clear-on-read bits in the registers are set to zero (`0`).

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.4.4.11 poke

```
virtual task poke(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This deposits the *value* in the DUT register corresponding to this abstraction class instance, as is, using a back-door access. Uses the HDL path for the design abstraction specified by *kind*. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the

uvm_reg_backdoor::read (see 19.5.2.7) and **uvm_reg_backdoor::write** (see 19.5.2.6) methods. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.4.4.12 peek

```
virtual task peek(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This samples the *value* in the DUT register corresponding to this abstraction class instance using a back-door access. The register value is sampled, not modified. Uses the HDL path for the design abstraction specified by *kind*. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_backdoor::read** method. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.4.4.13 update

```
virtual task update(
    output uvm_status_e status,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Updates the content of the register in the design with the mirrored value. Use the method **uvm_reg::needs_update** (see 18.4.4.4) to determine if an update is necessary.

The update may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the register through a physical access is mimicked (see 18.4.4.11). For example, read-only bits in the register will remain unchanged.

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the

uvm_reg_adapter (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method performs the reverse operation of **uvm_reg::mirror** (see 18.4.4.14).

18.4.4.14 mirror

```
virtual task mirror(
    output uvm_status_e status,
    input uvm_check_e check = UVM_NO_CHECK,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads the register and optionally compares the read back value with the current mirrored value if *check* is `UVM_CHECK` (see 17.2.2.3). The mirrored value is then updated using the **uvm_reg::predict** method (see 18.4.4.15), based on the read back value.

The mirroring may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the register through a physical access is mimicked (see 18.4.4.11). The content of write-only fields is mirrored and optionally checked.

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

If *check* is specified as `UVM_CHECK`, an error message shall be generated if the current mirrored value does not match the read back value. Any field whose check has been disabled with **uvm_reg_field::set_compare** (see 18.5.5.15) shall not be considered in the comparison.

This method performs the reverse operation of **uvm_reg::update** (see 18.4.4.13).

18.4.4.15 predict

```
virtual function bit predict (
    uvm_reg_data_t value,
    uvm_reg_byte_en_t be = -1,
```

```
    uvm_predict_e kind = UVM_PREDICT_DIRECT,
    uvm_door_e path = UVM_FRONTDOOR,
    uvm_reg_map map = null,
    string fname = "",
    int lineno = 0
)
```

Updates the mirrored and desired *value* for this register. The default value of *be* shall be `-1`. The default value of *kind* shall be `UVM_PREDICT_DIRECT`. The default value of *path* shall be `UVM_FRONTDOOR`. The default value of *lineno* shall be `0`.

This predicts the mirror (and desired) value of the fields in the register based on the specified observed *value* on a specified address *map*, or based on a calculated value. See 18.5.5.17 for more details.

This returns `TRUE` if the prediction was successful for each field in the register.

18.4.4.16 is_busy

```
function bit is_busy()
```

Returns `1` if the register is currently being read or written.

18.4.5 Front door

18.4.5.1 get_frontdoor

```
function uvm_reg_frontdoor get_frontdoor(
    uvm_reg_map map = null
)
```

Returns the user-defined front door for this register.

If *null*, no user-defined front door has been defined. A user-defined front door is defined by using the **uvm_reg::set_frontdoor** method (see 18.4.5.2).

If the register is mapped in multiple address maps, an address *map* shall be specified.

18.4.5.2 set_frontdoor

```
function void set_frontdoor(
    uvm_reg_frontdoor ftdr,
    uvm_reg_map map = null,
    string fname = "",
    int lineno = 0
)
```

Specifies a user-defined front door for this register. The default value of *lineno* shall be `0`.

By default, registers are mapped linearly into the address space of the address maps that instantiate them. If registers are accessed using a different mechanism, a user-defined access mechanism shall be defined and associated with the corresponding register abstraction class.

If the register is mapped in multiple address maps, an address *map* shall be specified.

18.4.6 Back door

18.4.6.1 get_backdoor

```
function uvm_reg_backdoor get_backdoor(
   bit inherited = 1
)
```

Returns the user-defined back door for this register.

If *null*, no user-defined back door has been defined. A user-defined back door is defined by using the **uvm_reg::set_backdoor** method (see 18.4.6.1).

If *inherited* is `TRUE`, this returns the back door of the parent block if none have been specified for this register. The default value of *inherited* shall be `1`, which is `TRUE`.

18.4.6.2 set_backdoor

```
function void set_backdoor(
   uvm_reg_backdoor bkdr,
   string fname = "",
   int lineno = 0
)
```

Specifies a user-defined back door for this register.The default value of *lineno* shall be `0`.

By default, registers are accessed via the built-in string-based DPI routines if an HDL path has been specified using the **uvm_reg::configure** (see 18.3.2.2) or **uvm_reg::add_hdl_path** (see 18.4.6.4) methods.

If this default mechanism is not suitable (e.g., because the register is not implemented in pure SystemVerilog) a user-defined access mechanism needs to be defined and associated with the corresponding register abstraction class. A user-defined back door is required if an active update of the mirror of this register abstraction class, based on observed changes of the corresponding DUT register, is used.

18.4.6.3 clear_hdl_path

```
function void clear_hdl_path (
   string kind = "RTL"
)
```

Deletes any HDL paths. The default value of *kind* shall be `"RTL"`.

This removes any previously specified HDL path to the register instance for the specified design abstraction.

18.4.6.4 add_hdl_path

```
function void add_hdl_path (
   uvm_hdl_path_slice slices[],
   string kind = "RTL"
)
```

Adds the specified HDL path to the register instance for the specified design abstraction. The default value of *kind* shall be `"RTL"`.

This method may be called more than once for the same design abstraction if the register is physically duplicated in the design abstraction.

18.4.6.5 add_hdl_path_slice

```
function void add_hdl_path_slice(
   string name,
   int offset,
   int size,
   bit first = 0,
   string kind = "RTL"
)
```

Appends the specified HDL slice to the HDL path of the register instance for the specified design abstraction. If *first* is `TRUE`, this starts the specification of a duplicate HDL implementation of the register. The default value of *first* shall be `0`, which is `FALSE`. The default value of *kind* shall be `"RTL"`.

18.4.6.6 has_hdl_path

```
function bit has_hdl_path (
   string kind = ""
)
```

Checks if a HDL path is specified.

This returns *True* if the register instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, it uses the default design abstraction specified for the parent block.

18.4.6.7 get_hdl_path

```
function void get_hdl_path (
       ref uvm_hdl_path_concat paths[$],
   input string kind = ""
)
```

Returns the incremental HDL path(s).

This returns the HDL path(s) defined for the specified design abstraction in the register instance. It returns only the component of the HDL paths that corresponds to the register, not a full hierarchical path. *paths* shall be a queue.

If no design abstraction is specified, the default design abstraction for the parent block is used.

18.4.6.8 get_hdl_path_kinds

```
function void get_hdl_path_kinds (
   ref string kinds[$]
)
```

Returns any design abstractions for which HDL paths have been defined. *kinds* shall be a queue.

18.4.6.9 get_full_hdl_path

```
function void get_full_hdl_path (
       ref uvm_hdl_path_concat paths[$],
```

```
    input string kind = "",
    input string separator = "."
)
```

Returns the full hierarchical HDL path(s).

This returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register instance. There may be more than one path returned even if only one path was defined for the register instance, if any of the parent components have more than one path defined for the same design abstraction. *paths* shall be a queue.

If no design abstraction is specified, the default design abstraction for each ancestor block is used to retrieve each incremental path. The default value of *separator* shall be `"."`.

18.4.6.10 backdoor_read

```
virtual task backdoor_read(
    uvm_reg_item rw
)
```

User-defined back-door read access.

The implementation shall use the UVM HDL back-door access support routines (see 19.6) to perform a read for this register.

18.4.6.11 backdoor_write

```
virtual task backdoor_write(
    uvm_reg_item rw
)
```

User-defined back-door write access.

This overrides the default string-based DPI back-door access write for this register type.

18.4.6.12 backdoor_watch

```
virtual task backdoor_watch()
```

User-defined DUT register change monitor.

This watches the DUT register corresponding to this abstraction class instance for any change in value and return when a value change occurs.

18.4.7 Coverage

18.4.7.1 include_coverage

```
static function void include_coverage(
    string scope,
    uvm_reg_cvr_t models,
    uvm_object accessor = null
)
```

Specifies which coverage model shall be included in various block, register, or memory abstraction class instances.

The coverage models are specified by `OR`ing or adding the **uvm_coverage_model_e** coverage model identifiers (see 17.2.2.9) corresponding to the coverage model to be included.

The *scope* specifies a hierarchical name or pattern identifying a block, memory, or register abstraction class instances. Any block, memory, or register whose full hierarchical name matches the specified scope shall have the specified functional coverage models included in them.

The *scope* can be specified as a POSIX® regular expression or simple pattern. See C.2.4 for more details.[7]

The specification of which coverage model to include in which abstraction class is stored in a **uvm_reg_cvr_t** resource (see 17.2.1.6) in the **uvm_resource_db** resource database (see C.3.2), in the `uvm_reg::` scope namespace.

18.4.7.2 build_coverage

```
protected function uvm_reg_cvr_t build_coverage(
   uvm_reg_cvr_t models
)
```

Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9). This returns the sum of all coverage *models* to be built in the register model.

18.4.7.3 add_coverage

```
virtual protected function void add_coverage(
   uvm_reg_cvr_t models
)
```

Specifies that additional coverage models are available.

This adds the specified coverage model to the coverage *models* available in this class. Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9).

This method shall only be called in the constructor of subsequently derived classes.

18.4.7.4 has_coverage

```
virtual function bit has_coverage(
   uvm_reg_cvr_t models
)
```

Checks if the register has coverage model(s).

This returns *True* if the register abstraction class contains a coverage model for all of the *models* specified. Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9).

[7]POSIX is a registered trademark in the U.S. Patent & Trademark Office, owned by The Institute of Electrical and Electronics Engineers, Incorporated.

18.4.7.5 get_coverage

```
virtual function bit get_coverage(
   uvm_reg_cvr_t is_on
)
```

Checks if coverage measurement is on.

This returns `1` if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See 18.4.7.6 for more details.

18.4.7.6 set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(
   uvm_reg_cvr_t is_on
)
```

Turns on coverage measurement.

This turns the collection of functional coverage measurements on or off for this register. The functional coverage measurement is turned on for every coverage model specified using **uvm_coverage_model_e** coverage model identifiers (see 17.2.2.9). Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. This returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the register abstraction classes, then enabled during construction. See the **uvm_reg::has_coverage** method (see 18.4.7.4) to identify the available functional coverage models.

18.4.7.7 sample

```
protected virtual function void sample(
   uvm_reg_data_t data,
   uvm_reg_data_t byte_en,
   bit is_read,
   uvm_reg_map map
)
```

This is a functional coverage measurement method.

This method is invoked by the register abstraction class whenever it is read or written with the specified *data* via the specified address *map*. It is invoked after the read or write operation has completed, but before the mirror has been updated.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

18.4.7.8 sample_values

```
virtual function void sample_values()
```

This is a functional coverage measurement method for field values.

This method is invoked by the user or by the **uvm_reg_block::sample_values** method (see 18.1.4.7) of the parent block to trigger the sampling of the current field values in the register-level functional coverage model.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model.

18.4.8 Callbacks

18.4.8.1 pre_write

```
virtual task pre_write(
    uvm_reg_item rw
)
```

Called before register write.

If the specified data value, access *path*, or address *map* are modified, the updated data value, access path, or address map is used to perform the register operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks.

18.4.8.2 post_write

```
virtual task post_write(
    uvm_reg_item rw
)
```

Called after register write.

If the specified *status* is modified, the updated status is returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks.

18.4.8.3 pre_read

```
virtual task pre_read(
    uvm_reg_item rw
)
```

Called before register read.

If the specified access *path* or address *map* are modified, the updated access path or address map is used to perform the register operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks.

18.4.8.4 post_read

```
virtual task post_read(
   uvm_reg_item rw
)
```

Called after register read.

If the specified read back data or *status* is modified, the updated read back data or status is returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks.

18.5 uvm_reg_field

The field abstraction class.

A field represents a set of bits that behave consistently as a single entity.

A field is contained within a single register, but may have different access policies depending on the address map used to access the register (thus the field).

18.5.1 Class declaration

```
class uvm_reg_field extends uvm_object
```

18.5.2 Common methods

value

```
rand uvm_reg_data_t value
```

This is a mirrored field value; it can be sampled in a functional coverage model or constrained when randomized.

18.5.3 Methods

18.5.3.1 new

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

Initializes a new field instance.

18.5.3.2 configure

```
function void configure(
   uvm_reg parent,
   int unsigned size,
   int unsigned lsb_pos,
   string access,
   bit volatile,
   uvm_reg_data_t reset,
```

```
    bit has_reset,
    bit is_rand,
    bit individually_accessible
)
```

This is an instance-specific configuration.

It specifies the *parent* register of this field, its *size* in bits, the position of its LSB within the register relative to the LSB of the register, its *access* policy, volatility, "HARD" *reset* value, whether the field value is actually reset (the *reset* value is ignored if `FALSE`), whether the field value may be randomized and whether the field is the only one to occupy a byte lane in the register.

See 18.5.4.6 for a specification of the predefined field access policies.

If the field access policy is a predefined policy and NOT one of "`RW`", "`WRC`", "`WRS`", "`WO`", "`W1`", or "`WO1`", the value of *is_rand* is ignored and the `rand_mode` for the field instance is turned off since it cannot be written.

18.5.4 Introspection

18.5.4.1 get_parent

```
virtual function uvm_reg get_parent()
```

Returns the parent register.

18.5.4.2 get_lsb_pos

```
virtual function int unsigned get_lsb_pos()
```

Returns the position of the field.

This returns the index of the least significant bit (LSB) of the field in the register that instantiates it. An offset of `0` indicates a field that is aligned with the LSB of the register.

18.5.4.3 get_n_bits

```
virtual function int unsigned get_n_bits()
```

Returns the width, in number of bits, of the field.

18.5.4.4 get_max_size

```
static function int unsigned get_max_size()
```

Returns the width, in number of bits, of the largest field.

18.5.4.5 get_access

```
virtual function string get_access(
    uvm_reg_map map = null
)
```

Returns the access policy of the field.

This returns the current access policy of the field when written and read through the specified address *map*. If the register containing the field is mapped in multiple address maps, an address map shall be specified. The access policy of a field from a specific address map may be restricted by the register's access policy in that address map, e.g., a `RW` field may only be writable through one of the address maps and read-only through all of the other maps.

If the field access contradicts the map's access value (e.g., a field access of `WO` and map access value of `RO`), the method's return value is `"NOACCESS"` (see 18.5.4.6).

18.5.4.6 set_access

```
virtual function string set_access(
    string mode
)
```

Modifies the access policy of the field to the specified one and return the previous access policy.

The predefined access policies are as follows (W = write; R= read). The effects of a read operation are applied after the current value of the field is sampled. The read operation returns the current value, not the value affected by the read operation (if any). For R, the "no effect" behavior either returns `0`'s or the return value; an "error" needs to define an error for negative testing.

a) **"RO"**—W: no effect, R: no effect.
b) **"RW"**—W: as is, R: no effect.
c) **"RC"**—W: no effect, R: clears all bits.
d) **"RS"**—W: no effect, R: sets all bits.
e) **"WRC"**—W: as is, R: clears all bits.
f) **"WRS"**—W: as is, R: sets all bits.
g) **"WC"**—W: clears all bits, R: no effect.
h) **"WS"**—W: sets all bits, R: no effect.
i) **"WSRC"**—W: sets all bits, R: clears all bits.
j) **"WCRS"**—W: clears all bits, R: sets all bits.
k) **"W1C"**—W: `1/0` clears/no effect on matching bit, R: no effect.
l) **"W1S"**—W: `1/0` sets/no effect on matching bit, R: no effect.
m) **"W1T"**—W: `1/0` toggles/no effect on matching bit, R: no effect.
n) **"W0C"**—W: `1/0` no effect on/clears matching bit, R: no effect.
o) **"W0S"**—W: `1/0` no effect on/sets matching bit, R: no effect.
p) **"W0T"**—W: `1/0` no effect on/toggles matching bit, R: no effect.
q) **"W1SRC"**—W: `1/0` sets/no effect on matching bit, R: clears all bits.
r) **"W1CRS"**—W: `1/0` clears/no effect on matching bit, R: sets all bits.
s) **"W0SRC"**—W: `1/0` no effect on/sets matching bit, R: clears all bits.
t) **"W0CRS"**—W: `1/0` no effect on/clears matching bit, R: sets all bits.
u) **"WO"**—W: as is, R: error.
v) **"WOC"**—W: clears all bits, R: error.
w) **"WOS"**—W: sets all bits, R: error.
x) **"W1"**—W: first one after HARD reset is as is, other W have no effects, R: no effect.
y) **"WO1"**—W: first one after HARD reset is as is, other W have no effects, R: error.
z) **"NOACCESS"**—W: no effect, R: no effect.

It is important to remember that modifying the access of a field makes the register model diverge from the specification that was used to create it.

18.5.4.7 define_access

```
static function bit define_access(
    string name
)
```

Defines a new access policy value.

Because field access policies are specified using string values, there is no way for SystemVerilog to verify if a specific access value is valid or not. To help catch typing errors, user-defined access values shall be defined using this method to avoid being reported as an invalid access policy.

The name of field access policies are always converted to all uppercase.

This returns `TRUE` if the new access policy was not previously defined. It returns `FALSE` otherwise, but does not issue an error message.

18.5.4.8 is_known_access

```
virtual function bit is_known_access(
    uvm_reg_map map = null
)
```

Checks if the access policy is a built-in one.

This returns `TRUE` if the current access policy of the field, when written and read through the specified address map, is a built-in access policy.

18.5.4.9 set_volatility

```
virtual function void set_volatility(
    bit volatile
)
```

Modifies the volatility of the field (*volatile*) to the specified one.

It is important to remember that modifying the volatility of a field makes the register model diverge from the specification that was used to create it.

18.5.4.10 is_volatile

```
virtual function bit is_volatile()
```

Indicates if the field value is volatile. UVM uses the IEEE Std 1685™ definition of "volatility" [B3].[8]

If `TRUE`, the mirrored value in the register cannot be trusted. This typically indicates a field whose change in value cannot be observed by UVM. The nature or cause of the change is not specified.

If `FALSE`, the mirrored value in the register can be trusted.

[8]The numbers in brackets correspond to those of the bibliography in Annex A.

18.5.5 Access

18.5.5.1 get

```
virtual function uvm_reg_data_t get(
    string fname = "",
    int lineno = 0
)
```

Returns the desired value of the field. The default value of *lineno* shall be `0`.

This does not actually read the value of the field in the design, only the desired value in the abstraction class. Unless set to a different value using **uvm_reg_field::set** (see 18.5.5.2), the desired value and the mirrored value are identical. Use the **uvm_reg_field::read** (see 18.5.5.10) or **uvm_reg_field::peek** (see 18.5.5.12) methods to retrieve the actual field value.

If the field is write-only, the desired/mirrored value is the value last written and presumed to reside in the bits implementing it. Although a physical read operation would return something different, the returned value is the actual content.

18.5.5.2 set

```
virtual function void set(
    uvm_reg_data_t value,
    string fname = "",
    int lineno = 0
)
```

Sets the desired value for this field to the specified *value* modified by the field access policy. This does not actually set the value of the field in the design, only the desired value in the abstraction class. Use the **uvm_reg::update** method (see 18.4.4.13) to update the actual register with the desired value or the **uvm_reg_field::write** method (see 18.5.5.9) to actually write the field and update its mirrored value. The default value of *lineno* shall be `0`.

The final desired value in the mirror is a function of the field access policy and the set value, just like a normal physical write operation to the corresponding bits in the hardware. As such, this method [when eventually followed by a call to **uvm_reg::update** (see 18.4.4.13)] is a zero-time functional replacement for the **uvm_reg_field::write** method (see 18.5.5.9). For example, the desired value of a read-only field is not modified by this method and the desired value of a write-once field can only be set if the field has not yet been written to using a physical (for example, front-door) write operation.

Use the **uvm_reg_field::predict** (see 18.5.5.17) to modify the mirrored value of the field.

18.5.5.3 get_mirrored_value

```
virtual function uvm_reg_data_t get_mirrored_value(
    string fname = "",
    int lineno = 0
)
```

Returns the mirrored value of the field. The default value of *lineno* shall be `0`.

This does not actually read the value of the field in the design, only the mirrored value in the abstraction class.

If the field is write-only, the desired/mirrored value is the value last written and presumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content.

18.5.5.4 reset

```
virtual function void reset(
    string kind = "HARD"
)
```

Resets the desired and mirrored value for this field.

This sets the desired and mirror value of the field to the reset event specified by *kind*. If the field does not have a reset value specified for the specified reset *kind*, the field is unchanged. The default value of *kind* shall be `"HARD"`.

This does not actually reset the value of the field in the design, only the value mirrored in the field abstraction class.

Write-once fields can be modified after a "`HARD`" reset operation.

18.5.5.5 has_reset

```
virtual function bit has_reset(
    string kind = "HARD",
    bit delete = 0
)
```

Check if the field has a reset value specified. The default value of *kind* shall be `"HARD"`.

Return *TRUE* if this field has a reset value specified for the specified reset *kind*. If *delete* is `TRUE`, removes the reset value, if any. The default value of *delete* shall be `0`, which is `FALSE`.

18.5.5.6 get_reset

```
virtual function uvm_reg_data_t get_reset(
    string kind = "HARD"
)
```

Returns the specified reset value for this field.

This returns the reset value for this field for the specified reset *kind*. It returns the current field value is no reset value has been specified for the specified reset event. The default value of *kind* shall be `"HARD"`.

18.5.5.7 set_reset

```
virtual function void set_reset(
    uvm_reg_data_t value,
    string kind = "HARD"
)
```

Specifies or modifies the reset *value* for this field corresponding to the cause specified by *kind*. The default value of *kind* shall be `"HARD"`.

18.5.5.8 needs_update

```
virtual function bit needs_update()
```

Checks if the abstract model contains different desired and mirrored values.

If a desired field value has been modified in the abstraction class without actually updating the field in the DUT, the state of the DUT (and more specifically, what the abstraction class thinks the state of the DUT is) is outdated. This method returns `TRUE` if the state of the field in the DUT needs to be updated to match the desired value. The mirror values or actual content of DUT field are not modified. Use **uvm_reg::update** (see 18.4.4.13) to actually update the DUT field.

18.5.5.9 write

```
virtual task write (
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This writes the *value* in the DUT field that corresponds to this abstraction class instance.

The write may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the register will remain unchanged.

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field supports byte-enabling, then only the field is written. Otherwise, the entire register containing the field is written and the mirrored values of the other fields in the same register are used in a best effort not to modify their value. If a back-door access is used, a peek-modify-poke process is used in a best effort not to modify the value of the other fields in the register.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.5.5.10 read

```
virtual task read (
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads and returns the *value* in the DUT field that corresponds to this abstraction class instance.

The read may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of reading the register through a physical access is mimicked. For example, clear-on-read field bits are set to zero (`0`).

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field supports byte-enabling, then only the field is read. Otherwise, the entire register containing the field is read and the mirrored values of the other fields in the same register are updated. If a back-door access is used, the entire containing register is peeked and the mirrored value of the other fields in the register is updated.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.5.5.11 poke

```
virtual task poke (
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Deposits the specified *value* in this DUT field corresponding to this abstraction class instance, as is, using a back-door access. The entire register shall automatically be peeked prior to the poke operation in order to not modify the value of the other fields in the register. Uses the HDL path for the design abstraction

specified by *kind*. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_backdoor::read** (see 19.5.2.7) and **uvm_reg_backdoor::write** (see 19.5.2.6) methods. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.5.5.12 peek

```
virtual task peek (
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This samples the *value* in the DUT register corresponding to this abstraction class instance using a back-door access. The field value is sampled, not modified. Uses the HDL path for the design abstraction specified by *kind*. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_backdoor::read** method. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

This method is affected by the auto-prediction configuration value (see 18.2.5.2).

18.5.5.13 mirror

```
virtual task mirror(
    output uvm_status_e status,
    input uvm_check_e check = UVM_NO_CHECK,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads the register and optionally compares the read back value with the current mirrored value if *check* is `UVM_CHECK` (see 17.2.2.3). The mirrored value is then updated using the **uvm_reg::predict** method (see 18.4.4.15), based on the read back value.

The mirroring may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the register through a physical access is mimicked (see 18.4.4.11). The content of write-only fields is mirrored and optionally checked.

If front door is specified, and if the register is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

If *check* is specified as `UVM_CHECK`, an error message shall be generated if the current mirrored value does not match the read back value, unless **set_compare** (see 18.5.5.15) was used to disable the check.

18.5.5.14 get_compare

```
function uvm_check_e get_compare()
```

Returns the compare policy for this field.

18.5.5.15 set_compare

```
function void set_compare(
   uvm_check_e check = UVM_CHECK
)
```

Specifies the compare policy during a mirror update. The field value is checked against its mirror only when both the *check* argument in **uvm_reg_block::mirror** (see 18.1.5.6), **uvm_reg::mirror** (see 18.4.4.14), or **uvm_reg_field::mirror** (see 18.5.5.13) and the compare policy for the field is `UVM_CHECK` (see 17.2.2.3).

18.5.5.16 is_indv_accessible

```
function bit is_indv_accessible (
   uvm_door_e path,
   uvm_reg_map local_map
)
```

Checks if this field can be written individually, i.e., without affecting other fields in the containing register.

18.5.5.17 predict

```
function bit predict (
   uvm_reg_data_t value,
   uvm_reg_byte_en_t be = -1,
   uvm_predict_e kind = UVM_PREDICT_DIRECT,
   uvm_door_e path = UVM_FRONTDOOR,
   uvm_reg_map map = null,
   string fname = "",
   int lineno = 0
)
```

Updates the mirrored and desired value for this field. The default value of *be* shall be `-1`. The default value of *kind* shall be `UVM_PREDICT_DIRECT`. The default value of *path* shall be `UVM_FRONTDOOR`. The default value of *lineno* shall be `0`.

This predicts the mirror and desired value of the field based on the specified observed *value* on a bus using the specified address *map*.

If *kind* is specified as `UVM_PREDICT_READ` (see 17.2.2.8), the value was observed in a read transaction on the specified address *map* or back door [if *path* is `UVM_BACKDOOR` (see 17.2.2.2)]. If *kind* is specified as `UVM_PREDICT_WRITE` (see 17.2.2.8), the value was observed in a write transaction on the specified address *map* or back door [if *path* is `UVM_BACKDOOR` (see 17.2.2.2)]. If *kind* is specified as `UVM_PREDICT_DIRECT` (see 17.2.2.8), the value was computed and is updated as is, without regard to any access policy.

This method does not allow an update of the mirror (or desired) values when the register containing this field is busy executing a transaction because the results are unpredictable and indicative of a race condition in the testbench.

This returns `TRUE` if the prediction was successful.

18.5.6 Callbacks

18.5.6.1 pre_write

```
virtual task pre_write(
    uvm_reg_item rw
)
```

Called before field write.

If the specified data value, access *path*, or address *map* are modified, the updated data value, access path, or address map is used to perform the register operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callbacks are invoked after the invocation of this method.

18.5.6.2 post_write

```
virtual task post_write(
    uvm_reg_item rw
)
```

Called after field write.

If the specified *status* is modified, the updated status is returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callbacks are invoked before the invocation of this method.

18.5.6.3 pre_read

```
virtual task pre_read(
    uvm_reg_item rw
)
```

Called before field read.

If the specified access *path* or address *map* are modified, the updated access path or address map is used to perform the register operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callbacks are invoked after the invocation of this method.

18.5.6.4 post_read

```
virtual task post_read(
    uvm_reg_item rw
)
```

Called after field read.

If the specified read back data or *status* is modified, the updated read back data or status is returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callbacks are invoked before the invocation of this method.

18.6 uvm_mem

The memory abstraction base class.

A memory is a collection of contiguous locations. A memory may be accessible via more than one address map.

Unlike registers, memories are not mirrored because of the potentially large data space; tests that walk the entire memory space would negate any benefit from sparse memory modeling techniques.

18.6.1 Class declaration

```
virtual class uvm_mem extends uvm_object
```

18.6.2 Variables

mam

```
uvm_mem_mam mam
```

This is the memory allocation manager for the memory corresponding to this abstraction class instance. It can be used to allocate regions of consecutive addresses of specific sizes, such as DMA buffers, or to locate virtual register array.

18.6.3 Methods

18.6.3.1 new

```
function new (
    string name,
    longint unsigned size,
    int unsigned n_bits,
    string access = "RW",
    int has_coverage = UVM_NO_COVERAGE
)
```

Creates a new instance and type-specific configuration; this creates an instance of a memory abstraction class with the specified *name*.

size specifies the total number of memory locations. *n_bits* specifies the total number of bits in each memory location. *access* specifies the access policy of this memory and may be one of **"RW"** for RAMs and **"RO"** for ROMs. The default value of *access* shall be `"RW"`.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the **uvm_coverage_model_e** type (see 17.2.2.9). The default value of *has_coverage* shall be `UVM_NO_COVERAGE`.

18.6.3.2 configure

```
function void configure (
uvm_reg_block parent,
   string hdl_path = ""
)
```

This is an instance-specific configuration; it specifies the *parent* block of this memory.

If this memory is implemented in a single HDL variable, its name is specified as the *hdl_path*. Otherwise, if the memory is implemented as a concatenation of variables (usually one per bank), then the HDL path shall be specified using the **add_hdl_path** (see 18.6.7.4) or **add_hdl_path_slice** (see 18.6.7.4) methods.

18.6.3.3 set_offset

```
virtual function void set_offset (
   uvm_reg_map map,
   uvm_reg_addr_t offset,
   bit unmapped = 0
)
```

Modifies the offset of the memory.

The offset of a memory within an address map is set using the **uvm_reg_map::add_mem** method (see 18.2.3.4). This method is used to modify that offset dynamically.

Modifying the offset of a memory makes the abstract model diverge from the specification that was used to create it.

18.6.4 Introspection

18.6.4.1 get_parent

```
virtual function uvm_reg_block get_parent()
```

Returns the parent block.

18.6.4.2 get_n_maps

```
virtual function int get_n_maps()
```

Returns the number of address maps mapping this memory.

18.6.4.3 is_in_map

```
function bit is_in_map (
   uvm_reg_map map
)
```

Returns TRUE if this memory is in the specified address map.

18.6.4.4 get_maps

```
virtual function void get_maps (
   ref uvm_reg_map maps[$]
)
```

Returns all of the address maps where this memory is mapped. *maps* shall be a queue.

18.6.4.5 get_rights

```
virtual function string get_rights (
   uvm_reg_map map = null
)
```

Returns the access rights of this memory.

This returns "RW", "RO", or "WO". The access rights of a memory is always "RW", unless it is a shared memory with access restriction in a particular address *map*.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, a warning message shall be issued and "RW" is returned.

18.6.4.6 get_access

```
virtual function string get_access(
   uvm_reg_map map = null
)
```

Returns the access policy of the memory when written and read via an address *map*.

If the memory is mapped in more than one address *map*, an address *map* shall be specified. If access restrictions are present when accessing a memory through the specified address *map*, the access mode returned takes the access restrictions into account, e.g., a read-write memory accessed through a domain with read-only restrictions would return "RO".

18.6.4.7 get_size

```
function longint unsigned get_size()
```

Returns the number of unique memory locations in this memory.

18.6.4.8 get_n_bytes

```
function int unsigned get_n_bytes()
```

Returns the width, in number of bytes, of each memory location. Rounds up to next whole byte if the memory word is not a multiple of 8 bits.

18.6.4.9 get_n_bits

```
function int unsigned get_n_bits()
```

Returns the width, in number of bits, of each memory location.

18.6.4.10 get_max_size

```
static function int unsigned get_max_size()
```

Returns the maximum width, in number of bits, of all memories.

18.6.4.11 get_virtual_registers

```
virtual function void get_virtual_registers(
   ref uvm_vreg regs[$]
)
```

Returns the virtual registers in this memory.

This fills the specified array with the abstraction class for all of the virtual registers implemented in this memory. The order in which the virtual registers are located in the array is not specified. *regs* shall be a queue.

18.6.4.12 get_virtual_fields

```
virtual function void get_virtual_fields(
   ref uvm_vreg_field fields[$]
)
```

Returns the virtual fields in this memory.

This fills the specified dynamic array with the abstraction class for all of the virtual fields implemented in this memory. The order in which the virtual fields are located in the array is not specified. *fields* shall be a queue.

18.6.4.13 get_vreg_by_name

```
virtual function uvm_vreg get_vreg_by_name(
   string name
)
```

Finds a virtual register with the specified *name* implemented in this memory and returns its abstraction class instance. If no virtual register with the specified name is found, this returns *null*.

18.6.4.14 get_vfield_by_name

```
virtual function uvm_vreg_field get_vfield_by_name(
   string name
)
```

Finds a virtual field with the specified *name* implemented in this memory and returns its abstraction class instance. If no virtual field with the specified name is found, this returns *null*.

18.6.4.15 get_offset

```
virtual function uvm_reg_addr_t get_offset (
   uvm_reg_addr_t offset = 0,
   uvm_reg_map map = null
)
```

Returns the base *offset* of the specified location in this memory in an address *map*. The default value of *offset* shall be `0`.

If *map* is *null* and the memory is mapped in only one address map, that address map is used. If *map* is *null* and the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, a warning message shall be issued.

18.6.4.16 get_address

```
virtual function uvm_reg_addr_t get_address(
   uvm_reg_addr_t offset = 0,
   uvm_reg_map map = null
)
```

Returns the base external physical address of the specified location in this memory if accessed through the specified address *map*. The default value of *offset* shall be `0`.

If *map* is *null* and the memory is mapped in only one address map, that address map is used. If *map* is *null* and the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, a warning message shall be issued.

18.6.4.17 get_addresses

```
virtual function int get_addresses(
   uvm_reg_addr_t offset = 0,
   uvm_reg_map map = null,
   ref uvm_reg_addr_t addr[]
)
```

Identifies the external physical address(es) of a memory location. The default value of *offset* shall be `0`.

This computes all of the external physical addresses that need to be accessed to completely read or write the specified location in this memory. The addressed are specified in little endian order. This returns the number of bytes transferred on each access.

If *map* is *null* and the memory is mapped in only one address map, that address map is used. If *map* is *null* and the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message shall be generated.

18.6.5 HDL access

18.6.5.1 write

```
virtual task write(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This initiates a write (using *value* for data) to the memory location that corresponds to this abstraction class instance at the specified *offset*.

The write may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the memory through a physical access is mimicked. For example, read-only memory will remain unchanged.

If front door is specified, and if the memory is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

18.6.5.2 read

```
virtual task read(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads and returns the *value* from the memory location that corresponds to this abstraction class instance at the specified *offset*.

The read may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of reading the memory through a physical access is mimicked.

If front door is specified, and if the memory is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

18.6.5.3 burst_write

```
virtual task burst_write(
    output uvm_status_e status,
    input uvm_reg_data_t value[],
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This initiates a burst-write (using the elements in *value* for data) to the memory locations that correspond to this abstraction class instance beginning at the specified *offset*.

The write may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of writing the memory through a physical access is mimicked. For example, read-only memory will remain unchanged.

If front door is specified, and if the memory is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

18.6.5.4 burst_read

```
virtual task burst_read(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    ref uvm_reg_data_t value[],
    input uvm_door_e path = UVM_DEFAULT_DOOR,
```

```
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This initiates a burst-read that returns the date in the elements in *value* from the memory location that corresponds to this abstraction class instance at the specified *offset*.

The read may be performed using either front-door or back-door operations (as defined by *path*). If back door is specified, the effect of reading the memory through a physical access is mimicked.

If front door is specified, and if the memory is mapped in more than one address map, an address *map* shall be specified. The value of *parent* sequence and *extension* are set into the **uvm_reg_item** (see 19.1.1), which is provided to the **uvm_reg_frontdoor** (see 19.4.2) or **uvm_reg_backdoor** (see 19.5) associated with this request. If the built-in front door is being used and *parent* is not *null*, the bus item returned by the **uvm_reg_adapter** (see 19.2.1) shall be started as a child of *parent*. If the built-in front door is used, the bus item returned by the adapter shall be started with the priority *prior*. Optionally, users may provide additional information for the physical access with the *extension* argument. The *status* output argument reflects the success or failure of the operation.

The filename (*fname*) and line number (*lineno*) arguments are available for an implementation to use for debug purposes only; their value shall have no functional effect on the outcome of this method.

18.6.5.5 poke

```
virtual task poke(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Deposits the *value* in the DUT memory location corresponding to this abstraction class instance at the specified *offset*, as is, using a back-door access.

Uses the HDL path for the design abstraction specified by *kind*. The default value of *lineno* shall be `0`.

18.6.5.6 peek

```
virtual task peek(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the current value from a memory location.

This samples the *value* in the DUT memory location corresponding to this abstraction class instance at the specified *offset* using a back-door access. The memory location value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind*. The default value of *lineno* shall be `0`.

18.6.6 Front door

18.6.6.1 get_frontdoor

```
function uvm_reg_frontdoor get_frontdoor(
   uvm_reg_map map = null
)
```

Returns the user-defined front door for this memory.

If *null*, no user-defined front door has been defined. A user-defined front door is defined by using the **uvm_mem::set_frontdoor** method (see18.6.6.1).

If the memory is mapped in multiple address maps, an address *map* shall be specified.

18.6.6.2 set_frontdoor

```
function void set_frontdoor(
   uvm_reg_frontdoor ftdr,
   uvm_reg_map map = null,
   string fname = "",
   int lineno = 0
)
```

Specifies a user-defined front door for this memory. The default value of *lineno* shall be `0`.

By default, memories are mapped linearly into the address space of the address maps that instantiate them. If memories are accessed using a different mechanism, a user-defined access mechanism shall be defined and associated with the corresponding memory abstraction class.

If the memory is mapped in multiple address maps, an address *map* shall be specified.

18.6.7 Back door

18.6.7.1 get_backdoor

```
function uvm_reg_backdoor get_backdoor(
   bit inherited = 1
)
```

Returns the user-defined back door for this memory.

If *null*, no user-defined back door has been defined. A user-defined back door is defined by using the **uvm_mem::set_backdoor** method (see 18.6.7.2).

If *inherited* is `TRUE`, this returns the back door of the parent block if none have been specified for this memory. The default value of *inherited* shall be `1`, which is `TRUE`.

18.6.7.2 set_backdoor

```
function void set_backdoor(
    uvm_reg_backdoor bkdr,
    string fname = "",
    int lineno = 0
)
```

Specifies a user-defined back door for this memory. The default value of *lineno* shall be `0`.

By default, memories are accessed via the built-in string-based DPI routines if an HDL path has been specified using the **uvm_mem::configure** (see 18.6.3.2) or **uvm_mem::add_hdl_path** (see 18.6.7.4) methods.

If this default mechanism is not suitable (e.g., because the register is not implemented in pure SystemVerilog) a user-defined access mechanism needs to be defined and associated with the corresponding memory abstraction class.

18.6.7.3 clear_hdl_path

```
function void clear_hdl_path (
    string kind = "RTL"
)
```

Deletes any HDL paths. The default value of *kind* shall be `"RTL"`.

This removes any previously specified HDL path to the memory instance for the specified design abstraction.

18.6.7.4 add_hdl_path

```
function void add_hdl_path (
    uvm_hdl_path_slice slices[],
    string kind = "RTL"
)
```

Adds the specified HDL path to the memory instance for the specified design abstraction. The default value of *kind* shall be `"RTL"`.

This method may be called more than once for the same design abstraction if the memory is physically duplicated in the design abstraction.

18.6.7.5 add_hdl_path_slice

```
function void add_hdl_path_slice(
    string name,
    int offset,
    int size,
    bit first = 0,
    string kind = "RTL"
)
```

Appends the specified HDL slice to the HDL path for the specified design abstraction. If *first* is `TRUE`, this starts the specification of a duplicate HDL implementation of the memory. The default value of *first* shall be `0`, which is `FALSE`. The default value of *kind* shall be `"RTL"`.

18.6.7.6 has_hdl_path

```
function bit has_hdl_path (
    string kind = ""
)
```

Checks if a HDL path is specified.

This returns *True* if the memory instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, it uses the default design abstraction specified for the parent block.

18.6.7.7 get_hdl_path

```
function void get_hdl_path (
    ref uvm_hdl_path_concat paths[$],
    input string kind = ""
)
```

Returns the incremental HDL path(s).

This returns the HDL path(s) defined for the specified design abstraction in the memory instance. It returns only the component of the HDL paths that corresponds to the memory, not a full hierarchical path. *paths* shall be a queue.

If no design abstraction is specified, the default design abstraction for the parent block is used.

18.6.7.8 get_hdl_path_kinds

```
function void get_hdl_path_kinds (
    ref string kinds[$]
)
```

Returns any design abstractions for which HDL paths have been defined. *kinds* shall be a queue.

18.6.7.9 get_full_hdl_path

```
function void get_full_hdl_path (
    ref uvm_hdl_path_concat paths[$],
    input string kind = "",
    input string separator = "."
)
```

Returns the full hierarchical HDL path(s).

This returns the full hierarchical HDL path(s) defined for the specified design abstraction in the memory instance. If any of the parent components have more than one path defined for the same design abstraction, there may be more than one path returned (even if only one path was defined for the memory instance).

If no design abstraction is specified, the default design abstraction for each ancestor block is used to retrieve each incremental path.

18.6.7.10 backdoor_read

```
virtual task backdoor_read(
    uvm_reg_item rw
)
```

User-defined back-door read access.

The implementation shall use the UVM HDL back-door access support routines (see 19.6) to perform a read for this register.

18.6.7.11 backdoor_write

```
virtual task backdoor_write(
    uvm_reg_item rw
)
```

User-defined back-door write access.

This overrides the default string-based DPI back-door access write for this memory type.

18.6.8 Coverage

18.6.8.1 build_coverage

```
protected function uvm_reg_cvr_t build_coverage(
    uvm_reg_cvr_t models
)
```

Checks if all of the specified coverage models need to be built.

This checks which of the specified coverage model need to be built in this instance of the memory abstraction class, as specified by calls to **uvm_reg::include_coverage** (see 18.4.7.1).

Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9). This returns the sum of all coverage models to be built in the memory model.

18.6.8.2 add_coverage

```
virtual protected function void add_coverage(
    uvm_reg_cvr_t models
)
```

Specifies that additional coverage models are available.

This adds the specified coverage model to the coverage models available in this class. Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9).

This method shall only be called in the constructor of subsequently derived classes.

18.6.8.3 has_coverage

```
virtual function bit has_coverage(
    uvm_reg_cvr_t models
)
```

Checks if the register has coverage model(s).

This returns *True* if the memory abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e** (see 17.2.2.9).

18.6.8.4 get_coverage

```
virtual function bit get_coverage(
    uvm_reg_cvr_t is_on
)
```

Checks if coverage measurement is on.

This returns *True* if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See 18.6.8.4 for more details.

18.6.8.5 set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(
    uvm_reg_cvr_t is_on
)
```

Turns on coverage measurement.

This turns the collection of functional coverage measurements on or off for this memory. The functional coverage measurement is turned on for every coverage model specified using **uvm_coverage_model_e** coverage model identifiers (see 17.2.2.9). Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. This returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the memory abstraction classes, then enabled during construction. See the **uvm_mem::has_coverage** method (see 18.6.8.3) to identify the available functional coverage models.

18.6.8.6 sample

```
protected virtual function void sample(
    uvm_reg_addr_t offset,
    bit is_read,
    uvm_reg_map map
)
```

This is a functional coverage measurement method.

This method is invoked by the memory abstraction class whenever it is read or written with the specified *data* via the specified address *map*. It is invoked after the read or write operation has completed, but before the mirror has been updated.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

18.6.9 Callbacks

18.6.9.1 pre_write

```
virtual task pre_write(
   uvm_reg_item rw
)
```

Called before memory write.

If the specified data value access *path* or address *map* are modified, the updated data value, access path, or address map is used to perform the memory operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

18.6.9.2 post_write

```
virtual task post_write(
   uvm_reg_item rw
)
```

Called after memory write.

If the specified *status* is modified, the updated status is returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

18.6.9.3 pre_read

```
virtual task pre_read(
   uvm_reg_item rw
)
```

Called before memory read.

If the specified access *path* or address *map* are modified, the updated access path or address map is used to perform the memory operation. If the *status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

18.6.9.4 post_read

```
virtual task post_read(
   uvm_reg_item rw
)
```

Called after memory read.

If the specified read back data or *status* is modified, the updated read back data or status is returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

18.7 uvm_reg_indirect_data

The indirect data access abstraction class.

This models the behavior of a register used to indirectly access a register array, indexed by a second *address* register.

This class should not be instantiated directly. A type-specific class extension should be used to provide a factory-enabled constructor and specify the *n_bits* and *coverage* models.

18.7.1 Class declaration

```
class uvm_reg_indirect_data extends uvm_reg
```

18.7.2 Methods

18.7.2.1 new

```
function new(
   string name = "uvm_reg_indirect",
   int unsigned n_bits,
   int has_cover
)
```

Creates an instance of this class.

This should not be called directly, other than via `super.new`. The value of *n_bits* needs to match the number of bits in the indirect register array.

18.7.2.2 configure

```
function void configure (
   uvm_reg idx,
   uvm_reg reg_a[],
   uvm_reg_block blk_parent,
   uvm_reg_file regfile_parent = null
)
```

Configures the indirect data register.

The *idx* register specifies the index, in the *reg_a* register array, of the register to access. The *idx* needs to be written to first. A read or write operation to this register will subsequently read or write the indexed register in the register array.

The number of bits in each register in the register array shall be equal to *n_bits* of this register.

See also 18.4.2.2.

18.8 uvm_reg_fifo

This special register models a DUT FIFO accessed via write/read, where writes push to the FIFO and reads pop from it.

Back-door access is not enabled, as it is not yet possible to force complete FIFO state, i.e., the write and read indexes used to access the FIFO data.

18.8.1 Class declaration

```
class uvm_reg_fifo extends uvm_reg
```

18.8.2 Common variables

fifo

```
rand uvm_reg_data_t fifo[$]
```

This is the abstract representation of the FIFO. It is constrained to be no larger than the *size* parameter and is public to enable subtypes to add constraints on it and randomize. *fifo* shall be a queue.

18.8.3 Methods

18.8.3.1 new

```
function new(
    string name = "reg_fifo",
    int unsigned size,
    int unsigned n_bits,
    int has_cover
)
```

Creates an instance of a FIFO register having *size* elements of *n_bits* each.

18.8.3.2 set_compare

```
function void set_compare(
    uvm_check_e check = UVM_CHECK
)
```

Specifies the compare policy during a mirror (read) of the DUT FIFO. The DUT read value is checked against its mirror only when both the *check* argument in the **mirror** call (see 18.8.5.8) and the compare policy for the field is `UVM_CHECK` (see 17.2.2.3).

18.8.4 Introspection

18.8.4.1 size

```
function int unsigned size()
```

This is the number of entries currently in the FIFO.

18.8.4.2 capacity

```
function int unsigned capacity()
```

The maximum number of entries, or depth, of the FIFO.

18.8.5 Access

18.8.5.1 get

```
virtual function uvm_reg_data_t get(
    string fname = "",
    int lineno = 0
)
```

Returns the next value from the abstract FIFO, but does not pop it. Used to find the expected value in a **mirror** operation (see 18.8.5.8). See 18.4.4.1 for additional information.

18.8.5.2 set

```
virtual function void set(
    uvm_reg_data_t value,
    string fname = "",
    int lineno = 0
)
```

Pushes the given value to the abstract FIFO. This method may be called several times before an **update** (see 18.8.5.7) as a means of preloading the DUT FIFO. Calls to **set** a full FIFO are ignored. Call **update** to update the DUT FIFO with the appropriate values. See 18.4.4.2 for additional information.

18.8.5.3 write

```
virtual task write(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Pushes the given value to the DUT FIFO. If auto-prediction is enabled, the written value is also pushed to the abstract FIFO before the call returns. If auto-prediction is not enabled [via **uvm_reg_map::set_auto_predict** (see 18.2.5.2)], the value is pushed to abstract FIFO only when the write operation is observed on the target bus. This mode requires using the **uvm_reg_predictor** class (see 19.3). If the write is called by an **update** operation (see 18.8.5.7), the abstract FIFO already contains the written value and is thus not affected by either prediction mode. See 18.4.4.9 for additional information.

18.8.5.4 read

```
virtual task read(
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
```

```
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped. See 18.4.4.10 for additional information.

18.8.5.5 poke

```
virtual task poke(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This deposits the *value* in the DUT register corresponding to this abstraction class instance, as is, using a back-door access. See 18.4.4.11 for additional information.

18.8.5.6 peek

```
virtual task poke(
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This samples the *value* in the DUT register corresponding to this abstraction class instance using a back-door access. See 18.4.4.12 for additional information.

18.8.5.7 update

```
virtual task update(
    output uvm_status_e status,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Pushes (writes) all values preloaded using **set** (see 18.8.5.2) to the DUT. This method needs to be used after **set** and before any blocking statements, otherwise reads/writes to the DUT FIFO may cause the mirror to become out of sync with the DUT. See 18.4.4.13 for additional information.

18.8.5.8 mirror

```
virtual task mirror(
    output uvm_status_e status,
    input uvm_check_e check = UVM_NO_CHECK,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped when the *check* argument is set and comparison is enabled with **set_compare** (see 18.8.3.2). See 18.4.4.14 for additional information.

18.8.5.9 predict

```
virtual function bit predict (
    uvm_reg_data_t value,
    uvm_reg_byte_en_t be = -1,
    uvm_predict_e kind = UVM_PREDICT_DIRECT,
    uvm_door_e path = UVM_FRONTDOOR,
    uvm_reg_map map = null,
    string fname = "",
    int lineno = 0
)
```

Updates the mirrored FIFO. See 18.4.4.15 for additional information.

18.9 uvm_vreg

A virtual register is a collection of fields, overlaid on top of a memory, usually in an array. The semantics and layout of virtual registers comes from an agreement between the software and the hardware, not any physical structures in the DUT. **uvm_reg** is the virtual register abstraction base class.

A virtual register represents a set of fields that are logically implemented in consecutive memory locations. All virtual register accesses eventually turn into memory accesses. A virtual register array may be implemented on top of any memory abstraction class and possibly dynamically resized and/or relocated.

18.9.1 Class declaration

```
class uvm_vreg extends uvm_object
```

18.9.1.1 Methods

18.9.1.1.1 new

```
function new(
    string name,
    int unsigned n_bits
)
```

Creates a new instance and type-specific configuration; this creates an instance of a virtual register abstraction class with the specified *name*.

n_bits specifies the total number of bits in a virtual register. Not all bits need to be mapped to a virtual field. This value is usually a multiple of `8`.

18.9.1.1.2 configure

```
function void configure(
   uvm_reg_block parent,
   uvm_mem mem = null,
   longint unsigned size = 0,
   uvm_reg_addr_t offset = 0,
   int unsigned incr = 0
)
```

This is an instance-specific configuration.

This specifies the *parent* block of this virtual register array. If one of the other parameters is specified, the virtual register is presumed to be dynamic and can be later (re-)implemented using the **uvm_vreg::implement** method (see 18.9.1.1.3).

If *mem* is specified, the virtual register array is presumed to be statically implemented in the memory corresponding to the specified memory abstraction class and the *size*, *offset*, and *incr* also need to be specified. Static virtual register arrays cannot be re-implemented. The default values for *size*, *offset*, and *incr* shall each be `0`.

18.9.1.1.3 implement

```
virtual function bit implement(
   longint unsigned n,
   uvm_mem mem = null,
   uvm_reg_addr_t offset = 0,
   int unsigned incr = 0
)
```

Dynamically implements, resizes, or relocates a virtual register array.

This implements an array of virtual registers of the specified size, in the specified memory and *offset*. If an *offset* increment is specified, each virtual register is implemented at the specified *offset* increment from the previous one. If an *offset* increment of `0` is specified, virtual registers are packed as closely as possible in the memory. The default values for *size* and *offset*, shall be `0`.

If no memory is specified, the virtual register array is in the same memory, at the same base offset using the same *offset* increment as originally implemented. Only the number of virtual registers in the virtual register array is modified.

The initial value of the newly implemented or relocated set of virtual registers is whatever values are currently stored in the memory now implementing them.

This returns `TRUE` if the memory can implement the number of virtual registers at the specified base offset and *offset* increment. Otherwise, it returns `FALSE`.

The memory region used to implement a virtual register array is reserved in the memory allocation manager associated with the memory to prevent it from being allocated for another purpose.

18.9.1.1.4 allocate

```
virtual function uvm_mem_region allocate(
    longint unsigned n,
    uvm_mem_mam mam,
    uvm_mem_mam_policy alloc = null
)
```

Randomly implements, resizes, or relocates a virtual register array.

This implements a virtual register array of the specified size in a randomly allocated region of the appropriate size in the address space managed by the specified memory allocation manager. If a memory allocation policy is specified, it is passed to the **uvm_mem_mam::request_region** method (see 18.12.5.2).

The initial value of the newly implemented or relocated set of virtual registers is whatever values are currently stored in the memory region now implementing them.

This returns a reference to a **uvm_mem_region** memory region descriptor (see 18.12.7) if the memory allocation manager was able to allocate a region that can implement the virtual register array with the specified allocation policy. Otherwise, it returns *null*.

A region implementing a virtual register array cannot be released using the **uvm_mem_mam::release_region** method (see 18.12.5.3); instead, use the **uvm_vreg::release_region** method (see 18.9.1.1.6).

18.9.1.1.5 get_region

```
virtual function uvm_mem_region get_region()
```

Returns the region where the virtual register array is implemented.

This returns a reference to the **uvm_mem_region** memory region descriptor (see 18.12.7) that implements the virtual register array.

It returns *null* if the virtual registers array is not currently implemented. A region implementing a virtual register array cannot be released using the **uvm_mem_mam::release_region** method (see 18.12.5.3); instead, use the **uvm_vreg::release_region** method (see 18.9.1.1.6).

18.9.1.1.6 release_region

```
virtual function void release_region()
```

Dynamically unimplements a virtual register array.

This releases the memory region used to implement a virtual register array and returns it to the pool of available memory that can be allocated by the memory's default allocation manager. The virtual register array is subsequently considered as unimplemented and can no longer be accessed.

Statically implemented virtual registers cannot be released.

18.9.1.2 Introspection

18.9.1.2.1 get_parent

```
virtual function uvm_reg_block get_parent()
```

Returns the parent block.

18.9.1.2.2 get_memory

```
virtual function uvm_mem get_memory()
```

Returns the memory where the virtual register array is implemented.

18.9.1.2.3 get_n_maps

```
virtual function int get_n_maps()
```

Returns the number of address maps mapping this virtual register array.

18.9.1.2.4 is_in_map

```
function bit is_in_map (
   uvm_reg_map map
)
```

Returns `TRUE` if this virtual register array is in the specified address *map*.

18.9.1.2.5 get_maps

```
virtual function void get_maps (
   ref uvm_reg_map maps[$]
)
```

Returns all of the address *maps* where this virtual register array is mapped. *maps* shall be a queue.

18.9.1.2.6 get_rights

```
virtual function string get_rights(
   uvm_reg_map map = null
)
```

Returns the access rights of this virtual register array.

This returns **"RW"**, **"RO"**, or **"WO"**. The access rights of a virtual register array is always **"RW**", unless it is implemented in a shared memory with access restriction in a particular address map.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message shall be generated and `"RW"` is returned.

18.9.1.2.7 get_access

```
virtual function string get_access(
   uvm_reg_map map = null
)
```

Returns the access policy of the virtual register array when written and read via an address map.

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* shall be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions returns "RO".

18.9.1.2.8 get_size

```
virtual function int unsigned get_size()
```

Returns the size of the virtual register array.

18.9.1.2.9 get_n_bytes

```
virtual function int unsigned get_n_bytes()
```

Returns the width, in bytes, of a virtual register.

The width of a virtual register is always a multiple of the width of the memory locations used to implement it. For example, a virtual register containing two 1-byte fields implemented in a memory with 4-byte memory locations is 4-bytes wide.

18.9.1.2.10 get_n_memlocs

```
virtual function int unsigned get_n_memlocs()
```

Returns the number of memory locations used by a single virtual register.

18.9.1.2.11 get_incr

```
virtual function int unsigned get_incr()
```

Returns the number of memory locations between two individual virtual registers in the same array.

18.9.1.2.12 get_fields

```
virtual function void get_fields(
   ref uvm_vreg_field fields[$]
)
```

Returns the virtual fields in this virtual register.

Fills the specified array with the abstraction class for all of the virtual fields contained in this virtual register. Fields are ordered from least significant position to most significant position within the register. *fields* shall be a queue.

18.9.1.2.13 get_field_by_name

```
virtual function uvm_vreg_field get_field_by_name(
   string name
)
```

Returns the named virtual field in this virtual register.

This finds a virtual field with the specified name in this virtual register and returns its abstraction class. If no fields are found, it returns *null*.

18.9.1.2.14 get_offset_in_memory

```
virtual function uvm_reg_addr_t get_offset_in_memory(
   longint unsigned idx
)
```

Returns the offset of a virtual register.

This returns the base offset of the specified virtual register, in the overall address space of the memory that implements the virtual register array.

18.9.1.2.15 get_address

```
virtual function uvm_reg_addr_t get_address(
   longint unsigned idx,
   uvm_reg_map map = null
)
```

Returns the base external physical address of the specified virtual register if accessed through the specified address *map*.

If no address map is specified and the memory implementing the virtual register array is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, a warning message shall be issued.

18.9.1.3 HDL access

18.9.1.3.1 write

```
virtual task write(
   input longint unsigned idx,
   output uvm_status_e status,
   input uvm_reg_data_t value,
   input uvm_door_e path = UVM_DEFAULT_DOOR,
   input uvm_reg_map map = null,
   input uvm_sequence_base parent = null,
   input uvm_object extension = null,
   input string fname = "",
   input int lineno = 0
)
```

This writes the *value* in the DUT memory location(s) (specified by *idx*) that implements the virtual register array corresponding to this abstraction class instance using the specified access *path*. See 18.6.5.1 for additional information.

18.9.1.3.2 read

```
virtual task read(
    input longint unsigned idx,
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This reads from the DUT memory location(s) (specified by *idx*) that implements the virtual register array corresponding to this abstraction class instance using the specified access *path* and returns the read back *value*. See 18.6.5.2 for additional information.

18.9.1.3.3 poke

```
virtual task poke(
    input longint unsigned idx,
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Deposits the specified value in a virtual register; this deposits the *value* in the DUT memory location(s) (specified by *idx*) that implements the virtual register array corresponding to this abstraction class instance using the memory back-door access. See 18.6.5.5 for additional information.

18.9.1.3.4 peek

```
virtual task peek(
    input longint unsigned idx,
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Samples the DUT memory location(s) (specified by *idx*) that implements the virtual register array corresponding to this abstraction class instance using the memory back-door access and returns the sampled *value*. See 18.6.5.6 for additional information.

18.9.1.3.5 reset

```
function void reset(
    string kind = "HARD"
)
```

Resets the semaphore that prevents concurrent access to the virtual register. This semaphore shall be explicitly reset if a thread accessing this virtual register array was killed before the access was completed. The default value of *kind* shall be "HARD".

18.9.1.4 Callbacks

18.9.1.4.1 pre_write

```
virtual task pre_write(
    longint unsigned idx,
    ref uvm_reg_data_t wdat,
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before virtual register write.

If the specified data value, access *path*, or address *map* are modified, the updated data value, access path, or address map are used to perform the virtual register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks.

18.9.1.4.2 post_write

```
virtual task post_write(
    longint unsigned idx,
    uvm_reg_data_t wdat,
    uvm_door_e path,
    uvm_reg_map map,
    ref uvm_status_e status
)
```

Called after virtual register write.

If the specified *status* is modified, the updated status is returned by the virtual register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks.

18.9.1.4.3 pre_read

```
virtual task pre_read(
    longint unsigned idx,
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before virtual register read.

If the specified access *path* or address *map* are modified, the updated access path or address map are used to perform the virtual register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks.

18.9.1.4.4 post_read

```
virtual task post_read(
    longint unsigned idx,
    ref uvm_reg_data_t rdat,
    input uvm_door_e path,
    input uvm_reg_map map,
    ref uvm_status_e status
)
```

Called after virtual register read.

If the specified read back data or *status* is modified, the updated read back data or status is returned by the virtual register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks.

18.9.2 uvm_vreg_cbs

The pre/post read/write callback facade class.

18.9.2.1 Class declaration

```
class uvm_vreg_cbs extends uvm_callback
```

18.9.2.2 Callbacks

18.9.2.2.1 pre_write

```
virtual task pre_write(
    uvm_vreg rg,
    longint unsigned idx,
    ref uvm_reg_data_t wdat,
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before a write operation.

The registered callback methods are invoked after the invocation of the **uvm_vreg::pre_write** method (see 18.9.1.4.1). All virtual register callbacks are executed after the corresponding virtual field callbacks. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks.

If the specified *wdat* value, access *path*, or address *map* are modified, the updated value, access path, or address map are used to perform the virtual register operation.

18.9.2.2.2 post_write

```
virtual task post_write(
   uvm_vreg rg,
   longint unsigned idx,
   uvm_reg_data_t wdat,
   uvm_door_e path,
   uvm_reg_map map,
   ref uvm_status_e status
)
```

Called after register write.

The registered callback methods are invoked before the invocation of the **uvm_vreg::post_write** method (see 18.9.1.4.2). All virtual register callbacks are executed before the corresponding virtual field callbacks. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks.

If the specified *status* is modified, the updated status is returned by the virtual register operation.

18.9.2.2.3 pre_read

```
virtual task pre_read(
   uvm_vreg rg,
   longint unsigned idx,
   ref uvm_door_e path,
   ref uvm_reg_map map
)
```

Called before register read.

The registered callback methods are invoked after the invocation of the **uvm_vreg::pre_read** method (see 18.9.1.4.3). All virtual register callbacks are executed after the corresponding virtual field callbacks. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks.

If the specified access *path* or address *map* are modified, the updated access path or address map are used to perform the virtual register operation.

18.9.2.2.4 post_read

```
virtual task post_read(
   uvm_vreg rg,
   longint unsigned idx,
   ref uvm_reg_data_t rdat,
   input uvm_door_e path,
   input uvm_reg_map map,
   ref uvm_status_e status
)
```

Called after register read.

The registered callback methods are invoked before the invocation of the **uvm_vreg::post_read** method (see 18.9.1.4.4). All virtual register callbacks are executed before the corresponding virtual field callbacks. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks.

If the specified read back *rdat* value or *status* is modified, the updated read back value or status is returned by the virtual register operation.

18.10 uvm_vreg_field

This subclause defines the virtual field and callback classes.

A virtual field is a set of contiguous bits in one or more memory locations. The semantics and layout of virtual fields comes from an agreement between the software and the hardware, not any physical structures in the DUT. **uvm_reg_field** is the virtual field abstraction base class.

18.10.1 Class declaration

```
class uvm_vreg_field extends uvm_object
```

18.10.2 Methods

18.10.2.1 new

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

Creates a new virtual field instance.

This method should not be used directly. The **uvm_vreg_field::type_id::create** method should be used instead.

18.10.2.2 configure

```
function void configure(
   uvm_vreg parent,
   int unsigned size,
   int unsigned lsb_pos
)
```

This is an instance-specific configuration.

This specifies the *parent* virtual register of this virtual field, its *size* in bits, and the position of its LSB within the virtual register relative to the LSB of the virtual register.

18.10.3 Introspection

18.10.3.1 get_parent

```
virtual function uvm_vreg get_parent()
```

Returns the parent of the virtual field.

18.10.3.2 get_lsb_pos_in_register

```
virtual function int unsigned get_lsb_pos_in_register()
```

Returns the position of the virtual field; Or returns the index of the LSB of the virtual field in the virtual register that instantiates it. An offset of `0` indicates a field that is aligned with the LSB of the register.

18.10.3.3 get_n_bits

```
virtual function int unsigned get_n_bits()
```

Returns the width, in bits, of the virtual field.

18.10.3.4 get_access

```
virtual function string get_access(
   uvm_reg_map map = null
)
```

Returns the access policy of the virtual field register when written and read via an address map.

If the memory implementing the virtual field is mapped in more than one address map, an address *map* shall be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions returns `"RO"`.

18.10.4 HDL access

18.10.4.1 write

```
virtual task write(
   input longint unsigned idx,
   output uvm_status_e status,
   input uvm_reg_data_t value,
   input uvm_door_e path = UVM_DEFAULT_DOOR,
   input uvm_reg_map map = null,
   input uvm_sequence_base parent = null,
   input uvm_object extension = null,
   input string fname = "",
   input int lineno = 0
)
```

This writes the *value* in the DUT memory location(s) (specified by *idx*) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*. See 18.6.5.1 for additional information.

18.10.4.2 read

```
virtual task read(
   input longint unsigned idx,
   output uvm_status_e status,
   output uvm_reg_data_t value,
   input uvm_door_e path = UVM_DEFAULT_DOOR,
   input uvm_reg_map map = null,
   input uvm_sequence_base parent = null,
   input uvm_object extension = null,
```

```
    input string fname = "",
    input int lineno = 0
)
```

This reads from the DUT memory location(s) (specified by *idx*) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path* and returns the read back *value*. See 18.6.5.2 for additional information.

18.10.4.3 poke

```
virtual task poke(
    input longint unsigned idx,
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This deposits the *value* in the DUT memory location(s) (specified by *idx*) that implements the virtual field corresponding to this abstraction class instance using the specified access *path*. See 18.6.5.5 for additional information.

18.10.4.4 peek

```
virtual task peek(
    input longint unsigned idx,
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

This samples from the DUT memory location(s) (specified by *idx*) that implements the virtual field corresponding to this abstraction class instance using the specified access *path* and returns the read back *value*. See 18.6.5.6 for additional information.

18.10.5 Callbacks

18.10.5.1 pre_write

```
virtual task pre_write(
    longint unsigned idx,
    ref uvm_reg_data_t wdat,
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before virtual field write.

If the specified data value, access *path*, or address *map* are modified, the updated data value, access path, or address map are used to perform the virtual register operation.

The virtual field callback methods are invoked before the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks.

18.10.5.2 post_write

```
virtual task post_write(
    longint unsigned idx,
    uvm_reg_data_t wdat,
    uvm_door_e path,
    uvm_reg_map map,
    ref uvm_status_e status
)
```

Called after virtual field write.

If the specified *status* is modified, the updated status is returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks.

18.10.5.3 pre_read

```
virtual task pre_read(
    longint unsigned idx,
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before virtual field read.

If the specified access *path* or address *map* are modified, the updated access path or address map are used to perform the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks.

18.10.5.4 post_read

```
virtual task post_read(
    longint unsigned idx,
    ref uvm_reg_data_t rdat,
    uvm_door_e path,
    uvm_reg_map map,
    ref uvm_status_e status
)
```

Called after virtual field read.

If the specified read back data *rdat* or *status* is modified, the updated read back data or status is returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks.

18.10.6 uvm_vreg_field_cbs

The pre/post read/write callback facade class.

18.10.6.1 Class declaration

```
virtual class uvm_vreg_field_cbs extends uvm_callback
```

18.10.6.2 Callbacks

18.10.6.2.1 pre_write

```
virtual task pre_write(
   uvm_vreg_field field,
   longint unsigned idx,
   ref uvm_reg_data_t wdat,
   ref uvm_door_e path,
   ref uvm_reg_map map
)
```

Called before a write operation.

The registered callback methods are invoked before the invocation of the virtual register pre-write callbacks and after the invocation of the **uvm_vreg_field::pre_write** method (see 18.10.5.1).

If the specified *wdat* value, access *path*, or address *map* are modified, the updated value, access path, or address map are used to perform the register operation.

18.10.6.2.2 post_write

```
virtual task post_write(
   uvm_vreg_field field,
   longint unsigned idx,
   uvm_reg_data_t wdat,
   uvm_door_e path,
   uvm_reg_map map,
   ref uvm_status_e status
)
```

Called after a write operation.

The registered callback methods are invoked after the invocation of the virtual register post-write callbacks and before the invocation of the **uvm_vreg_field::post_write** method (see 18.10.5.2).

If the specified *status* is modified, the updated status is returned by the operation.

18.10.6.2.3 pre_read

```
virtual task pre_read(
   uvm_vreg_field field,
   longint unsigned idx,
```

```
    ref uvm_door_e path,
    ref uvm_reg_map map
)
```

Called before a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register pre-read callbacks and after the invocation of the **uvm_vreg_field::pre_read** method (see 18.10.5.3).

If the specified access *path* or address *map* are modified, the updated access path or address map are used to perform the register operation.

18.10.6.2.4 post_read

```
virtual task post_read(
    uvm_vreg_field field,
    longint unsigned idx,
    ref uvm_reg_data_t rdat,
    uvm_door_e path,
    uvm_reg_map map,
    ref uvm_status_e status
)
```

Called after a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register post-read callbacks and before the invocation of the **uvm_vreg_field::post_read** method (see 18.10.5.4).

If the specified read back data *rdat* or *status* is modified, the updated read back data or status is returned by the operation.

18.11 uvm_reg_cbs

This subclause defines the base class used for all register callback extensions. It also includes predefined callback extensions for use on read-only and write-only registers. **uvm_reg_cbs** is the facade class for field, register, memory and back-door access callback methods.

18.11.1 Class declaration

```
class uvm_reg_cbs extends uvm_callback
```

18.11.2 Methods

18.11.2.1 new

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

This creates an instance of the register call back class with the specified *name*.

18.11.2.2 pre_write

```
virtual task pre_write(
    uvm_reg_item rw
)
```

Called before a write operation.

All registered **pre_write** callback methods are invoked after the invocation of the **pre_write** method of associated object [**uvm_reg_backdoor** (see 19.5), **uvm_reg** (see 18.4), **uvm_reg_field** (see 18.5), or **uvm_mem** (see 18.6)].

a) *Back door*—**uvm_reg_backdoor::pre_write** (see 19.5.2.13) and **uvm_reg_cbs::pre_write** callbacks for back door.

b) *Register*—**uvm_reg::pre_write** (see 18.4.8.1) and **uvm_reg_cbs::pre_write** callbacks for reg; then for each field: **uvm_reg_field::pre_write** (see 18.5.6.1) and **uvm_reg_cbs::pre_write** callbacks for field.

   When the element being written is a **uvm_reg**, all **pre_write** callback methods are invoked before the contained **uvm_reg_field**s.

c) *RegField*—**uvm_reg_field::pre_write** (see 18.5.6.1) and **uvm_reg_cbs::pre_write** callbacks for field.

d) *Memory*—**uvm_mem::pre_write** (see 18.6.9.1) and **uvm_reg_cbs::pre_write** callbacks for mem.

The *rw* argument holds information about the operation.

— Modifying the *value* modifies the actual value written.
— For memories, modifying the offset modifies the *offset* used in the operation.
— For non back-door operations, modifying the access *path* or address *map* modifies the actual path or map used in the operation.

If the *rw.status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted. See 19.1.1 for more details on *rw*.

18.11.2.3 post_write

```
virtual task post_write(
    uvm_reg_item rw
)
```

Called after a write operation.

All registered **post_write** callback methods are invoked before the invocation of the **post_write** method of associated object [**uvm_reg_backdoor** (see 19.5), **uvm_reg** (see 18.4), **uvm_reg_field** (see 18.5), or **uvm_mem** (see 18.6)].

a) *Back door*—**uvm_reg_cbs::post_write** callbacks for back door, **uvm_reg_backdoor::post_write** (see 19.5.2.14).

b) *Register*—**uvm_reg_cbs::post_write** callbacks for reg, **uvm_reg::post_write** (see 18.4.8.2); then for each field: **uvm_reg_cbs::post_write** callbacks for field, **uvm_reg_field::post_write** (see 18.5.6.2).

   When the element being written is a **uvm_reg**, all **post_write** callback methods are invoked before the contained **uvm_reg_field**s.

c) *RegField*— **uvm_reg_cbs::post_write** callbacks for field, **uvm_reg_field::post_write** (see 18.5.6.2).

d) *Memory*—**uvm_reg_cbs::post_write** callbacks for mem, **uvm_mem::post_write** (see 18.6.9.2).

The *rw* argument holds information about the operation.

— Modifying the *status* member modifies the returned status.

— Modifying the *value* or *offset* members has no effect, as the operation has already completed.

See 19.1.1 for more details on *rw*.

18.11.2.4 pre_read

```
virtual task pre_read(
    uvm_reg_item rw
)
```

Called before a read operation.

All registered **pre_read** callback methods are invoked after the invocation of the **pre_read** method of associated object [**uvm_reg_backdoor** (see 19.5), **uvm_reg** (see 18.4), **uvm_reg_field** (see 18.5), or **uvm_mem** (see 18.6)].

a) *Back door*—**uvm_reg_backdoor::pre_read** (see 19.5.2.11) and **uvm_reg_cbs::pre_read** callbacks for back door.

b) *Register*—**uvm_reg::pre_read** (see 18.4.8.3) and **uvm_reg_cbs::pre_read** callbacks for reg; then for each field: **uvm_reg_field::pre_read** (see 18.5.6.3) and **uvm_reg_cbs::pre_read** callbacks for field.

   When the element being written is a **uvm_reg**, all **pre_read** callback methods are invoked before the contained **uvm_reg_field**s.

c) *RegField*—**uvm_reg_field::pre_read** (see 18.5.6.3) and **uvm_reg_cbs::pre_read** callbacks for field.

d) *Memory*—**uvm_mem::pre_read** (see 18.6.9.3) and **uvm_reg_cbs::pre_read** callbacks for mem.

The *rw* argument holds information about the operation.

— The *value* member of *rw* is not used and has no effect if modified.

— For memories, modifying the offset modifies the *offset* used in the operation.

— For non back-door operations, modifying the access *path* or address *map* modifies the actual path or map used in the operation.

If the *rw.status* is modified to anything other than `UVM_IS_OK` (see 17.2.2.1), the operation is aborted. See 19.1.1 for more details on *rw*.

18.11.2.5 post_read

```
virtual task post_read(
    uvm_reg_item rw
)
```

Called after a read operation.

All registered **post_read** callback methods are invoked before the invocation of the **post_read** method of associated object [**uvm_reg_backdoor** (see 19.5), **uvm_reg** (see 18.4), **uvm_reg_field** (see 18.5), or **uvm_mem** (see 18.6)].

a) *Back door*—**uvm_reg_cbs::post_read** callbacks for back door, **uvm_reg_backdoor::post_read** (see 19.5.2.12).

b) *Register*—**uvm_reg_cbs::post_read** callbacks for reg, **uvm_reg::post_read** (see 18.4.8.4); then for each field: **uvm_reg_cbs::post_read** callbacks for field, **uvm_reg_field::post_read** (see 18.5.6.4).

   When the element being written is a **uvm_reg**, all **post_read** callback methods are invoked before the contained **uvm_reg_field**s.

c) *RegField*— **uvm_reg_cbs::post_read** callbacks for field, **uvm_reg_field::post_read** (see 18.5.6.4).

d) *Memory*—**uvm_reg_cbs::post_read** callbacks for mem, **uvm_mem::post_read** (see 18.6.9.4).

The *rw* argument holds information about the operation.

— Modifying the read back *value* or *status* modifies the actual returned value or status.
— Modifying the *value* or *offset* members has no effect, as the operation has already completed.

See 19.1.1 for more details on *rw*.

18.11.2.6 post_predict

```
virtual function void post_predict(
    input uvm_reg_field fld,
    input uvm_reg_data_t previous,
    inout uvm_reg_data_t value,
    input uvm_predict_e kind,
    input uvm_door_e path,
    input uvm_reg_map map
)
```

Called by the **uvm_reg_field::predict** method (see 18.5.5.17) after a successful **UVM_PREDICT_READ** or **UVM_PREDICT_WRITE** prediction (see 17.2.2.8).

previous is the previous value in the mirror and *value* is the latest predicted value. Any change to *value* modifies the predicted mirror value.

18.11.2.7 encode

```
virtual function void encode(
    ref uvm_reg_data_t data[]
)
```

This is a data encoder.

The registered callback methods are invoked in order of registration after all the **pre_write** methods (see 18.11.2.2) have been called. The encoded data is passed through each invocation in sequence. This allows the **pre_write** methods to deal with clear-text data.

By default, the data is not modified.

18.11.2.8 decode

```
virtual function void decode(
    ref uvm_reg_data_t data[]
)
```

This is a data decoder.

The registered callback methods are invoked in reverse order of registration before all the **post_read** methods (see 18.11.2.5) are called. The decoded data is passed through each invocation in sequence. This allows the **post_read** methods to deal with clear-text data.

The reversal of the invocation order is to allow the decoding of the data to be performed in the opposite order of the encoding, with both operations specified in the same callback extension.

By default, the data is not modified.

18.11.3 Types

For a description of the convenience callback types for **uvm_reg_cbs**, see D.4.6.

18.11.4 uvm_reg_read_only_cbs

This is a predefined register callback class for read-only registers that generates an error if a **write** operation is attempted.

18.11.4.1 Class declaration

```
virtual class uvm_reg_read_only_cbs extends uvm_reg_cbs
```

18.11.4.2 Methods

18.11.4.2.1 pre_write

```
virtual task pre_write(
   uvm_reg_item rw
)
```

Generates an error message and sets status to **UVM_NOT_OK** (see17.2.2.1).

18.11.4.2.2 add

```
static function void add(
   uvm_reg rg
)
```

Adds this callback to the specified register and its contained fields.

18.11.4.2.3 remove

```
static function void remove(
   uvm_reg rg
)
```

Removes this callback from the specified register and its contained fields.

18.11.5 uvm_reg_write_only_cbs

This is a predefined register callback method for write-only registers that generates an error if a **read** operation is attempted.

18.11.5.1 Class declaration

```
virtual class uvm_reg_write_only_cbs extends uvm_reg_cbs
```

18.11.5.2 Methods

18.11.5.2.1 pre_read

```
virtual task pre_read(
   uvm_reg_item rw
)
```

Generates an error message and sets status to **UVM_NOT_OK** (see17.2.2.1).

18.11.5.2.2 add

```
static function void add(
   uvm_reg rg
)
```

Adds this callback to the specified register and its contained fields.

18.11.5.2.3 remove

```
static function void remove(
   uvm_reg rg
)
```

Removes this callback from the specified register and its contained fields.

18.12 uvm_mem_mam

The memory allocation management utility class is similar to C's `malloc and free` functions. A single instance of this class is used to manage the exclusive allocation of consecutive memory locations called *regions*. The regions can subsequently be accessed like little memories of their own, without knowing in which memory or offset they are actually located.

The memory allocation manager should be used by any application-level process that requires reserved space in the memory, such as DMA buffers.

A region shall remain reserved until it is explicitly released.

18.12.1 Class declaration

```
class uvm_mem_mam
```

18.12.2 Types

18.12.2.1 alloc_mode_e

```
typedef enum {GREEDY, THRIFTY} alloc_mode_e
```

The memory allocation mode.

Specifies how to allocate a memory region.

a) *GREEDY*—Consume new, previously unallocated memory.
b) *THRIFTY*—Reuse previously released memory as much as possible.

18.12.2.2 locality_e

```
typedef enum {BROAD, NEARBY} locality_e
```

Location of memory regions.

Specifies where to locate new memory regions.

a) *BROAD*—Locate new regions randomly throughout the address space.
b) *NEARBY*—Locate new regions adjacent to existing regions.

18.12.3 Variables

default_alloc

```
uvm_mem_mam_policy default_alloc
```

This is the region allocation policy.

This object is repeatedly randomized when allocating new regions.

18.12.4 Methods

18.12.4.1 new

```
function new(
    string name,
    uvm_mem_mam_cfg cfg,
    uvm_mem mem = null
)
```

Creates a new manager instance of a memory allocation manager with the specified *name* and configuration. This instance manages all memory region allocation within the address range specified in the configuration descriptor.

If a reference to a memory abstraction class is provided, the memory locations within the regions can be accessed through the region descriptor, using the **uvm_mem_region::read** (see 18.12.7.2.9) and **uvm_mem_region::write** (see 18.12.7.2.8) methods. See 18.12.9 for more details on *cfg*.

18.12.4.2 reconfigure

```
function uvm_mem_mam_cfg reconfigure(
    uvm_mem_mam_cfg cfg = null
)
```

Reconfigures the manager.

This modifies the maximum and minimum addresses of the address space managed by the allocation manager, allocation mode, or locality. The number of bytes per memory location cannot be modified once

an allocation manager has been constructed. All currently allocated regions must fall within the new address space.

This returns the previous configuration. If no new configuration is specified, this simply returns the current configuration.

18.12.5 Memory management

18.12.5.1 reserve_region

```
function uvm_mem_region reserve_region(
   bit [63:0] start_offset,
   int unsigned n_bytes,
   string fname = "",
   int lineno = 0
)
```

Reserves a specific memory region of the specified number of bytes starting at the specified offset. A descriptor of the reserved region is returned. If the specified region cannot be reserved, *null* is returned. The default value of *lineno* shall be `0`.

It may not be possible to reserve a region because it overlaps with an already allocated region or it lies outside the address range managed by the memory manager.

Regions can be reserved to create “holes” in the managed address space.

18.12.5.2 request_region

```
function uvm_mem_region request_region(
   int unsigned n_bytes,
   uvm_mem_mam_policy alloc = null,
   string fname = "",
   int lineno = 0
)
```

Requests and reserves a memory region of the specified number of bytes starting at a random location. If a policy is specified, it is randomized to determine the start offset of the region. If no policy is specified, the policy found in the **uvm_mem_mam::default_alloc** class property (see 18.12.4.2) is randomized. The default value of *lineno* shall be `0`.

A descriptor of the allocated region is returned. If no region can be allocated, *null* is returned.

It may not be possible to allocate a region because there is no area in the memory with enough consecutive locations to meet the size requirements or there is another contradiction when randomizing the policy.

If the memory allocation is configured to `THRIFTY` or `NEARBY`, a suitable region is first sought procedurally.

18.12.5.3 release_region

```
function void release_region(
   uvm_mem_region region
)
```

Releases a previously allocated memory region. An error shall be generated if the specified region has not been previously allocated or is no longer allocated. See 18.12.7 for more details on *region*.

18.12.5.4 release_all_regions

```
function void release_all_regions()
```

Forcibly releases all allocated memory regions.

18.12.6 Introspection

18.12.6.1 convert2string

```
function string convert2string()
```

Creates a human-readable description of the state of the memory manager and the currently allocated regions.

18.12.6.2 for_each

```
function uvm_mem_region for_each(
   bit reset = 0
)
```

This iterates over all currently allocated regions.

If *reset* is `TRUE`, this resets the iterator and returns the first allocated region. It returns *null* when there are no additional allocated regions to iterate. The default value of *reset* shall be `0`, which is `FALSE`.

18.12.6.3 get_memory

```
function uvm_mem get_memory()
```

Returns the managed memory implementation.

This returns the reference to the memory abstraction class for the memory implementing the locations managed by this instance of the allocation manager. It returns *null* if no memory abstraction class was specified at construction time.

18.12.7 uvm_mem_region

Each instance of this class describes an allocated memory region. Instances of this class are created only by the memory manager and returned by the **uvm_mem_mam::reserve_region** (see 18.12.5.1) and **uvm_mem_mam::request_region** (see 18.12.5.2) methods.

18.12.7.1 Class declaration

```
class uvm_mem_region
```

18.12.7.2 Methods

18.12.7.2.1 get_start_offset

```
function bit [63:0] get_start_offset()
```

Returns the start offset of the region.

This returns the address offset, within the memory, where this memory region starts.

18.12.7.2.2 get_end_offset

```
function bit [63:0] get_end_offset()
```

Returns the end offset of the region.

This returns the address offset, within the memory, where this memory region ends.

18.12.7.2.3 get_len

```
function int unsigned get_len()
```

The size of the memory region.

This returns the number of consecutive memory locations (not necessarily bytes) in the allocated region.

18.12.7.2.4 get_n_bytes

```
function int unsigned get_n_bytes()
```

The number of bytes in the region.

This returns the number of consecutive bytes in the allocated region. If the managed memory contains more than one byte per address, the number of bytes in an allocated region may be greater than the number of requested or reserved bytes.

18.12.7.2.5 release_region

```
function void release_region()
```

Releases this region.

18.12.7.2.6 get_memory

```
function uvm_mem get_memory()
```

Returns the memory where the region resides.

This returns a reference to the memory abstraction class for the memory implementing this allocated memory region. It returns *null* if no memory abstraction class was specified for the allocation manager that allocated this region.

18.12.7.2.7 get_virtual_registers

```
function uvm_vreg get_virtual_registers()
```

Returns the virtual register array in this region.

This returns a reference to the virtual register array abstraction class implemented in this region. It returns *null* if the memory region is not known to implement virtual registers.

18.12.7.2.8 write

```
task write(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes to the memory location that corresponds to the specified *offset* within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.1 for additional information.

18.12.7.2.9 read

```
task read(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads from the memory location that corresponds to the specified *offset* within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.2 for additional information.

18.12.7.2.10 burst_write

```
task burst_write(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value[],
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes to the memory locations that corresponds to the specified burst within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.3 for additional information.

18.12.7.2.11 burst_read

```
task burst_read(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value[],
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads from the memory locations that corresponds to the specified burst within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.4 for additional information.

18.12.7.2.12 poke

```
task poke(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Deposits the specified value in the memory location that corresponds to the specified *offset* within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.5 for additional information.

18.12.7.2.13 peek

```
task peek(
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input uvm_sequence_base parent = null,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Samples the memory location that corresponds to the specified *offset* within this region. This requires that the memory abstraction class be associated with the memory allocation manager that allocated this region. See 18.6.5.6 for additional information.

18.12.8 uvm_mem_mam_policy

An instance of this class is randomized to determine the starting offset of a randomly allocated memory region. This class can be extended to provide additional constraints on the starting offset, such as word

alignment or location of the region within a memory page. If a procedural region allocation policy is required, it can be implemented in the `pre_randomize`/`post_randomize` method.

18.12.8.1 Class declaration

```
class uvm_mem_mam_policy
```

18.12.8.2 Variables

18.12.8.2.1 len

```
int unsigned len
```

The number of addresses required.

18.12.8.2.2 start_offset

```
rand bit [63:0] start_offset
```

The starting offset of the region.

18.12.8.2.3 min_offset

```
bit [63:0] min_offset
```

The minimum address offset in the managed address space.

18.12.8.2.4 max_offset

```
bit [63:0] max_offset
```

The maximum address offset in the managed address space.

18.12.8.2.5 in_use

```
uvm_mem_region in_use[$]
```

The regions already allocated in the managed address space. *in_use* shall be a queue.

18.12.9 uvm_mem_mam_cfg

This specifies the memory managed by an instance of a **uvm_mem_mam** memory allocation manager class (see 18.12).

18.12.9.1 Class declaration

```
class uvm_mem_mam_cfg
```

18.12.9.2 Variables

18.12.9.2.1 n_bytes

```
rand int unsigned n_bytes
```

The number of bytes in each memory location.

18.12.9.2.2 start_offset

```
rand bit [63:0] start_offset
```

The starting address of managed space.

18.12.9.2.3 end_offset

```
rand bit [63:0] end_offset
```

The last address of managed space.

18.12.9.2.4 mode

```
rand uvm_mem_mam::alloc_mode_e mode
```

The region allocation mode.

18.12.9.2.5 locality

```
rand uvm_mem_mam::locality_e locality
```

The region location mode.

19. Register layer interaction with RTL design

19.1 Generic register operation descriptors

This subclause defines the abstract register transaction item. It also defines a descriptor for a physical bus operation that is used by **uvm_reg_adapter** subtypes (see 19.2.1) to convert from a protocol-specific address/data/rw operation to a bus-independent, canonical *rw* operation.

19.1.1 uvm_reg_item

Defines an abstract register transaction item. No bus-specific information is present, although a handle to a **uvm_reg_map** (see 18.2) is provided in case a user wishes to implement a custom address translation algorithm.

19.1.1.1 Class declaration

```
class uvm_reg_item extends uvm_sequence_item
```

19.1.1.2 Methods for item use

19.1.1.2.1 Element kind

```
virtual function void set_element_kind (uvm_elem_kind_e element_kind)
virtual function uvm_elem_kind_e get_element_kind()
```

Kind of element being accessed: **UVM_REG**, **UVM_MEM**, or **UVM_FIELD** (see 18.2.5). **get_element_kind** shall return the most recent *element_kind* assigned via **set_element_kind**. The value returned by **get_element_kind** prior to any **set_element_kind** call is undefined.

19.1.1.2.2 Element

```
virtual function void set_element (uvm_object element)
virtual function uvm_object get_element()
```

A handle to the `RegModel` model element associated with this transaction. Use the element king (see 19.1.1.2.1) to determine the type to cast to: **uvm_reg** (see 18.4), **uvm_mem** (see 18.6), or **uvm_reg_field** (see 18.5). **get_element** shall return the most recent *element* assigned via **set_element**. The value returned by **get_element** prior to any **set_element** call shall be *null*.

19.1.1.2.3 Kind

```
virtual function void set_kind (uvm_access_e kind)
virtual function uvm_access_e get_kind()
```

Kind of access: **UVM_READ**, **UVM_WRITE**, **UVM_BURST_READ**, or **UVM_BURST_WRITE** (see 17.2.2.6). **get_kind** shall return the most recent *kind* assigned via **set_kind**. The value returned by **get_kind** prior to any **set_kind** call is undefined.

19.1.1.2.4 Data value

```
virtual function void set_value (uvm_reg_data_t value, int unsigned idx=0)
virtual function uvm_reg_data_t get_value (int unsigned idx=0)
```

```
virtual function void set_value_size (int unsigned sz)
virtual function int unsigned get_value_size()

virtual function void set_value_array (const ref uvm_reg_data_t value[])
virtual function void get_value_array (ref uvm_reg_data_t value[])
```

The value to write to, or after completion, the value read from the DUT.

get_value_size shall return the size of the item's internal value array. By default, the item stores a single value element (i.e., **get_value_size** returns `1`). Burst operations may change the size of the item's internal value array using **set_value_size**. Data values at indexes less than *sz* are unaffected by **set_value_size**. Any new data values created by passing a *sz* greater than the current **get_value_size** shall be initialized to `0`.

A warning message shall be issued if the *idx* of **set_value** is equal to or greater than the current **get_value_size**, and the request shall be ignored. If **get_value** is called with an *idx* equal to or greater than **get_value_size**, then a warning message shall be issued and `0` shall be returned.

Additionally, methods are provided for efficiently getting or setting item value(s) using dynamic arrays. **set_value_array** shall be functionally equivalent to calling **set_value_size**, followed by **set_value** for each value in the *value* array. **get_value_array** shall be functionally equivalent to calling `new[item.get_value_size]` on *value* and then passing in the values returned by **get_value**.

19.1.1.2.5 Offset

```
virtual function void set_offset (uvm_reg_addr_t offset)
virtual function uvm_reg_addr_t get_offset()
```

For memory accesses, the offset address. For bursts, the starting offset address. **get_offset** shall return the most recent *offset* assigned via **set_offset**. The value returned by **get_offset** prior to any **set_offset** call is `0`.

19.1.1.2.6 Status

```
virtual function void set_status (uvm_status_e status)
virtual function uvm_status_e get_status()
```

The result of the transaction: **UVM_IS_OK**, **UVM_HAS_X**, or **UVM_NOT_OK** (see 17.2.2.1). **get_status** shall return the most recent *status* assigned via **set_status**. The value returned by **get_status** prior to any **set_status** call is undefined.

19.1.1.2.7 Local map

```
virtual function void set_local_map (uvm_reg_map map)
virtual function uvm_reg_map get_local_map()
```

The local map used to obtain addresses. Users may customize address-translation using this map. Access to the sequencer and bus adapter can be obtained by retrieving this map's root map, then calling **uvm_reg_map::get_sequencer** (see 18.2.4.8) and **uvm_reg_map::get_adapter** (see 18.2.4.9). **get_local_map** shall return the most recent *map* assigned via **set_local_map**. The value returned by **get_local_map** prior to any **set_local_map** call is *null*.

19.1.1.2.8 Map

```
virtual function void set_map (uvm_reg_map map)
virtual function uvm_reg_map get_map()
```

The original map specified for the operation. The actual *map* used may differ when a test or sequence written at the block level is reused at the system level. **get_map** shall return the most recent *map* assigned via **set_map**. The value returned by **get_map** prior to any **set_map** call is *null*.

19.1.1.2.9 Door

```
virtual function void set_door (uvm_door_e door)
virtual function uvm_door_e get_door()
```

The door being used (see 17.2.2.2). **get_door** shall return the most recent *door* assigned via **set_door**. The value returned by **get_door** prior to any **set_door** call is undefined.

19.1.1.2.10 Parent sequence

```
virtual function void set_parent_sequence (uvm_sequence_base parent)
virtual function uvm_sequence_base get_parent_sequence()
```

The sequence from which the operation originated. **get_parent_sequence** shall return the most recent *parent* assigned via **set_parent_sequence**. The value returned by **get_parent_sequence** prior to any **set_parent_sequence** call is *null*.

19.1.1.2.11 Priority

```
virtual function void set_priority (int value)
virtual function int get_priority()
```

The priority requested of this transfer, as defined by **uvm_sequence_base::start_item** (see 14.2.6.2). **get_priority** shall return the most recent *value* assigned via **set_priority**. The value returned by **get_priority** prior to any **set_priority** call is `-1`.

19.1.1.2.12 Extension

```
virtual function void set_extension (uvm_object extension)
virtual function uvm_object get_extension()
```

A handle to optional user data, as conveyed in the call to **write**, **read**, **mirror**, or **update**, which is used to trigger the operation. **get_extension** shall return the most recent *extension* assigned via **set_extension**. The value returned by **get_extension** prior to any **set_extension** call is *null*.

19.1.1.2.13 Back door kind

```
virtual function void set_bd_kind (string bd_kind)
virtual function string get_bd_kind()
```

When *path* is `UVM_BACKDOOR` (see 19.1.1.2.9), this member specifies the abstraction kind for the back-door access, e.g., `"RTL"`. **get_bd_kind** shall return the most recent *bd_kind* assigned via **set_bd_kind**. The value returned by **get_bd_kind** prior to any **set_bd_kind** call is an empty string (`""`).

19.1.1.2.14 File name

```
virtual function void set_fname (string fname)
virtual function string get_fname()
```

The file name from where this transaction originated, if provided at the call site. **get_fname** shall return the most recent *fname* assigned via **set_fname**. The value returned by **get_fname** prior to any **set_fname** call is an empty string (`""`).

19.1.1.2.15 Line number

```
virtual function void set_line (int line)
virtual function int get_line()
```

The line number from where this transaction originated, if provided at the call site. **get_line** shall return the most recent *line* assigned via **set_line**. The value returned by **get_line** prior to any **set_line** call is `0`.

19.1.1.3 Methods for item sub-typing

19.1.1.3.1 new

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

Creates a new instance of this type, giving it the optional *name*.

19.1.1.3.2 convert2string

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

Returns a string showing the contents of this transaction.

19.1.2 uvm_reg_bus_op

A struct that defines a generic bus transaction for register and memory accesses, having *kind* (read or write), *address*, *data*, and *byte enable* information. If the bus is narrower than the register or memory location being accessed, there shall be multiples of these bus operations for every abstract **uvm_reg_item** transaction (see 19.1.1). In this case, *data* represents the portion of **uvm_reg_item::value** (see 19.1.1.2.4) being transferred during this bus cycle. If the bus is wide enough to perform the register or memory operation in a single cycle, *data* is the same as the **uvm_reg_item::value**.

uvm_reg_bus_op has the following *Properties*.

19.1.2.1 kind

```
uvm_access_e kind
```

The kind of access: **READ** or **WRITE**.

19.1.2.2 addr

```
uvm_reg_addr_t addr
```

This is the bus address.

19.1.2.3 data

```
uvm_reg_data_t data
```

The data to write or read. If the bus width is smaller than the register or memory width, *data* represents only the portion of *value* that is being transferred this bus cycle.

19.1.2.4 n_bits

```
int n_bits
```

The number of bits of **uvm_reg_item::value** (see 19.1.1.2.4) being transferred by this transaction.

19.1.2.5 byte_en

```
uvm_reg_byte_en_t byte_en
```

Specifies the enables for the byte lanes on the bus. Meaningful only when the bus supports byte enables and the operation originates from a field write/read.

19.1.2.6 status

```
uvm_status_e status
```

The result of the transaction: **UVM_IS_OK**, **UVM_HAS_X**, or **UVM_NOT_OK**. See 17.2.2.1.

19.2 Classes for adapting between register and bus operations

This subclause defines the classes used to convert transaction streams between generic register address/data reads and writes and physical bus accesses.

19.2.1 uvm_reg_adapter

This class defines an interface for converting between a **uvm_reg_bus_op** (see 19.1.2) and a specific bus transaction.

19.2.1.1 Class declaration

```
virtual class uvm_reg_adapter extends uvm_object
```

19.2.1.2 Common methods

19.2.1.2.1 new

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

Creates a new instance of this type, giving it the optional *name*.

19.2.1.2.2 supports_byte_enable

```
bit supports_byte_enable
```

Specifies this bit in extensions of this class if the bus protocol supports byte enables.

19.2.1.2.3 provides_responses

```
bit provides_responses
```

Specifies this bit in extensions of this class if the bus driver provides separate response items.

19.2.1.2.4 parent_sequence

```
uvm_sequence_base parent_sequence
```

Specifies this member in extensions of this class if the bus driver requires bus items be executed via a particular sequence base type. The sequence assigned to this member needs to implement `do_clone`.

19.2.1.2.5 reg2bus

```
pure virtual function uvm_sequence_item reg2bus(
    const ref uvm_reg_bus_op rw
)
```

Extensions of this class need to implement this method to convert the specified **uvm_reg_bus_op** (see 19.1.2) to a corresponding **uvm_sequence_item** subtype (see 14.1) that defines the bus transaction.

The method shall allocate a new bus-specific **uvm_sequence_item**, assign its members from the corresponding members from the given generic *rw* bus operation, then return it.

19.2.1.2.6 bus2reg

```
pure virtual function void bus2reg(
       uvm_sequence_item bus_item,
    ref uvm_reg_bus_op rw
)
```

Extensions of this class need to implement this method to copy members of the given bus-specific *bus_item* to corresponding members of the provided *rw* instance. Unlike **reg2bus** (see 19.2.1.2.5), the resulting transaction is not allocated from scratch. This is to accommodate applications where the bus response needs to be returned in the original request.

19.2.1.2.7 get_item

```
virtual function uvm_reg_item get_item()
```

This returns the bus-independent read/write information corresponding to the generic bus transaction currently translated to a bus-specific transaction. This function returns a value reference only when called in the **uvm_reg_adapter::reg2bus** method (see 19.2.1.2.5); otherwise, it returns *null*. The content of the returned **uvm_reg_item** instance (see 19.1.1) shall not be modified and is used strictly to obtain additional information about the operation.

19.2.2 uvm_reg_tlm_adapter

This class defines an interface for converting between **uvm_reg_bus_op** (see 19.1.2) and **uvm_tlm_gp** (see 12.3.4.3) items.

19.2.2.1 Class declaration

```
class uvm_reg_tlm_adapter extends uvm_reg_adapter
```

19.2.2.2 Methods

19.2.2.2.1 reg2bus

```
virtual function uvm_sequence_item reg2bus(
    const ref uvm_reg_bus_op rw
)
```

Converts a **uvm_reg_bus_op** struct (see 19.1.2) to a **uvm_tlm_gp** item (see 12.3.4.3).

19.2.2.2.2 bus2reg

```
virtual function void bus2reg(
       uvm_sequence_item bus_item,
    ref uvm_reg_bus_op rw
)
```

Converts a **uvm_tlm_gp** item (see 12.3.4.3) to a **uvm_reg_bus_op** (see 19.1.2), using the provided *rw* transaction.

19.3 uvm_reg_predictor

The **uvm_reg_predictor** class defines a predictor component, which is used to update the register model's mirror values based on transactions explicitly observed on a physical bus.

This class converts observed bus transactions of type `BUSTYPE` to generic registers transactions, determines how the register is being accessed based on the bus address, then updates the register's mirror value with the observed bus data, subject to the register's access mode. See 18.4.4.15 for details.

Memories can be large, so their accesses are not predicted.

19.3.1 Class declaration

```
class uvm_reg_predictor #(
    type BUSTYPE = int
) extends uvm_component
```

19.3.2 Variables

19.3.2.1 bus_in

```
uvm_analysis_imp #(
                       BUSTYPE,
    uvm_reg_predictor #(BUSTYPE)
) bus_in
```

Observed bus transactions of type `BUSTYPE` are received from this port and processed.

For each incoming transaction, the predictor attempts to find the register handle corresponding to the observed bus address.

If there is a match, the predictor calls the register's predict method, passing in the observed bus data. The register mirror is updated with this data, subject to its configured access behavior: `RW`, `RO`, etc. The predictor

also converts the bus transaction to a generic **uvm_reg_item** (see 19.1.1) and sends it out the **reg_ap** analysis port (see 19.3.2.2).

If the register is wider than the bus, the predictor collects the multiple bus transactions needed to determine the value being read or written.

19.3.2.2 reg_ap

```
uvm_analysis_port #(
    uvm_reg_item
) reg_ap
```

This is an analysis output port that publishes **uvm_reg_item** transactions (see 19.1.1) converted from bus transactions received on **bus_in** (see 19.3.2.1).

19.3.2.3 map

```
uvm_reg_map map
```

This is the map used to convert a bus address to the corresponding register or memory handle. It needs to be configured before the run phase.

19.3.2.4 adapter

```
uvm_reg_adapter adapter
```

This is the adapter used to convey the parameters of a bus operation in terms of a canonical **uvm_reg_bus_op** datum (see 19.1.2). The **uvm_reg_adapter** (see 19.2.1) needs to be configured before the run phase.

19.3.3 Methods

19.3.3.1 new

```
function new (
    string name,
    uvm_component parent
)
```

Creates a new instance of this type, giving it the optional *name* and *parent*.

19.3.3.2 pre_predict

```
virtual function void pre_predict(
    uvm_reg_item rw
)
```

Override this method to change the value or redirect the target register.

19.3.3.3 check_phase

```
virtual function void check_phase(
    uvm_phase phase
)
```

Checks that no pending register transactions are still queued.

19.4 Register sequence classes

This subclause defines the base classes used for register stimulus generation.

19.4.1 uvm_reg_sequence

This class provides base functionality for both user-defined `RegModel` test sequences and "register translation sequences."

- When used as a base for user-defined `RegModel` test sequences, this class provides convenience methods for reading and writing registers and memories. Users implement the `body` method to interact directly with the `RegModel` model [held in the **model** property (see 19.4.1.3.1)] or indirectly via the delegation methods in this class.
- When used as a translation sequence, objects of this class are executed directly on a bus sequencer, which are used in support of a layered sequencer use model; a predefined convert-and-execute algorithm is also provided.

Register operations do not require extending this class if none of the preceding services are needed. Register test sequences can be extended from the **uvm_sequence #(REQ,RSP)** base class (see 14.3) or even from outside a sequence.

This class defines convenience methods.

19.4.1.1 Class declaration

```
class uvm_reg_sequence #(
   type BASE = uvm_sequence #(uvm_reg_item)
) extends BASE
```

19.4.1.2 Common parameters

BASE

Specifies the sequence type from which to extend.

- When used as a translation sequence running on a bus sequencer, *BASE* shall be compatible with the sequence type expected by the bus sequencer.
- When used as a test sequence running on a particular sequencer, *BASE* shall be compatible with the sequence type expected by that sequencer.
- When used as a virtual test sequence without a sequencer, *BASE* does not need to be specified, i.e., the default specialization is adequate.

To maximize opportunities for reuse, user-defined RegModel sequences should "promote" the *BASE* parameter.

```
class my_reg_sequence #(type BASE=uvm_sequence #(uvm_reg_item))
                              extends uvm_reg_sequence #(BASE)
```

This way, the `RegModel` sequence can be extended from user-defined base sequences.

19.4.1.3 Variables

19.4.1.3.1 model

```
uvm_reg_block model
```

Block abstraction on which this sequence executes, defined only when this sequence is a user-defined test sequence.

19.4.1.3.2 adapter

```
uvm_reg_adapter adapter
```

This is an adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.

19.4.1.3.3 reg_seqr

```
uvm_sequencer #(
   uvm_reg_item
) reg_seqr
```

This is a layered upstream "register" sequencer.

It specifies the upstream sequencer between abstract register transactions and physical bus transactions. This is defined only when this sequence is a translation sequence to "pull" from an upstream sequencer.

19.4.1.4 Common methods

19.4.1.4.1 new

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

Creates a new instance, giving it the optional *name*.

19.4.1.4.2 body

```
virtual task body()
```

Continually retrieves a register transaction from the configured upstream sequencer, **reg_seqr** (see 19.4.1.3.3), and executes the corresponding bus transaction via **do_reg_item** (see 19.4.1.4.3).

User-defined `RegModel` test sequences need to override **body** and not call `super.body()`; otherwise, a warning shall be issued and the calling process does not return.

19.4.1.4.3 do_reg_item

```
virtual task do_reg_item(uvm_reg_item rw)
```

Executes the given register transaction, *rw*, via the sequencer on which this sequence was started. This uses the configured **adapter** (see 19.4.1.3.2) to convert the register transaction into the type expected by this sequencer.

19.4.1.5 Convenience read/write APIs

The following methods delegate to the corresponding method in the register or memory element. They allow a sequence **body** (see 19.4.1.4.2) to do reads and writes without having to explicitly supply itself to the *parent* sequence argument. Thus, a register write

```
model.regA.write(status, value, .parent(this))
```

could be written instead as

```
write_reg(model.regA, status, value)
```

19.4.1.5.1 write_reg

```
virtual task write_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes the given register *rg* using **uvm_reg::write** (see 18.4.4.9), supplying `this` as the *parent* argument. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

19.4.1.5.2 read_reg

```
virtual task read_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the given register *rg* using **uvm_reg::read** (see 18.4.4.10), supplying `this` as the *parent* argument. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

Thus,

```
read_reg(model.regA, status, value)
```

is equivalent to

```
model.regA.read(status, value, .parent(this))
```

19.4.1.5.3 poke_reg

```
virtual task poke_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Pokes the given register *rg* using **uvm_reg::poke** (see 18.4.4.11), supplying `this` as the *parent* argument. The default value of *lineno* shall be `0`.

Thus,

```
poke_reg(model.regA, status, value)
```

is equivalent to

```
model.regA.poke(status, value, .parent(this))
```

19.4.1.5.4 peek_reg

```
virtual task peek_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    output uvm_reg_data_t value,
    input string kind = "",
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Peeks the given register *rg* using **uvm_reg::peek** (see 18.4.4.12), supplying `this` as the *parent* argument. The default value of *lineno* shall be `0`.

Thus,

```
peek_reg(model.regA, status, value)
```

is equivalent to

```
model.regA.peek(status, value, .parent(this))
```

19.4.1.5.5 update_reg

```
virtual task update_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Updates the given register *rg* using **uvm_reg::update** (see 18.4.4.13), supplying `this` as the *parent* argument. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

Thus,

```
update_reg(model.regA, status, value)
```

is equivalent to

```
model.regA.update(status, value, .parent(this))
```

19.4.1.5.6 mirror_reg

```
virtual task mirror_reg(
    input uvm_reg rg,
    output uvm_status_e status,
    input uvm_check_e check = UVM_NO_CHECK,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Mirrors the given register *rg* using **uvm_reg::mirror** (see 18.4.4.14), supplying `this` as the *parent* argument. The default value of *check* shall be `UVM_NO_CHECK`. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

Thus,

```
mirror_reg(model.regA, status, value)
```

is equivalent to

```
model.regA.mirror(status, value, .parent(this))
```

19.4.1.5.7 write_mem

```
virtual task write_mem(
    input uvm_mem mem,
    output uvm status e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes the given memory *mem* using **uvm_mem::write** (see 18.6.5.1), supplying `this` as the *parent* argument. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

Thus,

```
write_mem(model.memA, status, offset, value)
```

is equivalent to

```
model.memA.write(status, offset, value, .parent(this))
```

19.4.1.5.8 read_mem

```
virtual task read_mem(
    input uvm_mem mem,
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the location *offset* in the given memory *mem* using **uvm_mem::read** (see 18.6.5.2), supplying `this` as the *parent* argument. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

Thus,

```
read_mem(model.memA, status, offset, value)
```

is equivalent to

```
model.memA.read(status, offset, value, .parent(this))
```

19.4.1.5.9 poke_mem

```
virtual task poke_mem(
    input uvm_mem mem,
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t value,
    input string kind = "",
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Pokes the location *offset* in the given memory *mem* using **uvm_mem::poke** (see 18.6.5.5), supplying `this` as the *parent* argument. The default value of *lineno* shall be `0`.

Thus,

```
poke_mem(model.memA, status, offset, value)
```

is equivalent to

```
model.memA.poke(status, offset, value, .parent(this))
```

19.4.1.5.10 peek_mem

```
virtual task peek_mem(
    input uvm_mem mem,
    output uvm_status_e status,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t value,
    input string kind = "",
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Peeks the location *offset* in the given memory *mem* using **uvm_mem::peek** (see 18.6.5.6), supplying `this` as the *parent* argument. The default value of *lineno* shall be `0`.

Thus,

```
peek_mem(model.memA, status, offset, value)
```

is equivalent to

```
model.memA.peek(status, offset, value, .parent(this))
```

19.4.2 uvm_reg_frontdoor

The facade class for register and memory front-door access; a user-defined front-door access sequence.

This is the base class for user-defined access to register and memory reads and writes through a physical interface. By default, different registers and memories are mapped to different addresses in the address space and are accessed via those physical addresses.

The front-door allows access using a non-linear and/or non-mapped mechanism. Users can extend this class to provide the physical access to these registers.

19.4.2.1 Class declaration

```
virtual class uvm_reg_frontdoor extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_sequence_item)
)
```

19.4.2.2 Variables

19.4.2.2.1 rw_info

```
uvm_reg_item rw_info
```

Holds information about the register being read or written.

19.4.2.2.2 sequencer

```
uvm_sequencer_base sequencer
```

This is the sequencer executing the operation.

19.4.2.3 Methods

new

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

This is a constructor; it creates a new object given the optional *name*.

19.5 uvm_reg_backdoor

The base class for user-defined back-door register and memory access.

This class can be extended by users to provide user-specific back-door access to registers and memories that are not implemented in pure SystemVerilog or that are not accessible using the default DPI back-door mechanism.

19.5.1 Class declaration

```
virtual class uvm_reg_backdoor extends uvm_object
```

19.5.2 Methods

19.5.2.1 new

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

Initializes an instance of the user-defined back-door class for the specified register or memory.

19.5.2.2 do_pre_read

```
protected task do_pre_read(
    uvm_reg_item rw
)
```

Executes the pre-read callbacks.

This method shall be called as the first statement in a user extension of the **read** method (see 19.5.2.7).

19.5.2.3 do_post_read

```
protected task do_post_read(
    uvm_reg_item rw
)
```

Executes the post-read callbacks

This method shall be called as the last statement in a user extension of the **read** method (see 19.5.2.7).

19.5.2.4 do_pre_write

```
protected task do_pre_write(
    uvm_reg_item rw
)
```

Executes the pre-write callbacks.

This method shall be called as the first statement in a user extension of the **write** method (see 19.5.2.6).

19.5.2.5 do_post_write

```
protected task do_post_write(
    uvm_reg_item rw
)
```

Execute the post-write callbacks

This method shall be called as the last statement in a user extension of the **write** method (see 19.5.2.6).

19.5.2.6 write

```
protected task write(
    uvm_reg_item rw
)
```

A user-defined back-door write operation.

This calls **do_pre_write** (see 19.5.2.4) and deposits the specified value in the specified register HDL implementation. Then it calls **do_post_write** (see 19.5.2.5) and returns an indication of the success of the operation.

19.5.2.7 read

```
virtual task read(
    uvm_reg_item rw
)
```

A user-defined back-door read operation; overload this method only if the back door requires the use of task.

This calls **do_pre_read** (see 19.5.2.2) and peeks the current value of the specified HDL implementation. Then it calls **do_post_read** (see 19.5.2.3) and returns the current value and an indication of the success of the operation.

By default, this calls **read_func** (see 19.5.2.8) to perform the peek step.

19.5.2.8 read_func

```
virtual function void read_func(
    uvm_reg_item rw
)
```

A user-defined back-door read operation.

This peeks the current value in the HDL implementation. It then returns the current value and an indication of the success of the operation.

19.5.2.9 is_auto_updated

```
virtual function bit is_auto_updated(
    uvm_reg_field field
)
```

Indicates if a **wait_for_change** method (see 19.5.2.10) is implemented for the given *field*.

This returns `TRUE` if and only if **wait_for_change** is implemented to watch for changes in the HDL implementation of the specified *field*.

19.5.2.10 wait_for_change

```
virtual local task wait_for_change(
    uvm_object element
)
```

Wait for a change in the value of the register or memory element in the DUT.

19.5.2.11 pre_read

```
virtual task pre_read(
    uvm_reg_item rw
)
```

Called before any user-defined back-door register reads.

The registered callback methods are invoked after the invocation of this method.

19.5.2.12 post_read

```
virtual task post_read(
    uvm_reg_item rw
)
```

Called after any user-defined back-door register reads.

The registered callback methods are invoked before the invocation of this method.

19.5.2.13 pre_write

```
virtual task pre_write(
    uvm_reg_item rw
)
```

Called before any user-defined back-door register writes.

The registered callback methods are invoked after the invocation of this method.

The written value, if modified, modifies the actual value to be written.

19.5.2.14 post_write

```
virtual task post_write(
   uvm_reg_item rw
)
```

Called after any user-defined back-door register writes.

The registered callback methods are invoked before the invocation of this method.

19.6 UVM HDL back-door access support routines

These routines provide an interface to the DPI/PLI/VHPI/application-API implementation of back-door access used by registers.

NOTE—To avoid having to use these APIs, define the `UVM_HDL_NO_DPI` macro at compile time.

19.6.1 Variables

UVM_HDL_MAX_WIDTH

```
parameter int UVM_HDL_MAX_WIDTH = `UVM_HDL_MAX_WIDTH
```

Specifies the maximum size bit vector for back-door access.

The default value of *UVM_HDL_MAX_WIDTH* shall be `` `UVM_HDL_MAX_WIDTH ``.

19.6.2 Methods

19.6.2.1 uvm_hdl_check_path

```
import "DPI-C" context function int uvm_hdl_check_path(
   string path
)
```

Checks that the given HDL *path* exists. This returns `0` if not found, `1` otherwise.

19.6.2.2 uvm_hdl_deposit

```
import "DPI-C" context function int uvm_hdl_deposit(
   string path,
   uvm_hdl_data_t value
)
```

Specifies the given HDL *path* as the specified *value*. This returns `1` if the call succeeded, `0` otherwise.

19.6.2.3 uvm_hdl_force

```
import "DPI-C" context function int uvm_hdl_force(
   string path,
   uvm_hdl_data_t value
)
```

Forces the *value* on the given *path*. This returns `1` if the call succeeded, `0` otherwise.

19.6.2.4 uvm_hdl_force_time

```
task uvm_hdl_force_time(
    string path,
    uvm_hdl_data_t value,
    time force_time = 0
)
```

Forces the *value* on the given *path* for the specified amount of *force_time*. If *force_time* is `0`, **uvm_hdl_deposit** (see 19.6.2.2) is called. This returns `1` if the call succeeded, `0` otherwise. The default value of *force_time* shall be `0`.

19.6.2.5 uvm_hdl_release_and_read

```
import "DPI-C" context function int uvm_hdl_release_and_read(
       string path,
    inout uvm_hdl_data_t value
)
```

Releases a value previously specified by **uvm_hdl_force** (see 19.6.2.3). This returns `1` if the call succeeded, `0` otherwise. *value* is reset to the HDL value after the release. For `reg`, the value remains the forced value until it has been procedurally reassigned. For `wire`, the value changes immediately to the resolved value of its continuous drivers, if any. If `none`, its value remains as forced until the next direct assignment.

19.6.2.6 uvm_hdl_release

```
import "DPI-C" context function int uvm_hdl_release(
    string path
)
```

Releases a value previously specified by **uvm_hdl_force** (see 19.6.2.3). This returns `1` if the call succeeded, `0` otherwise.

19.6.2.7 uvm_hdl_read

```
import "DPI-C" context function int uvm_hdl_read(
    string path,
    output uvm_hdl_data_t value
)
```

Returns the *value* at the given *path*. This returns `1` if the call succeeded, `0` otherwise.

Annex A

(informative)

Bibliography

[B1] IEEE Std 1003.1™-2008, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®).[9, 10]

[B2] IEEE Std 1666™, IEEE Standard for System C Language Reference Manual.

[B3] IEEE Std 1685™, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.

[9]The IEEE standards or products referred to in Annex A are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.
[10]IEEE publications are available from the Institute of Electrical and Electronics Engineers (http://standards.ieee.org).

Annex B

(normative)

Macros and defines

UVM includes some macros to allow the user to specify intent without the need to specify multiple types of SystemVerilog constructs. These macros assist with reporting, object behavior (interaction with the factory and field usage in comparing/copying, etc.), sequence specification, and UVM TLM connection.

UVM also includes some defines to specify sizing in the register space and to determine version of the UVM standard and/or implementation.

B.1 Report macros

This set of macros provides wrappers around the **uvm_report_*** reporting functions (see F.2.2). The macros serve two essential purposes, as follows:

— To reduce the processing overhead associated with filtered out info messages, a check is made against the report's verbosity setting and the action for the id/severity pair before any string formatting is performed.
— The `__FILE__ and `__LINE__ information is automatically provided to the underlying **uvm_report_*** call. Having the file and line number from where a report was issued aides in debugging.

The macros also enforce a verbosity setting of UVM_NONE for warnings, errors, and fatals.

While the implementation of the macros is not strictly defined, the following restrictions are placed upon the implementation:

— The implementation shall be a complete SystemVerilog statement.
— No time consuming statements shall be introduced.

B.1.1 Basic messaging macros

B.1.1.1 `uvm_info

```
`uvm_info(ID, MSG, VERBOSITY)
```

Calls **uvm_report_info** (see F.3.2.3) if *VERBOSITY* is lower than the configured verbosity of the associated reporter. *ID* is the message tag and *MSG* is the message text. The file and line are also sent to the **uvm_report_info** call.

B.1.1.2 `uvm_warning

```
`uvm_warning(ID, MSG)
```

Calls **uvm_report_warning** (see F.3.2.3) with a verbosity of UVM_NONE. *ID* is the message tag and MSG is the message text. The file and line are also sent to the **uvm_report_warning** call.

B.1.1.3 `uvm_error

```
`uvm_error(ID, MSG)
```

Calls **uvm_report_error** (see F.3.2.3) with a verbosity of `UVM_NONE`. *ID* is the message tag and *MSG* is the message text. The file and line are also sent to the **uvm_report_error** call.

B.1.1.4 `uvm_fatal

```
`uvm_fatal(ID, MSG)
```

Calls **uvm_report_fatal** (see F.3.2.3) with a verbosity of `UVM_NONE`. *ID* is the message tag and *MSG* is the message text. The file and line are also sent to the **uvm_report_fatal** call.

B.1.1.5 `uvm_info_context

```
`uvm_info_context(ID, MSG, VERBOSITY, RO)
```

Operates identically to **`uvm_info** (see B.1.1.1), but requires that the context, or **uvm_report_object** (see 6.3), in which the message is printed is also explicitly supplied as a macro argument. *RO* is an optional reference to a user-declared report message that will be used by the macro.

B.1.1.6 `uvm_warning_context

```
`uvm_warning_context(ID, MSG, RO)
```

Operates identically to **`uvm_warning** (see B.1.1.2), but requires that the context, or **uvm_report_object** (see 6.3), in which the message is printed is also explicitly supplied as a macro argument. *RO* is an optional reference to a user-declared report message that will be used by the macro.

B.1.1.7 `uvm_error_context

```
`uvm_error_context(ID, MSG, RO)
```

Operates identically to **`uvm_error** (see B.1.1.3), but requires that the context, or **uvm_report_object** (see 6.3), in which the message is printed is also explicitly supplied as a macro argument.

B.1.1.8 `uvm_fatal_context

```
`uvm_fatal_context(ID, MSG, RO)
```

Operates identically to **`uvm_fatal** (see B.1.1.4), but requires that the context, or **uvm_report_object** (see 6.3), in which the message is printed is also explicitly supplied as a macro argument. *RO* is an optional reference to a user-declared report message that will be used by the macro.

B.2 Utility and field macros for components and objects

B.2.1 Utility macros

The *utils* macros define the infrastructure needed to enable the object/component for correct factory operation. See B.2.1.2 and B.2.1.3 for details.

A *utils* macro should be used inside every user-defined class that extends **uvm_object** directly or indirectly (see 5.3), including **uvm_sequence_item** (see 14.1) and **uvm_component** (see 13.1).

Examples

Using the *utils* macro for a user-defined object.

```
class mydata extends uvm_object
       `uvm_object_utils(mydata)
       // declare data properties
   function new(string name="mydata_inst")
       super.new(name)
   endfunction
endclass
```

Using the *utils* macro for a user-defined component.

```
class my_comp extends uvm_component
       `uvm_component_utils(my_comp)
       // declare data properties
   function new(string name, uvm_component parent=null)
       super.new(name,parent)
   endfunction
endclass
```

B.2.1.1 `uvm_field_utils_begin and `uvm_field_utils_end

These macros provide a default implementation of **uvm_object::do_execute_op** (see 5.3.13.1).

These macros form a block in which **`uvm_field_*** macros (see B.2.2) can be placed, as shown in the following meta-example:

```
`uvm_field_utils_begin(TYPE)
   `uvm_field_* macros here
`uvm_field_utils_end
```

These macros do not perform factory registration nor implement the **get_type_name** and **create** methods. Use this form when custom implementations of these two methods are needed or to set up field macros for an abstract class (i.e., a virtual class).

B.2.1.2 `uvm_object_utils, `uvm_object_param_utils, `uvm_object_utils_begin, `uvm_object_param_utils_begin, and `uvm_object_utils_end; `uvm_object_abstract_utils, `uvm_object_abstract_param_utils, `uvm_object_abstract_utils_begin, and `uvm_object_abstract_param_utils_begin

The **`uvm_object*_utils_begin** and **_end** macros, as well as their **`uvm_component*** counterparts, use these macros.

uvm_object-based class declarations (see 5.3) may contain one of these forms of utility macros.

For simple objects with no field macros, use:

```
`uvm_object_utils(TYPE)
```

For simple objects with field macros, use:

```
`uvm_object_utils_begin(TYPE)
   `uvm_field_* macro invocations here
`uvm_object_utils_end
```

For parameterized objects with no field macros, use:

```
`uvm_object_param_utils(TYPE)
```

For parameterized objects with field macros, use:

```
`uvm_object_param_utils_begin(TYPE)
   `uvm_field_* macro invocations here
`uvm_object_utils_end
```

Simple (non-parameterized) objects use the **uvm_object_utils*** versions, which do the following:

- Implement **get_type_name** (see 5.3.4.7), which returns *TYPE* as a string.
- Implement **create** (see 5.3.5.1), which allocates an object of type *TYPE* by calling its constructor with no arguments. When defined, *TYPE*'s constructor shall have default values on all it arguments.
- Implement a `typedef` of type `uvm_object_registry#(TYPE, "TYPE")`, with the name *type_id*.
- Implement the static **get_type** method (see 5.3.4.5), which returns a factory proxy object for the type.
- Implement the virtual **get_object_type** method (see 5.3.4.6), which works just like the static **get_type** method (see 5.3.4.5), but operates on an already allocated object.

Parameterized classes shall use the **uvm_object_param_utils*** versions. They differ from **uvm_object_utils*** only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

Abstract classes shall use the **uvm_object_abstract*** versions. They only differ from the **uvm_object_utils*** and **uvm_object_param_utils*** in that they use **uvm_abstract_object_registry** instead of **uvm_object_registry**.

Any macros with `_begin` suffixes are the same as the non-suffixed versions except they also start a block in which **`uvm_field_*** macros (see B.2.2) can be placed. This block shall be terminated by **`uvm_object_utils_end**.

B.2.1.3 `uvm_component_utils, `uvm_component_param_utils, `uvm_component_utils_begin, `uvm_component_param_utils_begin, and `uvm_component_end; `uvm_component_abstract_utils, `uvm_component_abstract_param_utils, `uvm_component_abstract_utils_begin, and `uvm_component_abstract_param_utils_begin

uvm_component-based class declarations (see 13.1) may contain one of these forms of utility macros.

For simple components with no field macros, use:

```
`uvm_component_utils(TYPE)
```

For simple components with field macros, use:

```
`uvm_component_utils_begin(TYPE)
   `uvm_field_* macro invocations here
`uvm_component_utils_end
```

For parameterized components with no field macros, use:

```
`uvm_component_param_utils(TYPE)
```

For parameterized components with field macros, use:

```
`uvm_component_param_utils_begin(TYPE)
   `uvm_field_* macro invocations here
`uvm_component_utils_end
```

Simple (non-parameterized) components use the **uvm_component_utils*** versions, which do the following:

- Implement **get_type_name** (see 5.3.4.7), which returns *TYPE* as a string.
- Implement **create** (see 5.3.5.1), which allocates an object of type *TYPE* using a two argument constructor. *TYPE*'s constructor shall have a *name* and a *parent* argument.
- Implement a `typedef` of type `uvm_component_registry#(TYPE, "TYPE")`, with the name *type_id*.
- Implement the static **get_type** method (see 5.3.4.5), which returns a factory proxy object for the type.
- Implement the virtual **get_object_type** method (see 5.3.4.6), which works just like the static **get_type** method (see 5.3.4.5), but operates on an already allocated object.

Parameterized classes shall use the **uvm_component_param_utils*** versions. These differ from **uvm_component_utils*** only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

Abstract classes shall use the **uvm_component_abstract*** versions. They only differ from the **uvm_component_utils*** and **uvm_component_param_utils*** in that they use **uvm_abstract_component_registry** instead of **uvm_component_registry**.

Any macros with `_begin` suffixes are the same as the non-suffixed versions except they also start a block in which **`uvm_field_*** macros (see B.2.2) can be placed. This block shall be terminated by **`uvm_component_utils_end**.

B.2.1.4 `uvm_object_registry

```
`uvm_object_registry(T,S)
```

Registers a **uvm_object**-based class *T* and lookup string *S* with the factory (see 5.3). *S* typically is the name of the class in quotes (`""`). The **`uvm_object_utils** family of macros (see B.2.1.2) uses this macro.

B.2.1.5 `uvm_component_registry

```
`uvm_component_registry(T,S)
```

Registers a **uvm_component**-based class *T* and lookup string *S* with the factory (see 13.1). *S* typically is the name of the class in quotes (`""`). The **`uvm_component_utils** family of macros (see B.2.1.3) uses this macro.

B.2.2 Field macros

The **`uvm_field_*** macros are invoked inside of the **`uvm_*_utils_begin** / **`uvm_*_utils_end** macro blocks (see B.2.1) to form "automatic" implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.

NOTE—By using these macros, the **do_*** methods inherited from **uvm_object** (see 5.3) do not have to be implemented. However, be aware that the field macros expand into general inline code that is not as run-time efficient nor as flexible as direct implementations of the **do_*** methods.

Each **`uvm_field_*** macro is named according to the particular data type it handles: integrals, strings, objects, queues, etc.; each also has at least two arguments: *ARG* and *FLAG*.

a) *ARG*—is the instance name of the variable, whose type shall be compatible with the macro being invoked. *ARG* shall not be a constant, such as `const` variables, compile time constants, or elaboration time constants.

b) *FLAG*—specifies the operations, abstraction, radix, and recursion policy to be applied by the macro when operating on *ARG*. Flag options are bitwise `OR`ed to form a complete description.

Any number of *field operation types* (see F.2.1.9) and *field macro operation flags* (see F.2.1.10) may be bitwise `OR`ed, however, the negative flags (`UVM_NO*`) take precedence over their positive counterparts (including `UVM_ALL_ON` and `UVM_DEFAULT`). For example: `(UVM_COPY | UVM_NOCOPY)` is treated as `UVM_NOCOPY` within the macro. Additionally, the packing and unpacking operations are paired, such that `UVM_PACK` enables both packing and unpacking, and `UVM_NOPACK` disables both packing and unpacking.

All `uvm_radix_enum` values (see F.2.1.5), as well as all `uvm_recursion_policy_enum` values (see F.2.1.6), are supported as flags; however, only a single radix and a single recursion policy can be selected for a given declaration. This has the consequence of mandating the two `enum`s shall use compatible values. The result of bitwise `OR`ing multiple radix or recursion policy values within a single `FLAG` description is undefined. If no `uvm_radix_enum` value is provided, then the macro shall proceed as though `UVM_NORADIX` had been provided. If no `uvm_recursion_policy_enum` is provided, then the macro shall proceed as though `UVM_DEFAULT_POLICY` had been provided.

B.2.2.1 `uvm_field_* macros

These are macros that implement data operations for scalar properties.

B.2.2.1.1 `uvm_field_int

```
`uvm_field_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for any packed integral property.

ARG is an integral non constant property of the class, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.1.2 `uvm_field_object

```
`uvm_field_object(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a **uvm_object**-based property (see 5.3).

ARG is an object property of the class, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.1.3 `uvm_field_string

```
`uvm_field_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a string property.

ARG is a string property of the class, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.1.4 `uvm_field_enum

```
`uvm_field_enum(T, ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an enumerated property.

T is an enumerated type, *ARG* is an instance of that type, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.1.5 `uvm_field_real

```
`uvm_field_real(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for any real property.

ARG is a real property of the class, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.1.6 `uvm_field_event

```
`uvm_field_event(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an event property.

ARG is an event property of the class, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.2 `uvm_field_sarray_* macros

These are macros that implement data operations for one-dimensional static array properties.

B.2.2.2.1 `uvm_field_sarray_int

```
`uvm_field_sarray_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional static array of integrals.

ARG is a one-dimensional static array of integrals, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.2.2 `uvm_field_sarray_object

```
`uvm_field_sarray_object(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional static array of **uvm_object**-based objects (see 5.3).

ARG is a one-dimensional static array of **uvm_object**-based objects, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.2.3 `uvm_field_sarray_string

```
`uvm_field_sarray_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional static array of strings.

ARG is a one-dimensional static array of strings, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.2.4 `uvm_field_sarray_enum

```
`uvm_field_sarray_enum(T, ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional static array of enums.

T is a one-dimensional static array of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.3 `uvm_field_array_* macros

These are macros that implement data operations for one-dimensional dynamic array properties.

B.2.2.3.1 `uvm_field_array_int

```
`uvm_field_array_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional dynamic array of integrals.

ARG is a one-dimensional dynamic array of integrals, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.3.2 `uvm_field_array_object

```
`uvm_field_array_object(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional dynamic array of **uvm_object**-based objects (see 5.3).

ARG is a one-dimensional dynamic array of **uvm_object**-based objects, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.3.3 `uvm_field_array_string

```
`uvm_field_array_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional dynamic array of strings.

ARG is a one-dimensional dynamic array of strings, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.3.4 `uvm_field_array_enum

```
`uvm_field_array_enum(T, ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional dynamic array of enums.

T is a one-dimensional dynamic array of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.4 `uvm_field_queue_* macros

These are macros that implement data operations for dynamic queues.

B.2.2.4.1 `uvm_field_queue_int

```
`uvm_field_queue_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a queue of integrals.

ARG is a one-dimensional queue of integrals, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.4.2 `uvm_field_queue_object

```
`uvm_field_queue_object(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional queue of **uvm_object**-based objects (see 5.3).

ARG is a one-dimensional queue of **uvm_object**-based objects, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.4.3 `uvm_field_queue_string

```
`uvm_field_queue_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional queue of strings.

ARG is a one-dimensional queue of strings, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.4.4 `uvm_field_queue_enum

```
`uvm_field_queue_enum(T, ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for a one-dimensional queue of enums.

T is a one-dimensional queue of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.5 `uvm_field_aa_*_string macros

These are macros that implement data operations for associative arrays indexed by *string*.

B.2.2.5.1 `uvm_field_aa_int_string

```
`uvm_field_aa_int_string(ARG,FLAG)
```

Implements the data operations for an associative array of integrals indexed by *string*.

ARG is the name of a property that is an associative array of integrals with the *string* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.5.2 `uvm_field_aa_object_string

```
`uvm_field_aa_object_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of **uvm_object**-based objects (see 5.3).

ARG is the name of a property that is an associative array of objects with the *string* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.5.3 `uvm_field_aa_string_string

```
`uvm_field_aa_string_string(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of string indexed by *string*.

ARG is the name of a property that is an associative array of strings with the *string* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6 `uvm_field_aa_*_int macros

These are macros that implement data operations for associative arrays indexed by an integral type.

B.2.2.6.1 `uvm_field_aa_object_int

```
`uvm_field_aa_object_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of **uvm_object**-based objects (see 5.3) indexed by the *int* data type.

ARG is the name of a property that is an associative array of objects with the *int* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.2 `uvm_field_aa_int_int

```
`uvm_field_aa_int_int(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *int* data type.

ARG is the name of a property that is an associative array of integrals with the *int* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.3 `uvm_field_aa_int_int_unsigned

```
`uvm_field_aa_int_int_unsigned(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

ARG is the name of a property that is an associative array of integrals with the *int unsigned* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.4 `uvm_field_aa_int_integer

```
`uvm_field_aa_int_integer(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

ARG is the name of a property that is an associative array of integrals with the *integer* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.5 `uvm_field_aa_int_integer_unsigned

```
`uvm_field_aa_int_integer_unsigned(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *integer unsigned* data type.

ARG is the name of a property that is an associative array of integrals with the *integer unsigned* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.6 `uvm_field_aa_int_byte

```
`uvm_field_aa_int_byte(ARG,FLAG)
```

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

ARG is the name of a property that is an associative array of integrals with the *byte* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.7 `uvm_field_aa_int_byte_unsigned

```
`uvm_field_aa_int_byte_unsigned(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

ARG is the name of a property that is an associative array of integrals with the *byte unsigned* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.8 `uvm_field_aa_int_shortint

```
`uvm_field_aa_int_shortint(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *shortint* data type.

ARG is the name of a property that is an associative array of integrals with the *shortint* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.9 `uvm_field_aa_int_shortint_unsigned

```
`uvm_field_aa_int_shortint_unsigned(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

ARG is the name of a property that is an associative array of integrals with the *shortint unsigned* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.10 `uvm_field_aa_int_longint

```
`uvm_field_aa_int_longint(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

ARG is the name of a property that is an associative array of integrals with the *longint* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.11 `uvm_field_aa_int_longint_unsigned

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

ARG is the name of a property that is an associative array of integrals with the *longint unsigned* key, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.12 `uvm_field_aa_int_key

```
`uvm_field_aa_int_key(KEY,ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by any integral key data type.

KEY is the data type of the integral key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.2.6.13 `uvm_field_aa_int_enumkey

```
`uvm_field_aa_int_enumkey(KEY,ARG,FLAG=UVM_DEFAULT)
```

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

KEY is the enumeration type of the integral key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise `OR` of one or more flag settings as described in B.2.2.

B.2.3 Recording macros

The recording macros assist users who implement the **uvm_object::do_record** method (see 5.3.7.2). They help ensure that the fields are recorded using a user-specific API. Unlike the **uvm_recorder** policy

(see 16.4.1), fields recorded using the macros do not lose type information—they are passed directly to the user-specific API. This results in more efficient recording and no artificial limit on bit-widths.

B.2.3.1 `uvm_record_attribute

```
`uvm_record_attribute(TR_HANDLE, NAME, VALUE)
```

This is a macro to hide a tool-specific interface for recording attributes (fields) to a transaction database.

TR_HANDLE shall always be passed to **uvm_recorder::get_record_attribute_handle** (see 16.4.6.9).

The default implementation of the macro passes *NAME* and *VALUE* through to the **uvm_recorder::record_generic** method (see 16.4.6.7).

B.2.3.2 `uvm_record_int

```
`uvm_record_int(NAME,VALUE,SIZE[,RADIX])
```

This macro takes the same arguments as the **uvm_recorder::record_field** method (see 16.4.6.1) (including the optional *RADIX*).

The default implementation passes the name/value pair to **`uvm_record_attribute** if enabled (see B.2.3.1); otherwise, the information is passed to **uvm_recorder::record_field**.

B.2.3.3 `uvm_record_string

```
`uvm_record_string(NAME,VALUE)
```

This macro takes the same arguments as the **uvm_recorder::record_string** method (see 16.4.6.5).

The default implementation passes the name/value pair to **`uvm_record_attribute** if enabled (see B.2.3.1); otherwise, the information is passed to **uvm_recorder::record_string**.

B.2.3.4 `uvm_record_time

```
`uvm_record_time(NAME,VALUE)
```

This macro takes the same arguments as the **uvm_recorder::record_time** method (see 16.4.6.6).

The default implementation passes the name/value pair to **`uvm_record_attribute** if enabled (see B.2.3.1); otherwise, the information is passed to **uvm_recorder::record_time**.

B.2.3.5 `uvm_record_real

```
`uvm_record_real(NAME,VALUE)
```

This macro takes the same arguments as the **uvm_recorder::record_field_real** method (see 16.4.6.3).

The default implementation passes the name/value pair to **`uvm_record_attribute** if enabled (see B.2.3.1); otherwise, the information is passed to **uvm_recorder::record_field_real**.

B.2.3.6 `uvm_record_field

```
`uvm_record_field(NAME,VALUE)
```

This is a macro for recording arbitrary name-value pairs into a transaction recording database. It requires a valid transaction handle, as provided by the **uvm_transaction::begin_tr** (see 5.4.2.4) and **uvm_component::begin_tr** (see 13.1.6.3) methods.

The default implementation passes the name/value pair to **`uvm_record_attribute** if enabled (see B.2.3.1); otherwise, the information is passed to **uvm_recorder::record_generic** (see 16.4.6.7), with the *VALUE* being converted to a string using `%p` notation, i.e.,

```
recorder.record_generic(NAME,$sformatf("%p",VALUE))
```

B.2.4 Packing macros

The packing macros assist users who implement the **uvm_object::do_pack** method (see 5.3.10.2). They help ensure that the pack operation is the exact inverse of the unpack operation. See also B.2.5.

The `N` versions of these macros take an explicit size argument, which shall be a compile-time constant value greater than `0`.

B.2.4.1 `uvm_pack_intN

```
`uvm_pack_intN(VAR,SIZE)
```

Packs an integral variable.

B.2.4.2 `uvm_pack_enumN

```
`uvm_pack_enumN(VAR,SIZE)
```

Packs an enum variable.

B.2.4.3 `uvm_pack_sarrayN

```
`uvm_pack_sarrayN(VAR,SIZE)
```

Packs a static array of integrals.

B.2.4.4 `uvm_pack_arrayN

```
`uvm_pack_arrayN(VAR,SIZE)
```

Packs a dynamic array of integrals.

B.2.4.5 `uvm_pack_queueN

```
`uvm_pack_queueN(VAR,SIZE)
```

Packs a queue of integrals.

B.2.4.6 `uvm_pack_int

```
`uvm_pack_int(VAR)
```

Packs an integral variable without having to also specify the bit size.

B.2.4.7 `uvm_pack_enum

```
`uvm_pack_enum(VAR)
```

Packs enumeration value. Packing this does not require its type be specified.

B.2.4.8 `uvm_pack_string

```
`uvm_pack_string(VAR)
```

Packs a string variable.

B.2.4.9 `uvm_pack_real

```
`uvm_pack_real(VAR)
```

Packs a variable of type `real`.

B.2.4.10 `uvm_pack_sarray

```
`uvm_pack_sarray(VAR)
```

Packs a static array without having to also specify the bit size of its elements. The array elements shall be of `integral` type.

B.2.4.11 `uvm_pack_array

```
`uvm_pack_array(VAR)
```

Packs a dynamic array without having to also specify the bit size of its elements. The array size shall be non-zero. The array elements shall be of `integral` type.

B.2.4.12 `uvm_pack_queue

```
`uvm_pack_queue(VAR)
```

Packs a queue without having to also specify the bit size of its elements. The queue shall not be empty. The array elements shall be of `integral` type.

B.2.5 Unpacking macros

The unpacking macros assist users who implement the **uvm_object::do_unpack** method (see 5.3.11.2). They help ensure that the unpack operation is the exact inverse of the pack operation. See also B.2.4.

The `N` versions of these macros take an explicit size argument, which shall be a compile-time constant value greater than `0`.

B.2.5.1 `uvm_unpack_intN

```
`uvm_unpack_intN(VAR,SIZE)
```

Unpacks an integral variable.

B.2.5.2 `uvm_unpack_enumN

```
`uvm_unpack_enumN(VAR,SIZE,TYPE)
```

Unpacks an enum of type *TYPE* into *VAR*.

B.2.5.3 `uvm_unpack_sarrayN

```
`uvm_unpack_sarrayN(VAR,SIZE)
```

Unpacks a static array of integrals.

B.2.5.4 `uvm_unpack_arrayN

```
`uvm_unpack_arrayN(VAR,SIZE)
```

Unpacks into a dynamic array of integrals.

B.2.5.5 `uvm_unpack_queueN

```
`uvm_unpack_queueN(VAR,SIZE)
```

Unpacks into a queue of integrals.

B.2.5.6 `uvm_unpack_int

```
`uvm_unpack_int(VAR)
```

Unpacks an integral variable without having to also specify the bit size.

B.2.5.7 `uvm_unpack_enum

```
`uvm_unpack_enum(VAR)
```

Unpacks enumeration value, which requires its type be specified.

B.2.5.8 `uvm_unpack_string

```
`uvm_unpack_string(VAR)
```

Unpacks a string variable.

B.2.5.9 `uvm_unpack_real

```
`uvm_unpack_real(VAR)
```

Unpacks a variable of type `real`.

B.2.5.10 `uvm_unpack_sarray

```
`uvm_unpack_sarray(VAR)
```

Unpacks a static array without having to also specify the bit size of its elements. The array elements shall be of `integral` type.

B.2.5.11 `uvm_unpack_array

```
`uvm_unpack_array(VAR)
```

Unpacks a dynamic array without having to also specify the bit size of its elements. The array size shall be non-zero. The array elements shall be of `integral` type.

B.2.5.12 `uvm_unpack_queue

```
`uvm_unpack_queue(VAR)
```

Unpacks a queue without having to also specify the bit size of its elements. The queue shall not be empty. The array elements shall be of `integral` type.

B.3 Sequence-related macros

B.3.1 Sequence action macros

These macros are used to create and start sequences and sequence items.

B.3.1.1 `uvm_create

```
`uvm_create(SEQ_OR_ITEM, SEQR=get_sequencer())
```

This action creates a child item or sequence using **create_item** (see 14.2.6.1), passing the return value of *SEQ_OR_ITEM*`.get_type` as *type_var*, *SEQR* as *l_sequencer*, and the string equivalent of *SEQ_OR_ITEM* as *name*. After this action completes, the user can manually specify values, manipulate the `rand_mode` and `constraint_mode`, etc.

B.3.1.2 `uvm_send

```
`uvm_send(SEQ_OR_ITEM, PRIORITY=-1)
```

This action processes a child item or sequence, without randomization.

a) For items, the implementation shall perform the following steps in order:
   1) The **start_item** method (see 14.2.6.2) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.
   2) The **finish_item** method (see 14.2.6.3) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.

b) For sequences, the **start** method (see 14.2.3.1) is called on *SEQ_OR_ITEM* with `this` for *parent_sequence*, *PRIORITY* for *this_priority*, and `0` for *call_pre_post*.

B.3.1.3 `uvm_rand_send

```
`uvm_rand_send(SEQ_OR_ITEM, PRIORITY=-1, CONSTRAINTS={})
```

This action processes a child item or sequence, with randomization.

a) For items, the implementation shall perform the following steps in order:
   1) The **start_item** method (see 14.2.6.2) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.

2) *SEQ_OR_ITEM* is randomized with *CONSTRAINTS*.
3) The **finish_item** method (see 14.2.6.3) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.

b) For sequences, the implementation shall perform the following steps in order:
1) If the **get_randomize_enabled** method (see 14.2.2.2) on *SEQ_OR_ITEM* returns `1`, then the sequence is randomized with *CONSTRAINTS*.
2) The **start** method (see 14.2.3.1) is called on *SEQ_OR_ITEM* with `this` for *parent_sequence*, *PRIORITY* for *this_priority*, and `0` for *call_pre_post*.

B.3.1.4 `uvm_do

```
`uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1,
   CONSTRAINTS={})
```

This action creates and processes a child item or sequence, with randomization.

The implementation shall perform the following steps in order:

a) **`uvm_create** (see B.3.1.1) is passed *SEQ_OR_ITEM* and *SEQR*.
b) **`uvm_rand_send** (see B.3.1.3) is passed *SEQ_OR_ITEM*, *PRIORITY*, and *CONSTRAINTS*.

B.3.2 Sequence library macros

B.3.2.1 `uvm_add_to_sequence_library

```
`uvm_add_to_seq_lib(TYPE,LIBTYPE)
```

Adds the given sequence *TYPE* to the given sequence library *LIBTYPE*.

This can be invoked once for a specific combination of *TYPE* and *LIBTYPE* within a sequence class to statically add the *TYPE* sequence to the *LIBTYPE* library. The sequence will then be available for selection and execution in all instances of the given library type.

A sequence class can invoke this macro for any number of combinations of *TYPE* and *LIBTYPE*, potentially adding *TYPE* to multiple *LIBTYPE*s.

B.3.2.2 `uvm_sequence_library_utils

```
`uvm_sequence_library_utils(TYPE)
```

This shall be invoked in extensions to the **uvm_sequence_library** class (see 14.4), with *TYPE* equal to the type of the extension. It enables the extension class to hold a static list of member sequences. See also B.3.2.1 for more information.

B.3.3 Sequencer subtypes

`uvm_declare_p_sequencer

```
`uvm_declare_p_sequencer(SEQUENCER)
```

This macro is used to declare a variable *p_sequencer* whose type is specified by *SEQUENCER*.

B.4 Callback macros

These macros are used to register and execute callbacks extending from **uvm_callbacks** (see 10.7.2).

B.4.1 `uvm_register_cb

```
`uvm_register_cb(T,CB)
```

Registers the given *CB* callback type with the given *T* object type. If a type-callback pair is not registered, then a warning shall be issued if an attempt is made to use the pair (add, delete, etc.).

Callback type *CB* is used to construct the name of bit that holds the result of registration. When the callback type is parameterized, this results in errors. To avoid this, a `typedef`-ed name should be used instead.

The registration typically occurs in the component that executes the given type of callback.

B.4.2 `uvm_set_super_type

```
`uvm_set_super_type(T,ST)
```

Defines the super type of *T* to be *ST*. This allows for derived class objects to inherit type-wide callbacks that are registered with the base class.

The registration typically occurs in the component that executes the given type of callback.

B.4.3 `uvm_do_callbacks

```
`uvm_do_callbacks(T,CB,METHOD)
```

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e., *this* object), which is, or is based on, type *T*. This macro takes the following arguments:

- *CB* is the class type of the callback objects to execute. The class type shall have a function signature that matches the *METHOD* argument.
- *T* is the type associated with the callback. Typically, an instance of type *T* is passed as one the arguments in the *METHOD* call.
- *METHOD* is the method call to invoke, containing all required arguments as if they were invoked directly.

B.4.4 `uvm_do_obj_callbacks

```
`uvm_do_obj_callbacks(T,CB,OBJ,METHOD)
```

Calls the given *METHOD* of all callbacks based on type *CB* registered with the given object, *OBJ*, which is or is based on type *T*.

This macro is identical to **`uvm_do_callbacks** macro (see B.4.3), but it has an additional *OBJ* argument to allow the specification of an external object with which to associate the callback.

B.4.5 `uvm_do_callbacks_exit_on

```
`uvm_do_callbacks_exit_on(T,CB,METHOD,VAL)
```

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e., *this* object), which is, or is based on, type *T*, returning upon the first callback returning the bit value given by *VAL*. This macro takes the following arguments:

- *CB* is the class type of the callback objects to execute. The class type shall have a function signature that matches the *METHOD* argument.
- *T* is the type associated with the callback. Typically, an instance of type *T* is passed as one the arguments in the *METHOD* call.
- *METHOD* is the method call to invoke, containing all required arguments as if they were invoked directly.
- *VAL*, when `1`, means return upon the first callback invocation that returns `1`. When `0`, it means return upon the first callback invocation that returns `0`.

Since this macro calls `return`, its use is restricted to implementations of functions that return a `bit` value.

B.4.6 `uvm_do_obj_callbacks_exit_on

```
`uvm_do_obj_callbacks_exit_on(T,CB,OBJ,METHOD,VAL)
```

Calls the given *METHOD* of all callbacks of type *CB* registered with the given object *OBJ*, which is, or is based on, type *T*, returning upon the first callback returning the bit value given by *VAL*. This is the same as **`uvm_do_callbacks_exit_on** (see B.4.5) except this has a specific object instance (instead of the implicit *this* instance) as the third argument.

Since this macro calls `return`, its use is restricted to implementations of functions that return a `bit` value.

B.5 UVM TLM implementation port declaration macros

The UVM TLM implementation declaration macros are a way for components to provide multiple implementation ports of the same implementation interface. When an implementation port is defined using the built-in set of `imps`, there shall be exactly one implementation of the interface.

Be aware each **`uvm_*interface*_imp_decl** creates a new class of type **uvm_*interface*_imp_*suffix***, where *suffix* is the input argument to the macro. Given this, typically these macros should be put into separate packages to avoid collisions and to allow sharing of the definitions.

B.5.1 `uvm_blocking_put_imp_decl

```
`uvm_blocking_put_imp_decl(SFX)
```

Defines the class **uvm_blocking_put_impSFX** for providing blocking `put` implementations. *SFX* is the suffix for the new class type.

B.5.2 `uvm_nonblocking_put_imp_decl

```
`uvm_nonblocking_put_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_put_impSFX** for providing non-blocking `put` implementations. *SFX* is the suffix for the new class type.

B.5.3 `uvm_put_imp_decl

```
`uvm_put_imp_decl(SFX)
```

Defines the class **uvm_put_impSFX** for providing both blocking and non-blocking `put` implementations. *SFX* is the suffix for the new class type.

B.5.4 `uvm_blocking_get_imp_decl

```
`uvm_blocking_get_imp_decl(SFX)
```

Defines the class **uvm_blocking_get_impSFX** for providing blocking `get` implementations. *SFX* is the suffix for the new class type.

B.5.5 `uvm_nonblocking_get_imp_decl

```
`uvm_nonblocking_get_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_get_impSFX** for providing non-blocking `get` implementations. *SFX* is the suffix for the new class type.

B.5.6 `uvm_get_imp_decl

```
`uvm_get_imp_decl(SFX)
```

Defines the class **uvm_get_impSFX** for providing both blocking and non-blocking `get` implementations. *SFX* is the suffix for the new class type.

B.5.7 `uvm_blocking_peek_imp_decl

```
`uvm_blocking_peek_imp_decl(SFX)
```

Defines the class **uvm_blocking_peek_impSFX** for providing blocking `peek` implementations. *SFX* is the suffix for the new class type.

B.5.8 `uvm_nonblocking_peek_imp_decl

```
`uvm_nonblocking_peek_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_peek_impSFX** for providing non-blocking `peek` implementations. *SFX* is the suffix for the new class type.

B.5.9 `uvm_peek_imp_decl

```
`uvm_peek_imp_decl(SFX)
```

Defines the class **uvm_peek_impSFX** for providing both blocking and non-blocking `peek` implementations. *SFX* is the suffix for the new class type.

B.5.10 `uvm_blocking_get_peek_imp_decl

```
`uvm_blocking_get_peek_imp_decl(SFX)
```

Defines the class **uvm_blocking_get_peek_impSFX** for providing blocking `get_peek` implementations. *SFX* is the suffix for the new class type.

B.5.11 `uvm_nonblocking_get_peek_imp_decl

```
`uvm_nonblocking_peek_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_get_peek_impSFX** for providing non-blocking `get_peek` implementations. *SFX* is the suffix for the new class type.

B.5.12 `uvm_get_peek_imp_decl

```
`uvm_get_peek_imp_decl(SFX)
```

Defines the class **uvm_get_peek_impSFX** for providing both blocking and non-blocking `get_peek` implementations. *SFX* is the suffix for the new class type.

B.5.13 `uvm_blocking_master_imp_decl

```
`uvm_blocking_master_imp_decl(SFX)
```

Defines the class **uvm_blocking_master_impSFX** for providing blocking `master` implementations. *SFX* is the suffix for the new class type.

B.5.14 `uvm_nonblocking_master_imp_decl

```
`uvm_nonblocking_master_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_master_impSFX** for providing non-blocking `master` implementations. *SFX* is the suffix for the new class type.

B.5.15 `uvm_master_imp_decl

```
`uvm_master_imp_decl(SFX)
```

Defines the class **uvm_master_impSFX** for providing both blocking and non-blocking `master` implementations. *SFX* is the suffix for the new class type.

B.5.16 `uvm_blocking_slave_imp_decl

```
`uvm_blocking_slave_imp_decl(SFX)
```

Defines the class **uvm_blocking_slave_impSFX** for providing blocking `slave` implementations. *SFX* is the suffix for the new class type.

B.5.17 `uvm_nonblocking_slave_imp_decl

```
`uvm_nonblocking_slave_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_slave_impSFX** for providing non-blocking `slave` implementations. *SFX* is the suffix for the new class type.

B.5.18 `uvm_slave_imp_decl

```
`uvm_slave_imp_decl(SFX)
```

Defines the class **uvm_slave_impSFX** for providing both blocking and non-blocking `slave` implementations. *SFX* is the suffix for the new class type.

B.5.19 `uvm_blocking_transport_imp_decl

```
`uvm_blocking_transport_imp_decl(SFX)
```

Defines the class **uvm_blocking_transport_impSFX** for providing blocking `transport` implementations. *SFX* is the suffix for the new class type.

B.5.20 `uvm_nonblocking_transport_imp_decl

```
`uvm_nonblocking_transport_imp_decl(SFX)
```

Defines the class **uvm_nonblocking_transport_impSFX** for providing non-blocking `transport` implementations. *SFX* is the suffix for the new class type.

B.5.21 `uvm_transport_imp_decl

```
`uvm_transport_imp_decl(SFX)
```

Defines the class **uvm_transport_impSFX** for providing both blocking and non-blocking `transport` implementations. *SFX* is the suffix for the new class type.

B.5.22 `uvm_analysis_imp_decl

```
`uvm_analysis_imp_decl(SFX)
```

Defines the class **uvm_analysis_impSFX** for providing an analysis implementation. *SFX* is the suffix for the new class type.

The analysis implementation is the `write` function. **`uvm_analysis_imp_decl** allows a scoreboard (or another analysis component) to support input from many places.

B.6 Size defines

B.6.1 `UVM_FIELD_FLAG_SIZE

Defines the number of bits in **uvm_field_flag_t** (see F.2.1.2). This size may be defined by the user; if so, the defined value needs to be greater than or equal to **UVM_FIELD_FLAG_RESERVED_BITS** (see F.2.1.1).

The default value of **`UVM_FIELD_FLAG_SIZE** is implementation specific; however, it shall be greater than or equal to **UVM_FIELD_FLAG_RESERVED_BITS**.

While the user may use this define to extend the size of **uvm_field_flag_t**, the exact definition of the lower bits, i.e., `[UVM_FIELD_FLAG_RESERVED_BITS-1:0]`, is reserved for an implementation.

B.6.2 `UVM_MAX_STREAMBITS

Defines the maximum bit vector size for integral types. Can be defined by the user; otherwise, this defaults to `4096`.

B.6.3 `UVM_PACKER_MIN_BITS

Defines the minimum number of bits that the default implementation of **uvm_packer** (see 16.5) shall be capable of storing internally. The default value of **`UVM_PACKER_MIN_BITS** shall be `32768`.

B.6.4 `UVM_REG_ADDR_WIDTH

This is the maximum address width in bits. The default value is `64`. This macro is used to define the **uvm_reg_addr_t** type (see 17.2.1.3).

B.6.5 `UVM_REG_DATA_WIDTH

This is the maximum data width in bits. The default value is `64`. This macro is used to define the **uvm_reg_data_t** type (see 17.2.1.2).

B.6.6 `UVM_REG_BYTENABLE_WIDTH

This is the maximum number of byte enable bits. The default value is one per byte in **`UVM_REG_DATA_WIDTH** (see B.6.5). This macro is used to define the **uvm_reg_byte_en_t** type (see 17.2.1.5).

B.6.7 `UVM_REG_CVR_WIDTH

This is the maximum number of bits in a **uvm_reg_cvr_t** (see 17.2.1.6) coverage model set. The default value is `32`.

B.7 UVM version globals

B.7.1 UVM_VERSION

```
`define UVM_VERSION 2016
```

A define shall be provided that indicates the version of UVM being used.

B.7.2 UVM_VERSION_STRING

A string parameter shall be provided in `uvm_pkg` that creates a version string identifying the implementation being used.

Annex C

(normative)

Configuration and resource classes

C.1 Overview

The configuration and resources classes provide access to a centralized database where type specific information can be stored and retrieved. The **uvm_resource_db** (see C.3.2) is the low-level resource database that users can write to or read from. The **uvm_config_db** (see C.4.2) is layered on top of the resource database and provides a typed interface for a configuration setting that is consistent with the configuration interface of **uvm_component** (see 13.1.5).

Information can be read from or written to the database at any time during simulation. A resource may be associated with a specific hierarchical scope of a **uvm_component** (see 13.1) or it may be visible to all components regardless of their hierarchical position.

C.2 Resources

C.2.1 Introduction

A *resource* is a parameterized container that holds arbitrary data. Resources can be used to configure components, supply data to sequences, or enable sharing of information across disparate parts of a testbench. They are stored using scoping information such that their visibility can be constrained to certain parts of the testbench. Resource containers can hold any type of data, as constrained by the data types available in SystemVerilog. Resources can contain scalar objects, class handles, queues, lists, or even virtual interfaces.

Resources are stored in a resource database such that each resource can be retrieved by name or by type. The database is globally accessible. To support type lookup, each resource has a static type handle that uniquely identifies the type of each specialized resource container.

Each resource has a set of scopes over which it is visible (see C.2.4). When a resource is looked up, the scope of the entity doing the looking up is supplied to the lookup function. This is called the *current scope*. If the current scope is in the set of scopes over which a resource is visible, the resource can be returned in the lookup.

Multiple resources that have the same name are stored in a queue. Each resource is pushed into a queue with the first one at the front of the queue and each subsequent one behind it. The same happens for multiple resources that have the same type. The resource queues are searched front to back, so those placed earlier in the queue have precedence over those placed later.

The precedence of resources with the same name or same type can be altered. One way is to set the *name* (see C.2.3.2.1) of the resource container to any arbitrary value. The search algorithm returns the resource with the highest precedence. In the case where there are multiple resources that match the search criteria and have the same (highest) precedence, the earliest one located in the queue is the one returned. Another way to change the precedence is to use the **set_priority** function (see C.2.4.5.3) to move a resource to either the front or back of the queue.

The classes defined here form the low-level layer of the resource database. The classes include the resource container and the database that holds the containers. The following set of classes are defined in this C.2:

a) **uvm_resource_types**—A class for containing definitions of types used by resources. See C.2.2.
b) **uvm_resource_base**—The base (untyped) resource class living in the resource database. This class includes the interface for setting a resource as read-only, notification, scope management, and altering search priority. See C.2.3.
c) **uvm_resource_pool**—The resource database. This is a singleton class object. See C.2.4.
d) **uvm_resource#(T)**—A parameterized resource container. This class includes the interfaces for reading and writing each resource. Because the class is parameterized, all the access functions are type safe. See C.2.5.

C.2.2 uvm_resource_types

This class provides a namespace for types that are used by the resource facility.

C.2.2.1 rsrc_q_t

A type defined as **uvm_queue#(uvm_resource_base)**. See C.2.3.

C.2.2.2 priority_e

Specifies the priority of a resource; the values are:

`PRI_HIGH`—resource is moved to the front of the queue.
`PRI_LOW`—resource is moved to the back of the queue.

C.2.3 uvm_resource_base

This is a non-parameterized base class for resources. It supports interfaces for scope matching and virtual functions for printing the resource.

C.2.3.1 Class declaration

```
virtual class uvm_resource_base extends uvm_object
```

C.2.3.2 Common methods

C.2.3.2.1 new

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

This is a constructor for **uvm_resource_base**. The constructor takes two arguments, the *name* of the resource and *s*, a regular expression, which represents the set of scopes over which this resource is visible. The default value of *s* shall be `"*"`.

C.2.3.2.2 get_type_handle

```
pure virtual function uvm_resource_base get_type_handle()
```

Intended to return the type handle of the resource container.

C.2.3.3 Read-only interface

C.2.3.3.1 set_read_only

```
function void set_read_only()
```

Establishes this resource as a read-only resource. An attempt to call **uvm_resource#(T)::write** (see C.2.5) on the resource shall generate an error.

C.2.3.3.2 is_read_only

```
function bit is_read_only()
```

Returns `1` if this resource has been set to read-only, `0` otherwise.

C.2.3.4 Notification

wait_modified

```
task wait_modified()
```

This task blocks until the resource has been modified, i.e., until a **uvm_resource#(T)::write** operation (see C.2.5.4.2) has been performed.

C.2.4 uvm_resource_pool

The global (singleton) resource database.

Each resource is stored both by primary name and by type handle. Each resource has a regular expression that represents the set of scopes over which it is visible.

Resources are added to the pool by calling **set_scope** (see C.2.4.3.1); they are retrieved from the pool by calling **get_by_name** (see C.2.4.4.4) or **get_by_type** (see C.2.4.4.6).

C.2.4.1 Class declaration

```
class uvm_resource_pool
```

C.2.4.2 Common methods

C.2.4.2.1 new

```
function new()
```

C.2.4.2.2 get

```
static function uvm_resource_pool get()
```

Returns the global resource pool.

This method is provided as a wrapper function to conveniently retrieve the resource pool via the **uvm_coreservice_t::get_resource_pool** method (see F.4.1.4.22).

C.2.4.3 Scope

Resource scope matching shall be determined using **uvm_is_match** (see F.3.3.1), with the stored scope as the *expr* and the lookup as *str*.

C.2.4.3.1 set_scope

```
virtual function void set_scope (
   uvm_resource_base rsrc,
   string scope
)
```

Adds a resource to the resource pool, with the provided *scope*. If the resource already exists in the pool, then its scope is replaced with the new *scope*.

The resource is inserted with low priority (see C.2.4.5) into both the name map and type map so it can be located by either. Later, other objects that want to access the resource need to retrieve it using the **lookup** interface (see C.2.4.4).

If *rsrc* is *null*, the implementation shall issue a warning message, and the request is ignored.

To override existing resources, use the **set_override** (see C.2.4.3.2), **set_name_override** (see C.2.4.3.3), or **set_type_override** (see C.2.4.3.2) functions.

C.2.4.3.2 set_override

```
virtual function void set_override(
   uvm_resource_base rsrc,
   string scope
)
```

Adds a resource to the resource pool, placing it with high priority (see C.2.4.5) in both the name and type maps.

This is functionally identical to calling **set_scope** (see C.2.4.3.1), immediately followed by **set_priority** (see C.2.4.5.3).

C.2.4.3.3 set_name_override

```
virtual function void set_name_override(
   uvm_resource_base rsrc,
   string scope
)
```

Adds a resource to the resource pool, placing it with high priority (see C.2.4.5) in the name map and low priority in the type map.

This is functionally identical to calling **set_scope** (see C.2.4.3.1), immediately followed by **set_priority_name** (see C.2.4.5.2).

C.2.4.3.4 set_type_override

```
virtual function void set_type_override(
   uvm_resource_base rsrc,
   string scope
)
```

Adds a resource to the resource pool, placing it with high priority (see C.2.4.5) in the type map and low priority in the name map.

This is functionally identical to calling **set_scope** (see C.2.4.3.1), immediately followed by **set_priority_type** (see C.2.4.5.1).

C.2.4.3.5 get_scope

```
virtual function bit get_scope (
   uvm_resource_base rsrc,
   output string scope)
```

If *rsrc* exists within the pool, then *scope* is set to the scope of the resource within the pool, and `1` is returned. If *rsrc* does not exist within the pool, then *scope* is set to an *empty string* (`""`), and `0` is returned.

C.2.4.3.6 delete

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

If *rsrc* exists within the pool, then it is removed from all internal maps. If the *rsrc* is *null*, or does not exist within the pool, then the request is silently ignored.

C.2.4.4 Lookup

This group of functions is for finding resources in the resource database.

- **lookup_name** (see C.2.4.4.1) and **lookup_type** (see C.2.4.4.5) locate the set of resources that matches the name or type (respectively) and is visible in the current scope. These functions return a queue of resources.
- **get_highest_precedence** (see C.2.4.4.2) traverses a queue of resources and returns the one with the highest precedence, i.e., the one whose precedence member has the highest value.
- **get_by_name** (see C.2.4.4.4) and **get_by_type** (see C.2.4.4.6) use **lookup_name** (see C.2.4.4.1) and **lookup_type** (see C.2.4.4.5) (respectively) and **get_highest_precedence** (see C.2.4.4.2) to find the resource with the highest priority that matches the other search criteria.

C.2.4.4.1 lookup_name

```
virtual function uvm_resource_types::rsrc_q_t lookup_name(
   string scope = "",
   string name,
   uvm_resource_base type_handle = null,
   bit rpterr = 1
)
```

This looks up resources by *name*. It returns a queue of resources that match the *name*, *scope*, and *type_handle*. If no resources match or if *name* is an *empty string* (`""`), the queue is returned empty. If *rpterr* is set to `1`, a warning is issued when no matches are found. If *type_handle* is *null*, a type check is not made and only resources that match the *name* and *scope* are returned. The default value of *rpterr* shall be `1`.

C.2.4.4.2 get_highest_precedence

```
static function uvm_resource_base get_highest_precedence(
   ref uvm_resource_types::rsrc_q_t q
)
```

This traverses a queue, *q*, of resources and returns the one with the highest precedence. When more than one resource with the highest precedence value exists, the first one that has that precedence is the one that is returned.

C.2.4.4.3 sort_by_precedence

```
static function void sort_by_precedence(
    ref uvm_resource_types::rsrc_q_t q
)
```

Given a list of resources, this sorts the resources in precedence order. The highest precedence resource is first in the list and the lowest precedence is last. Resources that have the same precedence are ordered by whichever is most recently set first.

C.2.4.4.4 get_by_name

```
virtual function uvm_resource_base get_by_name(
    string scope = "",
    string name,
    uvm_resource_base type_handle,
    bit rpterr = 1
)
```

This looks up a resource by *name*, *scope*, and *type_handle* and returns the highest precedence match. The *rpterr* flag indicates whether to report errors or not. The default value of *rpterr* shall be `1`.

C.2.4.4.5 lookup_type

```
virtual function uvm_resource_types::rsrc_q_t lookup_type(
    string scope = "",
    uvm_resource_base type_handle
)
```

This is a convenience method, functionally equivalent to calling **get_highest_precedence** (see C.2.4.4.2) on the result of **lookup_name** (see C.2.4.4.1).

This looks up resources by type. It returns a queue of resources that match the *type_handle* and *scope*. If no resources match, the returned queue is empty.

C.2.4.4.6 get_by_type

```
virtual function uvm_resource_base get_by_type(
    string scope = "",
    uvm_resource_base type_handle
)
```

This is a convenience method, functionally equivalent to calling **lookup_type** (see C.2.4.4.5) and returning the first resource in the queue.

This looks up a resource by *type_handle* and *scope*.

C.2.4.4.7 lookup_regex

```
virtual function uvm_resource_types::rsrc_q_t lookup_regex(
    string re,
    string scope
)
```

Looks for all the resources whose name matches the regular expression argument and whose scope matches the current scope.

C.2.4.4.8 lookup_scope

```
virtual function uvm_resource_types::rsrc_q_t lookup_scope(
    string scope
)
```

This is a utility function that answers the question: For a given *scope*, what resources are visible to it? It locates all the resources that are visible to a particular scope. This operation could be quite expensive, as it has to traverse all of the resources in the database.

C.2.4.5 Prioritization

The resource pool supports prioritization of the resources contained within. This prioritization is represented by two values: the priority and the precedence.

Priority is used to determine how the resource pool should act when new resources are added to the pool with identical types and/or names to pre-existing resources within the pool. The type and name maps maintained within the pool are maps of queues, allowing multiple resources to appear at a given key within the map. Resources with high priority are moved to the front of the queue, whereas resources with low priority are moved to the back of the queue. The default priority when adding resources to the pool is *low*.

Precedence is used to determine how the resource pool should react when it encounters multiple resources that match a given *scope* and *name* for a lookup (see C.2.4.4). Resources with a higher precedence outrank resources with a lower precedence. Precedence has no effect on type-based lookups. The default precedence when adding resources to the pool is determined via **get_default_precedence** (see C.2.4.5.5).

C.2.4.5.1 set_priority_type

```
virtual function void set_priority_type(
       uvm_resource_base rsrc,
    uvm_resource_types::priority_e pri
)
```

This changes the priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority only in the type map, leaving the name map untouched.

C.2.4.5.2 set_priority_name

```
virtual function void set_priority_name(
       uvm_resource_base rsrc,
    uvm_resource_types::priority_e pri
)
```

This changes the priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority only in the name map, leaving the type map untouched.

C.2.4.5.3 set_priority

```
virtual function void set_priority (
       uvm_resource_base rsrc,
    uvm_resource_types::priority_e pri
)
```

This changes the search priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority in both the name and type maps.

C.2.4.5.4 set_default_precedence

```
static function void set_default_precedence(
   int unsigned precedence
)
```

Overrides the current default precedence being used by the resource pool.

This method is provided as a wrapper function to conveniently assign the resource pool default precedence via the **uvm_coreservice_t::set_resource_pool_default_precedence** method (see F.4.1.4.23).

C.2.4.5.5 get_default_precedence

```
static function int unsigned get_default_precedence()
```

This method is provided as a wrapper function to conveniently retrieve the resource pool default precedence via the **uvm_coreservice_t::get_resource_pool_default_precedence** method (see F.4.1.4.24).

C.2.4.5.6 set_precedence

```
virtual function void set_precedence(
   uvm_resource_base r,
   int unsigned
   p=uvm_resource_pool::get_default_precedence()
)
```

Assigns the precedence value of a specific resource within the pool.

An implementation shall issue an warning message if the user passes in a *null* or a resource that has not previously been placed within the pool, and the request shall be ignored.

C.2.4.5.7 get_precedence

```
virtual function int unsigned get_precedence(
   uvm_resource_base r
)
```

Returns the precedence value of a specific resource within the pool.

An implementation shall issue an warning message if the argument *r* is set to *null* or if the resource is not stored within the resource pool, and the function shall return the current default precedence, as determined by **get_default_precedence** (see C.2.4.5.5).

C.2.4.5.8 get_highest_precedence

```
static function uvm_resource_base get_highest_precedence(
   ref uvm_resource_types::rsrc_q_t q
)
```

In a queue of resources, this locates the resource with the highest precedence. This function is static so that it can be called from anywhere.

C.2.5 uvm_resource #(T)

This is a parameterized resource. It provides access methods to store in, read from, and write to a resource.

C.2.5.1 Class declaration

```
class uvm_resource #(
    type T = int
) extends uvm_resource_base
```

C.2.5.2 new

```
function new(
    string name
)
```

Constructs a resource with the given instance *name*. If *name* is not supplied, then the resource is unnamed.

C.2.5.3 Type interface

Resources can be identified by type using a static type handle. The parent class provides the virtual function interface **get_type_handle** (see C.2.5.3.2). This can be implemented by returning the static type handle.

C.2.5.3.1 get_type

```
static function uvm_resource #(T) get_type()
```

This is a static function that returns the static type handle. The return type is *uvm_resource #(T)*, which is the type of the parameterized class.

C.2.5.3.2 get_type_handle

```
function uvm_resource_base get_type_handle()
```

This returns the static type handle of this resource in a polymorphic fashion. The return type is *uvm_resource_base*. This function is not static and, therefore, can only be used by instances of a parameterized resource.

C.2.5.4 Read/write interface

read (see C.2.5.4.1) and **write** (see C.2.5.4.2) provide a type-safe interface for retrieving and specifying the object in the resource container. The interface is type safe because the value argument for **write** and the return value of **read** are *T*, the type supplied in the class parameter. If either of these functions is used in an incorrect type context, the compiler will complain.

C.2.5.4.1 read

```
function T read(
    uvm_object accessor = null
)
```

This returns the object stored in the resource container.

C.2.5.4.2 write

```
function void write(
    T t,
    uvm_object accessor = null
)
```

This modifies the object stored in this resource container. If the resource is read-only, this generates an error message and returns without modifying the object in the container. Lastly, it replaces the value in the container with the value supplied as the argument, *t*, and releases any processes blocked on **uvm_resource_base::wait_modified** (see C.2.3.4). If the value to be written is the same as the value already present in the resource: the write is not done, the accessor record is not updated, and the modified bit is not set.

C.3 UVM resource database

C.3.1 Introduction

The **uvm_resource_db** class (see C.3.2) provides a convenience interface for the resources facility. In many cases, basic operations such as creating and specifying a resource or retrieving a resource could take multiple lines of code using the interfaces in **uvm_resource_base** (see C.2.3) or **uvm_resource#(T)** (see C.2.5). The **uvm_resource_db** convenience layer reduces many of those operations to a single line of code.

C.3.2 uvm_resource_db

All of the functions in `uvm_resource_db#(T)` are static, so they need to be called using the Scope Resolution Operator (`::`), e.g., `uvm_resource_db#(int)::set("A", "*", 17, this)`.

The parameter value "`int`" identifies the resource type as `uvm_resource#(int)`. Thus, the type of the object in the resource container is `int`. This maintains the type-safety characteristics of resource operations.

C.3.2.1 Class declaration

```
class uvm_resource_db #(
    type T = uvm_object
)
```

C.3.2.2 Methods

C.3.2.2.1 set

```
static function void set(
    input string scope,
    input string name,
       T val,
    input uvm_object accessor = null
)
```

Create a new resource, write a *val* to it, and *set* it into the database using *name* and *scope* as the lookup parameters. The *accessor* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

C.3.2.2.2 set_default

```
static function rsrc_t set_default(
   string scope,
   string name
)
```

Adds a new item into the resources database. The item will not be written to so it uses its default value. The resource is created using *name* and *scope* as the lookup parameters.

C.3.2.2.3 set_anonymous

```
static function void set_anonymous(
   input string scope,
      T val,
   input uvm_object accessor = null
)
```

Creates a new resource, writes a *val* to it, and *set*s it into the database. The resource has no *name* and, therefore, cannot be entered into the name map. It can still be retrieved by type, using *scope* for lookup purposes. The *accessor* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

C.3.2.2.4 get_by_name

```
static function rsrc_t get_by_name(
   string scope,
   string name,
   bit rpterr = 1
)
```

Imports a resource by *name*. The first argument is the current *scope* of the resource to be retrieved and the second argument is the *name*. The *rpterr* flag indicates whether or not to issue a warning if no matching resource is found. The default value of *rpterr* shall be `1`.

C.3.2.2.5 get_by_type

```
static function rsrc_t get_by_type(
   string scope
)
```

Returns a resource by type. The type is specified in the `db` class parameter, so the only argument to this function is the *scope*.

C.3.2.2.6 read_by_name

```
static function bit read_by_name(
   input string scope,
   input string name,
   output T val,
   input uvm_object accessor = null
)
```

Locates a resource by *name* and *scope* and reads its value. The value is returned through the output argument *val*. The return value is a bit that indicates whether or not the read was successful. The *accessor* is available

for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

C.3.2.2.7 read_by_type

```
static function bit read_by_type(
    input string scope,
    output T val,
    input uvm_object accessor = null
)
```

Reads a value by type. The value is returned through the output argument *val*. The *scope* is used for the lookup. The return value is a bit that indicates whether or not the read is successful. The *accessor* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

C.3.2.2.8 write_by_name

```
static function bit write_by_name(
    input string scope,
    input string name,
    input T val,
    input uvm_object accessor = null
)
```

Writes a *val* into the resources database. First, look up the resource by *name* and *scope*. If it is not located, **write_by_name** returns `0`. If the resource is located, then *val* is written to the resource. The *accessor* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

Because the *scope* is matched to a resource that may be a regular expression, and consequently may target other scopes beyond the *scope* argument, be careful using this function. If a **get_by_name** match (see C.3.2.2.4) is found for *name* and *scope*, then *val* is written to that matching resource and, thus, may impact other scopes that also match the resource.

C.3.2.2.9 write_by_type

```
static function bit write_by_type(
    input string scope,
    input T val,
    input uvm_object accessor = null
)
```

Writes a *val* into the resources database. First, look up the resource by type. If it is not located, **write_by_name** returns `0`. If the resource is located, then *val* is written to the resource. The *accessor* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method.

Because the *scope* is matched to a resource that may be a regular expression and consequently may target other scopes beyond the *scope* argument, be careful using this function. If a **get_by_name** match (see C.3.2.2.4) is found for *name* and *scope*, then *val* is written to that matching resource and, thus, may impact other scopes that also match the resource.

C.3.2.2.10 get_highest_precedence

```
static function uvm_resource #(T) get_highest_precedence(
   ref uvm_resource_types::rsrc_q_t q
)
```

In a queue of resources, this locates the first one with the highest precedence whose type is *T*. This function is static so that it can be called from anywhere.

C.4 UVM configuration database

C.4.1 Introduction

The **uvm_config_db** class (see C.4.2) provides a convenience interface on top of the **uvm_resource_db** (see C.3.2) to simplify the basic interface that is used for configuring **uvm_component** instances (see 13.1).

C.4.2 uvm_config_db

All of the functions in `uvm_config_db#(T)` are static, so they need to be called using the Scope Resolution Operator (`::`), e.g., `uvm_config_db#(int)::set(this, "*", "A")`.

The parameter value "`int`" identifies the configuration type as an int property.

C.4.2.1 Class declaration

```
class uvm_config_db#(
   type T = int
) extends uvm_resource_db#(T)
```

C.4.2.2 Methods

C.4.2.2.1 set

```
static function void set(
   uvm_component cntxt,
   string inst_name,
   string field_name,
   T value
)
```

Creates a new or updates an existing configuration specification for *field_name* in *inst_name* from *cntxt*. The setting is made at *cntxt*, with the full scope of the *set* being `{cntxt,".",inst_name}`. If *cntxt* is *null*, then *inst_name* provides the complete scope information of the setting. *field_name* is the target field. Both *inst_name* and *field_name* may be simplified notation or regular expression style expressions.

If a setting is made at build time, the *cntxt* hierarchy is used to determine the setting's precedence in the database. Settings from hierarchically higher levels have higher precedence. All settings use `PRI_HIGH` priority. A precedence setting of **uvm_resource_pool:: set_default_precedence** (see C.2.4.5.4) is used for the implicit top-level component (see F.7), and each hierarchical level below it is decremented by `1`.

After build time, all settings use the default precedence and `PRI_HIGH` priority. So, if at run time, a low-level component makes a run-time setting of some field, that setting shall have precedence over a setting from the test level that was made earlier in the simulation.

C.4.2.2.2 get

```
static function bit get(
       uvm_component cntxt,
       string inst_name,
       string field_name,
   inout T value
)
```

Returns the value for *field_name* in *inst_name*, using the component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an *empty string* (`""`) if the *cntxt* is the instance to which the configuration object applies. *field_name* is the specific field in the scope that is being searched.

C.4.2.2.3 exists

```
static function bit exists(
   uvm_component cntxt,
   string inst_name,
   string field_name,
   bit spell_chk = 0
)
```

Checks if a value for *field_name* is available in *inst_name*, using the component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an *empty string* (`""`) if the *cntxt* is the instance to which the configuration object applies. *field_name* is the specific field in the scope that is being searched for. *spell_chk* is available for an implementation to use for debug purposes only; its value shall have no functional effect on outcome of this method. The function returns `1` if a config parameter exists and `0` if it does not exist. The default value of *spell_chk* shall be `0`.

C.4.2.2.4 wait_modified

```
static task wait_modified(
   uvm_component cntxt,
   string inst_name,
   string field_name
)
```

Waits for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*. The task blocks until a new configuration setting is applied that effects the specified field.

C.4.2.3 Types

There are several convenience types for **uvm_config_db** (see C.4.2).

C.4.2.3.1 uvm_config_int

```
typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int
```

This is a convenience type for `uvm_config_db#(uvm_bitstream_t)`.

C.4.2.3.2 uvm_config_string

```
typedef uvm_config_db#(string) uvm_config_string
```

This is a convenience type for `uvm_config_db#(string)`.

C.4.2.3.3 uvm_config_object

```
typedef uvm_config_db#(uvm_object) uvm_config_object
```

This is a convenience type for `uvm_config_db#(uvm_object)`.

C.4.2.3.4 uvm_config_wrapper

```
typedef uvm_config_db#(uvm_object_wrapper) uvm_config_wrapper
```

This is a convenience type for `uvm_config_db#(uvm_object_wrapper)`.

Annex D

(normative)

Convenience classes, interface, and methods

This annex details additional convenience classes, interfaces, and methods that can be used in UVM.

D.1 uvm_callback_iter

This class can be used as part of the callbacks classes (see 10.7).

The **uvm_callback_iter** class is an iterator class for iterating over callback queues of a specific callback type. The typical usage of the class is:

```
uvm_callback_iter#(mycomp,mycb) iter = new(this)
for(mycb cb = iter.first(); cb != null; cb = iter.next())
   cb.dosomething()
```

D.1.1 Class declaration

```
class uvm_callback_iter#(
   type T = uvm_object,
   type CB = uvm_callback
)
```

This class shall not have a **type_id** declared (see 8.2.2).

D.1.2 Methods

D.1.2.1 new

```
function new(
   T obj
)
```

Creates a new callback iterator object. It is required that the object context be provided.

D.1.2.2 first

```
function CB first()
```

Returns the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty, *null* is returned.

D.1.2.3 last

```
function CB last()
```

Returns the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty, *null* is returned.

D.1.2.4 next

```
function CB next()
```

Returns the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, *null* is returned.

D.1.2.5 prev

```
function CB prev()
```

Returns the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, *null* is returned.

D.1.2.6 get_cb

```
function CB get_cb()
```

Returns the last callback accessed via a **first** (see D.1.2.2), **next** (see D.1.2.4), **last** (see D.1.2.3), or **prev** (see D.1.2.5) call.

D.2 Component interfaces

These interfaces can be used with **uvm_component** (see 13.1).

D.2.1 Factory interface

The factory interface provides convenient access to a portion of the **uvm_factory** interface (see 8.3.1). For creating new objects and components, the preferred method of accessing the factory is via the object or component wrapper (see 8.2.4 and 8.2.3, respectively). The wrapper also provides functions for setting type and instance overrides.

D.2.1.1 create_component

```
function uvm_component create_component (
   string requested_type_name,
   string name
)
```

A convenience function for **uvm_factory::create_component_by_name** (see 8.3.1.5), this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to

```
uvm_factory factory = uvm_factory::get()
factory.create_component_by_name(requested_type_name,
   get_full_name(), name, this)
```

If the factory determines that a type or instance override exists, the type of the component created may be different from the requested type. See **set_type_override** (see D.2.1.5) and **set_inst_override** (see D.2.1.6). See also 8.3.1 for details on factory operation.

D.2.1.2 create_object

```
function uvm_component create_object (
   string requested_type_name,
   string name = ""
)
```

A convenience function for **uvm_factory::create_object_by_name** (see 8.3.1.5), this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to

```
uvm_factory factory = uvm_factory::get()
factory.create_object_by_name(requested_type_name,
   get_full_name(), name, this)
```

If the factory determines that a type or instance override exists, the type of the object created may be different from the requested type. See 8.3.1 for details on factory operation.

D.2.1.3 set_type_override_by_type

```
static function void set_type_override_by_type (
   uvm_object_wrapper original_type,
   uvm_object_wrapper override_type,
   bit replace = 1
)
```

A convenience function for **uvm_factory::set_type_override_by_type** (see 8.3.1.4.2); this method is equivalent to:

```
uvm_factory factory = uvm_factory::get()
factory.set_type_override_by_type(original_type, override_type, replace)
```

The *original_type* and *override_type* arguments are lightweight proxies to the types they represent. See D.2.1.4 for information on usage.

D.2.1.4 set_inst_override_by_type

```
function void set_inst_override_by_type(
   string relative_inst_path,
   uvm_object_wrapper original_type,
   uvm_object_wrapper override_type
)
```

A convenience function for **uvm_factory::set_inst_override_by_type** (see 8.3.1.4.1); this method is equivalent to:

```
uvm_factory factory = uvm_factory::get()
factory.set_inst_override_by_type( original_type,
   override_type,
   {get_full_name(),".",
   relative_inst_path})
```

D.2.1.5 set_type_override

```
static function void set_type_override (
   string original_type_name,
```

```
    string override_type_name,
    bit replace = 1
)
```

A convenience function for **uvm_factory::set_type_override_by_name** (see 8.3.1.4.2), this method configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by original_type_name. This method is equivalent to:

```
uvm_factory factory = uvm_factory::get()
factory.set_type_override_by_name(original_type_name,
    override_type_name, replace)
```

original_type_name typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** (see D.2.1.1) or **create_object** (see D.2.1.2) with the same string and matching instance path produce the type represented by *override_type_name*. *override_type_name* shall refer to a preregistered type in the factory. The default value of *replace* shall be `1`.

D.2.1.6 set_inst_override

```
function void set_inst_override(
    string relative_inst_path,
    string original_type_name,
    string override_type_name
)
```

A convenience function for **uvm_factory::set_inst_override_by_name** (see 8.3.1.4.1), this method registers a factory override for components and objects created at this level of hierarchy or below. This method is equivalent to:

```
uvm_factory factory = uvm_factory::get()
factory.set_inst_override_by_name( original_type_name,
    override_type_name,
    {get_full_name(),".",
    relative_inst_path})
```

relative_inst_path is relative to this component and may include wildcards. *original_type_name* represents a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** (see D.2.1.1) or **create_object** (see D.2.1.2) with the same string and matching instance path produce the type represented by *override_type_name*. *override_type_name* shall refer to a preregistered type in the factory.

D.2.2 Hierarchical reporting interface

This interface provides versions of the **set_report_*** methods in the **uvm_report_object** base class (see 6.3) that are applied recursively to this component and all its children.

When a report is issued and its associated action has the `LOG` bit set to `1`, the report is sent to its associated `FILE` descriptor.

D.2.2.1 set_report_id_verbosity_hier and set_report_severity_id_verbosity_hier

```
function void set_report_id_verbosity_hier (
    string id,
    int verbosity
)
```

```
function void set_report_severity_id_verbosity_hier(
   uvm_severity severity,
   string id,
   int verbosity
)
```

These methods recursively associate the specified *verbosity* with reports of the given *severity*, *id*, or *severity-id* pair. A *verbosity* associated with a particular *severity-id* pair takes precedence over a *verbosity* associated with *id*, which takes precedence over a *verbosity* associated with a *severity*.

For a list of severities and their default verbosities, refer to 6.4.

D.2.2.2 set_report_severity_action_hier, set_report_id_action_hier, and set_report_severity_id_action_hier

```
function void set_report_severity_action_hier (
   uvm_severity severity,
   uvm_action action
)

function void set_report_id_action_hier (
   string id,
   uvm_action action
)

function void set_report_severity_id_action_hier(
   uvm_severity severity,
   string id,
   uvm_action action
)
```

These methods recursively associate the specified *action* with reports of the given *severity*, *id*, or *severity-id* pair. A *action* associated with a particular *severity-id* pair takes precedence over a *action* associated with *id*, which takes precedence over a *action* associated with a *severity*.

For a list of severities and their default actions, refer to 6.4.

D.2.2.3 set_report_default_file_hier, set_report_severity_file_hier, set_report_id_file_hier, and set_report_severity_id_file_hier

```
function void set_report_default_file_hier (
   UVM_FILE file
)

function void set_report_severity_file_hier (
   uvm_severity severity,
   UVM_FILE file
)

function void set_report_id_file_hier (
   string id,
   UVM_FILE file
)

function void set_report_severity_id_file_hier(
   uvm_severity severity,
```

```
    string id,
    UVM_FILE file
)
```

These methods recursively associate the specified `FILE` descriptor with reports of the given *severity*, *id*, or *severity-id* pair. A `FILE` associated with a particular *severity-id* pair takes precedence over a `FILE` associated with *id*, which takes precedence over a `FILE` associated with a *severity*.

For a list of severities and other information related to the report mechanism, refer to 6.4.

D.2.2.4 set_report_verbosity_level_hier

```
function void set_report_verbosity_level_hier (
    int verbosity
)
```

This method recursively specifies the maximum *verbosity* level for reports for this component and all those below it. Any report from this component subtree whose *verbosity* exceeds this maximum is ignored.

See 6.4 for a list of predefined message verbosity levels and their meaning.

D.3 uvm_reg_block access methods

These access methods can be used with **uvm_reg_block** (see 18.1.5).

D.3.1 write_reg_by_name

```
virtual task write_reg_by_name(
    output uvm_status_e status,
    input string name,
    input uvm_reg_data_t data,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes the named register. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

This is equivalent to **get_reg_by_name** (see 18.1.3.14) followed by **uvm_reg::write** (see 18.4.4.9).

D.3.2 read_reg_by_name

```
virtual task read_reg_by_name(
    output uvm_status_e status,
    input string name,
    output uvm_reg_data_t data,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
```

```
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the named register. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

This is equivalent to **get_reg_by_name** (see 18.1.3.14) followed by **uvm_reg::read** (see 18.4.4.10).

D.3.3 write_mem_by_name

```
virtual task write_mem_by_name(
    output uvm_status_e status,
    input string name,
    input uvm_reg_addr_t offset,
    input uvm_reg_data_t data,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Writes the named memory. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

This is equivalent to **get_mem_by_name** (see 18.1.3.16) followed by **uvm_mem::write** (see 18.6.5.1).

D.3.4 read_mem_by_name

```
virtual task read_mem_by_name(
    output uvm_status_e status,
    input string name,
    input uvm_reg_addr_t offset,
    output uvm_reg_data_t data,
    input uvm_door_e path = UVM_DEFAULT_DOOR,
    input uvm_reg_map map = null,
    input uvm_sequence_base parent = null,
    input int prior = -1,
    input uvm_object extension = null,
    input string fname = "",
    input int lineno = 0
)
```

Reads the named memory. The default value of *path* shall be `UVM_DEFAULT_DOOR`. The default value of *prior* shall be `-1`. The default value of *lineno* shall be `0`.

This is equivalent to **get_mem_by_name** (see 18.1.3.16) followed by **uvm_mem::read** (see 18.6.5.2).

D.4 Callback typedefs

The following **uvm_callbacks#(T,CB)** (see 10.7.2) typedefs are provided as a convenience to the user.

D.4.1 uvm_phase_cb_pool

```
typedef uvm_callbacks#(uvm_phase, uvm_phase_cb) uvm_phase_cb_pool
```

D.4.2 uvm_heartbeat_cbs_t

```
typedef uvm_callbacks#(uvm_objection, uvm_heartbeat_callback)
   uvm_heartbeat_cbs_t
```

D.4.3 uvm_objection_cbs_t

```
typedef uvm_callbacks#(uvm_objection, uvm_objection_callback)
```

D.4.4 uvm_report_cb

```
typedef uvm_callbacks#(uvm_report_object, uvm_report_catcher)
```

D.4.5 uvm_report_cb_inter

```
typedef uvm_callback_iter#(uvm_report_object, uvm_report_catcher)
   uvm_report_cb_iter
```

D.4.6 uvm_reg_cbs typedefs

These callback types can be used as part of the typedefs for **uvm_reg_cbs** (see 18.11.3).

D.4.6.1 uvm_reg_cb

```
typedef uvm_callbacks#(uvm_reg, uvm_reg_cbs) uvm_reg_cb
```

D.4.6.2 uvm_reg_cb_iter

```
typedef uvm_callback_iter#(uvm_reg, uvm_reg_cbs) uvm_reg_cb_iter
```

D.4.6.3 uvm_reg_bd_cb

```
typedef uvm_callbacks#(uvm_reg_backdoor, uvm_reg_cbs) uvm_reg_bd_cb
```

D.4.6.4 uvm_reg_bd_cb_iter

```
typedef uvm_callback_iter#(uvm_reg_backdoor, uvm_reg_cbs) uvm_reg_bd_cb_iter
```

D.4.6.5 uvm_mem_cb

```
typedef uvm_callbacks#(uvm_mem, uvm_reg_cbs) uvm_mem_cb
```

D.4.6.6 uvm_mem_cb_iter

```
typedef uvm_callback_iter#(uvm_mem, uvm_reg_cbs) uvm_mem_cb_iter
```

D.4.6.7 uvm_reg_field_cb

```
typedef uvm_callbacks#(uvm_reg_field, uvm_reg_cbs) uvm_reg_field_cb
```

D.4.6.8 uvm_reg_field_cb_iter

```
typedef uvm_callback_iter#(uvm_reg_field, uvm_reg_cbs) uvm_reg_field_cb_iter
```

D.4.6.9 uvm_vreg_cb

```
typedef uvm_callbacks#(uvm_vreg, uvm_vreg_cbs) uvm_vreg_cb
```

D.4.6.10 uvm_vreg_cb_iter

```
typedef uvm_callback_iter#(uvm_vreg, uvm_vreg_cbs) uvm_vreg_cb_iter
```

D.4.6.11 uvm_vreg_field_cb

```
typedef uvm_callbacks#(uvm_vreg_field, uvm_vreg_field_cbs) uvm_vreg_field_cb
```

D.4.6.12 uvm_vreg_field_cb_iter

```
typedef uvm_callback_iter#(uvm_vreg_field, uvm_vreg_field_cbs)
   uvm_vreg_field_cb_iter
```

Annex E

(normative)

Test sequences

E.1 uvm_reg_hw_reset_seq

Tests the hard reset values of registers.

The test sequence performs the following steps:

a) Resets the DUT and the block abstraction class associated with this sequence.
b) Reads all of the registers in the block, via all of the available address maps, comparing the value read with the expected reset value.

If a bit-type resource named "NO_REG_TESTS" or "NO_REG_HW_RESET_TEST" in the "REG::" namespace matches the full name of the block or register, the block or register is not tested.

This is usually the first test executed on any DUT.

E.1.1 Class declaration

```
class uvm_reg_hw_reset_seq extends uvm_reg_sequence #(
    uvm_sequence #(uvm_reg_item)
)
```

E.1.2 Variables

model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.1.2.1 Methods

E.1.2.1.1 new

```
function new(
    string name = "uvm_reg_hw_reset_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.1.2.1.2 body

```
virtual task body()
```

Executes the **uvm_reg_hw_reset_seq** sequence.

E.2 Bit bashing test sequences

This subclause defines classes that test individual bits of the registers defined in a register model.

E.2.1 uvm_reg_single_bit_bash_seq

Verifies the implementation of a single register by attempting to write `1`'s and `0`'s to every bit in it, via every address map in which the register is mapped, making sure that the resulting value matches the mirrored value.

If a bit-type resource named "`NO_REG_TESTS`" or "`NO_REG_BIT_BASH_TEST`" in the "`REG::`" namespace matches the full name of the register, the register is not tested.

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

E.2.1.1 Class declaration

```
class uvm_reg_single_bit_bash_seq extends
uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

E.2.1.2 Variables

rg

```
uvm_reg rg
```

The register to be tested.

E.2.1.3 Methods

new

```
function new(
   string name = "uvm_reg_single_bit_bash_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.2.2 uvm_reg_bit_bash_seq

Verifies the implementation of all registers in a block, and its sub-blocks, recursively, by executing the **uvm_reg_single_bit_bash_seq** sequence (see E.2.1) on it.

If a bit-type resource named "`NO_REG_TESTS`" or "`NO_REG_BIT_BASH_TEST`" in the "`REG::`" namespace matches the full name of the block, the block is not tested.

E.2.2.1 Class declaration

```
class uvm_reg_bit_bash_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

E.2.2.2 Variables

E.2.2.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.2.2.2.2 req_seq

```
protected uvm_reg_single_bit_bash_seq reg_seq
```

The sequence used to test one register.

E.2.2.3 Methods

E.2.2.3.1 new

```
function new(
   string name = "uvm_reg_bit_bash_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.2.2.3.2 body

```
virtual task body()
```

Executes the **uvm_reg_bit_bash_seq** sequence.

E.3 Register access test sequences

This subclause defines sequences that test DUT register access via the available front-door and back-door paths defined in the provided register model.

E.3.1 uvm_reg_single_access_seq

Verifies the accessibility of a register by writing it through its default address map, then reading it via the back door and reversing the process, making sure the resulting value matches the mirrored value.

If a bit-type resource named "`NO_REG_TESTS`" or "`NO_REG_ACCESS_TEST`" in the "`REG::`" namespace matches the full name of the register, the register is not tested.

Registers that either do not have an available back door, or only contain read-only fields and/or fields with unknown access policies, cannot be tested.

The DUT should be idle and not modify any register during this test.

E.3.1.1 Class declaration

```
class uvm_reg_single_access_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

E.3.1.2 Variables

rg

```
uvm_reg rg
```

The register to be tested.

E.3.1.3 Methods

new

```
function new(
    string name = "uvm_reg_single_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.3.2 uvm_reg_access_seq

Verifies the accessibility of all registers in a block by executing the **uvm_reg_single_access_seq** sequence (see E.3.1) on every register within it.

If a bit-type resource named "NO_REG_TESTS" or "NO_REG_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

E.3.2.1 Class declaration

```
class uvm_reg_access_seq extends uvm_reg_sequence #(
    uvm_sequence #(uvm_reg_item)
)
```

E.3.2.2 Variables

E.3.2.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.3.2.2.2 req_seq

```
protected uvm_reg_single_access_seq reg_seq
```

The sequence used to test one register.

E.3.2.3 Methods

E.3.2.3.1 new

```
function new(
    string name = "uvm_reg_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.3.2.3.2 body

```
virtual task body()
```

Executes the **uvm_reg_access_seq** sequence.

E.3.3 uvm_reg_mem_access_seq

Verifies the accessibility of all registers and memories in a block by executing the **uvm_reg_access_seq** (see E.3.2) and **uvm_mem_access_seq** sequence (see E.5.2), respectively, on every register and memory within it.

Blocks and registers with the `NO_REG_TESTS` or `NO_REG_ACCESS_TEST` attributes are not verified.

E.3.3.1 Class declaration

```
class uvm_reg_mem_access_seq extends uvm_reg_sequence #(
    uvm_sequence #(uvm_reg_item)
)
```

E.3.3.2 Methods

new

```
function new(
    string name = "uvm_reg_mem_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.4 Shared register and memory access test sequences

This subclause defines sequences for testing registers and memories that are shared between two or more physical interfaces, i.e., are associated with more than one **uvm_reg_map** instance (see 18.2).

E.4.1 uvm_reg_shared_access_seq

Verifies the accessibility of a shared register by writing through each address map, then reading it via every other address map in which the register is readable, making sure the resulting value matches the mirrored value.

If a bit-type resource named "`NO_REG_TESTS`" or "`NO_REG_SHARED_ACCESS_TEST`" in the "`REG::`" namespace matches the full name of the register, the register is not tested.

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

E.4.1.1 Class declaration

```
class uvm_reg_shared_access_seq extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_reg_item)
)
```

E.4.1.2 Variables

rg

```
uvm_reg rg
```

The register to be tested.

E.4.1.3 Methods

new

```
function new(
    string name = "uvm_reg_shared_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.4.2 uvm_mem_shared_access_seq

Verifies the accessibility of a shared memory by writing through each address map, then reading it via every other address map in which the memory is readable, making sure the resulting value matches the mirrored value.

If a bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", "NO_REG_SHARED_ACCESS_TEST", or "NO_MEM_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

The DUT should be idle and not modify the memory during this test.

E.4.2.1 Class declaration

```
class uvm_mem_shared_access_seq extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_reg_item)
)
```

E.4.2.2 Variables

mem

```
uvm_mem mem
```

The memory to be tested.

E.4.2.3 Methods

new

```
function new(
    string name = "uvm_mem_shared_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.4.3 uvm_reg_mem_shared_access_seq

Verifies the accessibility of all shared registers and memories in a block by executing the **uvm_reg_shared_access_seq** (see E.4.1) and **uvm_mem_shared_access_seq** (see E.4.2) sequences, respectively, on every register and memory within it.

If a bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", "NO_REG_SHARED_ACCESS_TEST", or "NO_MEM_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

E.4.3.1 Class declaration

```
class uvm_reg_mem_shared_access_seq extends
uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

E.4.3.2 Variables

E.4.3.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.4.3.2.2 req_seq

```
protected uvm_reg_shared_access_seq reg_seq
```

The sequence used to test one register.

E.4.3.2.3 mem_seq

```
protected uvm_mem_shared_access_seq mem_seq
```

The sequence used to test one memory.

E.4.3.3 Methods

E.4.3.3.1 new

```
function new(
   string name = "uvm_reg_mem_shared_access_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.4.3.3.2 body

```
virtual task body()
```

Executes the **uvm_reg_mem_shared_access_seq** sequence.

E.5 Memory access test sequences

E.5.1 uvm_mem_single_access_seq

Verifies the accessibility of a memory by writing through its default address map, then reading it via the back door and reversing the process, making sure the resulting value matches the mirrored value.

If a bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

Memories without an available back door cannot be tested.

The DUT should be idle and not modify the memory during this test.

E.5.1.1 Class declaration

```
class uvm_mem_single_access_seq extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_reg_item)
)
```

E.5.1.2 Variables

mem

```
uvm_mem mem
```

The memory to be tested.

E.5.1.3 Methods

new

```
function new(
    string name = "uvm_mem_single_access_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.5.2 uvm_mem_access_seq

Verifies the accessibility of all memories in a block by executing the **uvm_mem_single_access_seq** sequence (see E.5.1) on every memory within it.

If a bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

E.5.2.1 Class declaration

```
class uvm_mem_access_seq extends uvm_reg_sequence #(
    uvm_sequence #(uvm_reg_item)
)
```

E.5.2.2 Variables

E.5.2.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.5.2.2.2 mem_seq

```
protected uvm_mem_single_access_seq mem_seq
```

The sequence used to test one memory.

E.5.2.3 Methods

E.5.2.3.1 new

```
function new(
   string name = "uvm_mem_access_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.5.2.3.2 body

```
virtual task body()
```

Executes the **uvm_mem_access_seq** sequence.

E.6 Memory walking-ones test sequences

This subclause defines sequences for applying a “walking-ones” algorithm on one or more memories.

E.6.1 uvm_mem_single_walk_seq

Runs the walking-ones algorithm on the memory given by the **mem** class property (see E.6.1.2), which needs to be assigned prior to starting this sequence.

If a bit-type resource named `"NO_REG_TESTS"`, `"NO_MEM_TESTS"`, or `"NO_MEM_WALK_TEST"` in the `"REG::"` namespace matches the full name of the memory, the memory is not tested.

The walking-ones algorithm is performed for each map in which the memory is defined.

E.6.1.1 Class declaration

```
class uvm_mem_single_walk_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

E.6.1.2 Variables

mem

```
uvm_mem mem
```

The memory to test; this needs to be assigned prior to starting the sequence.

E.6.1.3 Methods

E.6.1.3.1 new

```
function new(
   string name = "uvm_mem_single_walk_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.6.1.3.2 body

```
virtual task body()
```

Performs the walking-ones algorithm on each map of the memory specified in **mem** (see E.6.1.2).

E.6.2 uvm_mem_walk_seq

Verifies all the memories in a block by executing the **uvm_mem_single_walk_seq** sequence (see E.6.1) on every memory within it.

If a bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_WALK_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

E.6.2.1 Class declaration

```
class uvm_mem_walk_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

E.6.2.2 Variables

E.6.2.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.6.2.2.2 mem_seq

```
protected uvm_mem_single_walk_seq mem_seq
```

The sequence used to test one memory.

E.6.3 Methods

E.6.3.1 new

```
function new(
   string name = "uvm_mem_walk_seq")
   super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.6.3.2 body

```
virtual task body()
```

Executes the **uvm_mem_walk_seq** sequence, one block at a time.

E.7 uvm_reg_mem_hdl_paths_seq

Verifies the correctness of the HDL paths specified for registers and memories.

This sequence checks that the specified back-door paths are indeed accessible by the simulator. By default, the check is performed for the default design abstraction. If the simulation contains multiple models of the DUT, HDL paths for multiple design abstractions can be checked.

If a path is not accessible by the simulator, it cannot be used for read/write back-door accesses. In that case, a warning is issued. A simulator may have finer-grained access permissions, such as separate read or write permissions. These extra access permissions are not checked.

The test is performed in zero time and does not require any reads/writes to/from the DUT.

E.7.1 Class declaration

```
class uvm_reg_mem_hdl_paths_seq extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_reg_item)
)
```

E.7.2 Variables

abstractions

```
string abstractions[$]
```

When not empty, this checks the HDL paths for the specified design abstractions. If empty, it checks the HDL path for the default design abstraction, as specified with **uvm_reg_block::set_default_hdl_path** (see 18.1.6.8). *abstractions* shall be a queue.

E.7.3 Methods

new

```
function new(
    string name = "uvm_reg_mem_hdl_paths_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.8 uvm_reg_mem_built_in_seq

A sequence that executes a user-defined selection of predefined register and memory test sequences.

E.8.1 Class declaration

```
virtual class uvm_reg_mem_built_in_seq extends uvm_reg_sequence
#(
    uvm_sequence #(uvm_reg_item)
)
```

E.8.2 Variables

E.8.2.1 model

The block to be tested. This is declared in the base class, e.g., `uvm_reg_block model`.

E.8.2.2 tests

```
bit [63:0] tests = UVM_DO_ALL_REG_MEM_TESTS
```

By default, all the predefined register and memory tests are executed. *tests* can also be set to execute any combination of the predefined register and/or memory tests by bitwise `OR`ing the desired values defined by **uvm_reg_mem_tests_e** (see 17.2.2.10). The default value of *tests* shall be `UVM_DO_ALL_REG_MEM_TESTS`.

E.8.3 Methods

E.8.3.1 new

```
function new(
    string name = "uvm_reg_mem_built_in_seq")
    super.name(new)
endfunction
```

Creates a new instance of the class with the given *name*.

E.8.3.2 body

```
virtual task body()
```

Executes any or all the built-in register and memory sequences.

Annex F

(normative)

Package scope functionality

F.1 Overview

UVM provides other functionality at the package scope including methods, enums, defines, and classes. Some of these are targeted towards specific aspects of the functionality described in this standard and others are useful across multiple aspects.

F.2 Types and enumerations

F.2.1 Field automation

F.2.1.1 UVM_FIELD_FLAG_RESERVED_BITS

```
parameter UVM_FIELD_FLAG_RESERVED_BITS
```

Represents the number of implementation reserved bits in **uvm_field_flag_t** (see F.2.1.2).

The exact value of the **UVM_FIELD_FLAG_RESERVED_BITS** is implementation specific; however, it shall be sufficiently large to store the result of a bitwise `OR`ing of the **uvm_radix_enum** (see F.2.1.5), **uvm_recursion_policy_enum** (see F.2.1.6), and **`uvm_field_*** macro flags (see F.2.1.9).

F.2.1.2 uvm_field_flag_t

```
bit[`UVM_FIELD_FLAG_SIZE-1:0] uvm_field_flag_t
```

The field flag type is a type for storing flag values passed onto the **`uvm_field_*** macros (see B.2.2).

F.2.1.3 uvm_bitstream_t

```
logic signed [`UVM_MAX_STREAMBITS-1:0]
```

The bitstream type is used as an argument type for passing integral values in such methods as **uvm_object::set_local** (see 5.3.12), **uvm_config_int** (see C.4.2.3.1), **uvm_printer::print_field** (see 16.2.3.8), **uvm_recorder::record_field** (see 16.4.6.1), **uvm_packer::pack_field** (see 16.5.4.8), and **uvm_packer::unpack_field** (see 16.5.4.16).

F.2.1.4 uvm_integral_t

```
logic signed [63:0]
```

The integral type is used as an argument type for passing integral values of `64` bits or less in such methods as **uvm_printer::print_field_int** (see 16.2.3.9), **uvm_recorder::record_field_int** (see 16.4.6.2), **uvm_packer::pack_field_int** (see 16.5.4.9), and **uvm_packer::unpack_field_int** (see 16.5.4.17).

F.2.1.5 uvm_radix_enum

Specifies the radix for printing or recording; it can contain the following literals:

UVM_BIN—Selects the binary (%b) format.

UVM_DEC—Selects the decimal (%d) format.

UVM_UNSIGNED—Selects the unsigned decimal (%u) format.

UVM_UNFORMAT2—Selects the unformatted 2-value data (%u) format.

UVM_UNFORMAT4—Selects the unformatted 4-value data (%z) format.

UVM_OCT—Selects the octal (%o) format.

UVM_HEX—Selects the hexadecimal (%h) format.

UVM_STRING—Selects the string (%s) format.

UVM_TIME—Selects the time (%t) format.

UVM_ENUM—Selects the enumeration value (*name*) format.

UVM_REAL—Selects real (%g) in the exponential or decimal format, whichever results in the shorter printed output.

UVM_REAL_DEC—Selects real (%f) in the decimal format.

UVM_REAL_EXP—Selects real (%e) in the exponential format.

UVM_NORADIX—No radix information is provided, the printer/recorder can use its default radix.

F.2.1.6 uvm_recursion_policy_enum

Specifies the policy for recursively entering object-based member variables; it has the following parameters:

UVM_DEFAULT_POLICY—No policy information is provided, the operation can use its default policy.

UVM_DEEP—Deep recursion. The operation shall recursively enter object-based member variables of the target object.

UVM_SHALLOW—Shallow recursion. The operation shall not recursively enter object-based member variables of the target object, instead treating them as simple references.

UVM_REFERENCE—Zero recursion. The target object itself shall be treated as a simple reference.

F.2.1.7 uvm_active_passive_enum

Defines whether a component, usually an agent, is in “active” mode or “passive” mode; it has the following values:

UVM_PASSIVE—“Passive” mode.

UVM_ACTIVE—“Active” mode.

F.2.1.8 Field operation types

The following flags describe the operation types supported by **uvm_field_op** (see 5.3.13.2):

UVM_COPY—The field will participate in **uvm_object::copy** (see 5.3.8.1).

UVM_COMPARE—The field will participate in **uvm_object::compare** (see 5.3.9.1).

UVM_PRINT—The field will participate in **uvm_object::print** (see 5.3.6.1).

UVM_RECORD—The field will participate in **uvm_object::record** (see 5.3.7.1).

UVM_PACK—The field will participate in **uvm_object::pack** (see 5.3.10.1).

UVM_UNPACK—The field will participate in **uvm_object::unpack** (see 5.3.11.1).

UVM_SET—The field will participate in **uvm_object::configuration** methods (see 5.3.12) and during the **uvm_component::apply_config_settings** operation (see 13.1.5.1).

F.2.1.9 Field macro operation flags

The following values describe additional flags that are supported by the **`uvm_field_*** macros (see B.2.2):

UVM_ALL_ON—Turn all operations on.

UVM_NOCOPY—The field will not participate in **uvm_object::copy** (see 5.3.8.1).

UVM_NOCOMPARE—The field will not participate in **uvm_object::compare** (see 5.3.9.1).

UVM_NOPRINT—The field will not participate in **uvm_object::print** (see 5.3.6.1).

UVM_NORECORD—The field will not participate in **uvm_object::record** (see 5.3.7.1).

UVM_NOPACK—The field will not participate in **uvm_object::pack** (see 5.3.10.1) and **uvm_object::unpack** (see 5.3.11.1).

UVM_NOSET—The field will not participate in **uvm_object::configuration** methods (see 5.3.12) or during the **uvm_component::apply_config_settings** operation (see 13.1.5.1).

F.2.1.10 UVM_DEFAULT

The default value for FLAG (see B.2.2) is UVM_DEFAULT, which is functionally identical to UVM_ALL_ON (see F.2.1.9).

F.2.2 Reporting

F.2.2.1 uvm_severity

An enumerated type representing all possible values for report severity; it has the following values:

UVM_INFO—Informative message.

UVM_WARNING—Indicates a potential problem.

UVM_ERROR—Indicates a real problem. Simulation continues subject to the configured message action.

UVM_FATAL—Indicates a problem from which simulation cannot recover.

F.2.2.2 uvm_action_type

An enumerated type representing all possible report actions; it has the following values:

UVM_NO_ACTION—No action is taken.

UVM_DISPLAY—Sends the report to the standard output.

UVM_LOG—Sends the report to the log file(s) specified within the report object.

UVM_COUNT—The report server shall increment its internal quit counter, see 6.5.1.

UVM_EXIT—Terminates the simulation immediately.

UVM_STOP—Causes $stop to be executed, putting the simulation into interactive mode.

UVM_RM_RECORD—Sends the report to the recorder.

The numeric values of **uvm_action_type** shall be *one-hot*, with UVM_NO_ACTION having a value of 0, so as to allow bitwise ORing via **uvm_action** (see F.2.2.3).

F.2.2.3 uvm_action

Defines an integer representing possible report actions (see F.2.2.2). Each of the bits of this type determines whether a corresponding action is taken (1) or not (0). Multiple actions can be specified by doing a bitwise OR of any number of the enumerated values. The exact size of the **uvm_action** vector is implementation specific; however, it shall be sufficiently large to store the result of bitwise ORing all uvm_action_type values (see F.2.2.2).

F.2.2.4 uvm_verbosity

Defines the standard verbosity levels for reports; it has the following parameters:

UVM_NONE—The report is always printed (NONE = 0); the verbosity level setting cannot disable it.

UVM_LOW—The report is issued if the configured verbosity is set to UVM_LOW (LOW = 100) or above.

UVM_MEDIUM—The report is issued if the configured verbosity is set to UVM_MEDIUM (MEDIUM = 200) or above.

UVM_HIGH—The report is issued if the configured verbosity is set to UVM_HIGH (HIGH = 300) or above.

UVM_FULL—The report is issued if the configured verbosity is set to UVM_FULL (FULL = 400) or above.

F.2.3 Port type

uvm_port_type_e

Specifies the type of port; it has the following parameters:

UVM_PORT—The port requires the interface that is its type parameter.

UVM_EXPORT—The port provides the interface that is its type parameter via a connection to some other export or implementation.

UVM_IMPLEMENTATION—The port provides the interface that is its type parameter, and it is bound to the component that implements the interface.

F.2.4 Sequences

F.2.4.1 uvm_sequencer_arb_mode

Specifies a sequencer's arbitration mode; it has the following parameters:

UVM_SEQ_ARB_FIFO—Requests are granted in FIFO order (the default).

UVM_SEQ_ARB_WEIGHTED—Requests are granted randomly by weight.

UVM_SEQ_ARB_RANDOM—Requests are granted randomly.

UVM_SEQ_ARB_STRICT_FIFO—Requests at the highest priority are granted in FIFO order.

UVM_SEQ_ARB_STRICT_RANDOM—Requests at the highest priority are granted randomly.

UVM_SEQ_ARB_USER—Arbitration is delegated to the user-defined function, **user_priority_arbitration** (see 15.3.2.3), which specifies the next sequence to grant.

F.2.4.2 uvm_sequence_state_enum

Defines the current sequence state. These enumeration values shall be one-hot to support bitwise ORing, such as that used by **uvm_sequence_base::wait_for_sequence_state** (see 14.2.2.5).

UVM_CREATED—The sequence has been allocated.

UVM_PRE_START—The sequence is started and the **uvm_sequence_base::pre_start** task (see 14.2.3.2) is being executed.

UVM_PRE_BODY—The sequence is started and the **uvm_sequence_base::pre_body** task (see 14.2.3.3) is being executed.

UVM_BODY—The sequence is started and the **uvm_sequence_base::body** task (see 14.2.3.6) is being executed.

UVM_ENDED—The sequence has completed the execution of the **uvm_sequence_base::body** task (see 14.2.3.6).

UVM_POST_BODY—The sequence is started and the **uvm_sequence_base::post_body** task (see 14.2.3.8) is being executed.

UVM_POST_START—The sequence is started and the **uvm_sequence_base::post_start** task (see 14.2.3.9) is being executed.

UVM_STOPPED—The sequence has been forcibly ended by issuing a **uvm_sequence_base::kill** (see 14.2.5.11) on the sequence.

UVM_FINISHED—The sequence is completely finished executing and was not forcibly ended.

F.2.4.3 uvm_sequence_lib_mode

Specifies the random selection mode of a sequence library; it has the following parameters:

UVM_SEQ_LIB_RAND—Random sequence selection.

UVM_SEQ_LIB_RANDC—Random cyclic sequence selection.

UVM_SEQ_LIB_ITEM—Emits only items, no sequence execution.

UVM_SEQ_LIB_USER—Applies a user-defined random-selection algorithm.

F.2.5 Phasing

F.2.5.1 uvm_phase_type

This is the set of possible objects of a **uvm_phase** object (see 9.3.1); it has the following parameters:

UVM_PHASE_IMP—This phase object is used to traverse the component hierarchy and call the component phase method as well as the **phase_started** (see 13.1.4.3.1) and **phase_ended** (see 13.1.4.3.3) callbacks.

UVM_PHASE_NODE—This object represents a simple node instance in the graph (see 9.3.1). These nodes contain a reference to their corresponding UVM_PHASE_IMP object.

UVM_PHASE_SCHEDULE–This object represents a portion of the phasing graph, typically consisting of several UVM_PHASE_NODE types, in series, in parallel, or both.

UVM_PHASE_DOMAIN—This object represents an entire graph segment that executes in parallel with the “run” phase. Domains may define any network of UVM_PHASE_NODEs and UVM_PHASE_SCHEDULEs. The built-in domain, *uvm*, consists of a single schedule of all the run-time phases, starting with pre_reset and ending with post_shutdown.

F.2.5.2 uvm_phase_state

Following is the set of possible states of a phase:

UVM_PHASE_UNINITIALIZED—The state is uninitialized. This is the default state for phases and for nodes that have not yet been added to a schedule.

UVM_PHASE_DORMANT—The phase is part of a schedule, but not currently executing. It could be scheduled at some point in the future when its predecessors are done.

UVM_PHASE_SCHEDULED—The immediate predecessors of the phase are all done.

UVM_PHASE_SYNCING—All predecessors complete; waiting for all synced phases (e.g., across domains) to be at or beyond this point.

UVM_PHASE_STARTED—Any synced phases are at UVM_PHASE_SYNCING or beyond; call the **phase_started** callback (see 13.1.4.3.1) for each component in the domain, then wait a delta cycle.

UVM_PHASE_EXECUTING—Past the delta cycle after calling **phase_started** (see 13.1.4.3.1); executing phase behavior until terminated by all objections being dropped, a jump, or a timeout.

UVM_PHASE_READY_TO_END—No objections remain in this phase or in any predecessors of its successors or in any synched phases. In this state, if an objection is raised to this phase, the state returns to UVM_PHASE_EXECUTING. If no objection is raised before a delta cycle elapses, the state transitions to UVM_PHASE_ENDED.

UVM_PHASE_ENDED—The phase has completed execution; it is now running the **phase_ended** callback (see 13.1.4.3.3). Completes in a delta cycle.

UVM_PHASE_CLEANUP—(No jump is in progress.) All processes related to phase are being killed. Completes in a delta cycle.

UVM_PHASE_JUMPING—(A jump is in progress.) All processes related to phase are being killed. Completes in a delta cycle. All predecessors are forced into the UVM_PHASE_DONE state and the phase target is forced to UVM_DORMANT state.

UVM_PHASE_DONE—The phase has finished execution, which may enable a waiting successor phase to execute.

F.2.5.3 uvm_wait_op

Specifies the operand when using methods like **uvm_phase::wait_for_state** (see 9.3.1.8.3); it can be one of the following:

UVM_EQ—Equal.

UVM_NE—Not equal.

UVM_LT—Less than.

UVM_LTE—Less than or equal to.

UVM_GT—Greater than.

UVM_GTE—Greater than or equal to.

F.2.6 Objections

uvm_objection_event

Enumerates the possible objection events on which one could wait; it has the following parameters. See also 10.5.1.5.2.

UVM_RAISED—An objection was raised.

`UVM_DROPPED`—An objection was dropped.

`UVM_ALL_DROPPED`—All objections have been dropped.

F.2.7 uvm_apprepend

Specifies whether order-dependent API should put new items at the front or at the back, as follows:

`UVM_APPEND`—Put new items at the back.

`UVM_PREPEND`—Put new items at the front.

F.2.8 UVM_FILE

A type that can represent a SystemVerilog file descriptor or multichannel descriptor.

F.2.9 UVM_STDIN, UVM_STDOUT, and UVM_STDERR

```
parameter UVM_FILE UVM_STDIN = 32'h8000_0000
parameter UVM_FILE UVM_STDOUT = 32'h8000_0001
parameter UVM_FILE UVM_STDERR = 32'h8000_0002
```

The **UVM_STDIN**, **UVM_STDOUT**, and **UVM_STDERR** values align to the SystemVerilog file descriptor values for `STDIN`, `STDOUT`, and `STDERR`, respectively.

F.2.10 uvm_core_state

This is an enumeration containing the set of possible states for the UVM core; it has the following values:

`UVM_CORE_UNINITIALIZED`—The core has yet to be initialized.

`UVM_CORE_INITIALIZED`—The core has been initialized via **uvm_init** (see F.3.1.3).

`UVM_CORE_PRE_RUN`—**uvm_root::run_test** (see F.7.2.1) has been called, but the **pre_run_test** callbacks (see F.6.2.2) have yet to complete.

`UVM_CORE_RUNNING`—The **pre_run_test** callbacks have completed, and the core is now phasing all components.

`UVM_CORE_POST_RUN`—The core has completed phasing all components, but the **post_run_test** callbacks (see F.6.2.3) have yet to complete.

`UVM_CORE_FINISHED`—All **post_run_test** callbacks have completed, the core considers the test finished.

`UVM_CORE_PRE_ABORT`—The **die** method (see F.7.2.2) method has been called, but the **uvm_component::pre_abort** hooks (see 13.1.4.6), **uvm_run_test_callback::pre_abort** hooks (see F.6.2.4), and **report_summarize** method (see 6.5.1.16) have yet to complete.

`UVM_CORE_ABORTED`—The **die** method has been called, and the **pre_abort** and **post_run_test** callbacks have completed. `$finish` is about to be called.

F.3 Methods and types

F.3.1 Simulation control

F.3.1.1 get_core_state

```
function uvm_core_state get_core_state()
```

Returns the current status of the UVM core.

F.3.1.2 run_test

```
task run_test (
    string test_name = ""
)
```

This is a convenience function for **uvm_root::run_test** (see F.7.2.1).

F.3.1.3 uvm_init

```
function void uvm_init (
    uvm_coreservice_t cs = null
)
```

Initializes the UVM framework and sets the **uvm_coreservice_t** instance (see F.4) returned by **uvm_coreservice_t::get** (see F.4.1.3). If *cs* is *null*, then **uvm_coreservice_t::get** returns an implementation defined core service instance; otherwise, the provided *cs* is returned.

Additionally, the initialize hook on all registered **uvm_object_registry#(T,Tname)** (see 8.2.4) and **uvm_component_registry#(T,Tname)** (see 8.2.3) types shall be called, after which the UVM core state (see F.3.1.1) is set to `UVM_CORE_INITIALIZED`. Only the first call to **uvm_init** shall initialize the UVM framework. Subsequent calls to **uvm_init** are silently ignored.

F.3.2 Reporting

F.3.2.1 uvm_get_report_object

```
function uvm_report_object uvm_get_report_object()
```

Returns **uvm_root** (see F.7) to act as the **uvm_report_object** (see 6.3).

F.3.2.2 uvm_report_enabled

```
function bit uvm_report_enabled (
    int verbosity,
    uvm_severity severity = UVM_INFO,
    string id = ""
)
```

Returns `1` if the configured verbosity in the implicit top-level component (see F.7) for this severity/id is greater than or equal to verbosity, else returns `0`. See also 6.3.3.2.

The default value of *severity* shall be `UVM_INFO`.

F.3.2.3 uvm_report, uvm_report_info, uvm_report_warning, uvm_report_error, and uvm_report_fatal

```
function void uvm_report(
   uvm_severity severity,
   string id,
   string message,
   int verbosity = (severity ==
                    uvm_severity'(UVM_ERROR)) ?
                    UVM_NONE : (severity ==
                    uvm_severity'(UVM_FATAL)) ?
                    UVM_NONE : UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

function void uvm_report_info(
   string id,
   string message,
   int verbosity = UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

function void uvm_report_warning(
   string id,
   string message,
   int verbosity = UVM_MEDIUM,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

function void uvm_report_error(
   string id,
   string message,
   int verbosity = UVM_NONE,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)

function void uvm_report_fatal(
   string id,
   string message,
   int verbosity = UVM_NONE,
   string filename = "",
   int line = 0,
   string context_name = "",
   bit report_enabled_checked = 0
)
```

```
function void uvm_process_report_message(
    uvm_report_message report_message
)
```

These methods, defined in the package scope, delegate to the corresponding component methods in the implicit top-level component (see F.7). They can be used in module-based code to use the same reporting mechanism as class-based components. See 6.3 for details on the reporting mechanism.

F.3.3 Miscellaneous

F.3.3.1 uvm_is_match

```
function bit uvm_is_match (
    string expr,
    string str
)
```

Returns `1` if the two strings match, `0` otherwise.

The *expr* can be made to match multiple *str* by using either POSIX regular expression notation (IEEE Std 1003.1-2008™ [B1]) or a simplified notation.

Each resource within the pool has a set of scopes over which it is visible. When attempting to retrieve resources from the pool via the **lookup** interface (see C.2.4.4), the scope of the lookup is compared against any stored scopes set via **set_scope** (see C.2.4.3.1), **set_override** (see F.3.3.2), **set_name_override** (see C.2.4.3.3), or **set_type_override** (see C.2.4.3.4).

A single resource can be made to match multiple lookup scopes by using either POSIX regular expression notation (IEEE Std 1003.1-2008™ [B1]) or a simplified notation when setting its scope within the pool.

Regular expressions are identified as such when they are surrounded by '`/`' characters. The enclosing '`/`' characters are not part of the actual regular expression. For example, the scope `/^top\.*/` is interpreted as the regular expression `^top\.*`. Any expressions not surrounded by '`/`' shall be treated as simplified notation.

The simplified notation has only three meta-characters: `*`, `+`, and `?`. Table F.1 shows the regular expression equivalents of the simplified notation's meta-characters.

Table F.1—Simplified notation meta-character equivalents

Character	Meaning	Regular expression equivalent
*	Zero or more characters	.*
+	One or more characters	.+
?	Exactly one character	.

F.3.3.2 uvm_split_string

```
function automatic void uvm_split_string (
       string str,
       byte sep,
    ref string values[$]
)
```

Returns a queue of strings, *values*, that is the result of the *str* split based on the *sep*. *values* shall be a queue.

F.3.4 uvm_enum_wrapper#(T)

The **uvm_enum_wrapper#(T)** class is a utility mechanism provided as a convenience to the user. It provides a **from_name** method (see F.3.4.2), which is the logical inverse of the SystemVerilog `name` method that is built into all enumerations.

F.3.4.1 Class declaration

```
class uvm_enum_wrapper#(
    type T = uvm_active_passive_enum
)
```

T shall be an enumerated type.

F.3.4.2 Methods

from_name

```
static function bit from_name(
       string name,
    ref T value
)
```

This attempts to convert a string name to an enumerated value.

If the conversion is successful, this method return `1`, otherwise `0`.

The *name* passed in to the method is case sensitive and needs to exactly match the value that would be produced by `enum::name`.

F.4 Core service

The UVM core service provides a common point for all central UVM services such as **uvm_factory** (see 8.3.1), **uvm_report_server** (see 6.5.1), etc. The service class provides a static **::get** (see F.4.1.3), which returns an instance adhering to **uvm_coreservice_t**.

The rest of the **set**_*facility* **get**_*facility* pairs provide access to internal UVM services.

F.4.1 uvm_coreservice_t

F.4.1.1 Class declaration

```
virtual class uvm_coreservice_t
```

F.4.1.2 Constructor

```
function new
```

Constructor for the **uvm_coreservice_t** type. This constructor takes no arguments.

F.4.1.3 get

```
static function uvm_coreservice_t get()
```

Returns the **uvm_coreservice_t** instance, as defined by **uvm_init** (see F.3.1.3). If **get_core_state** (see F.3.1.1) returns `UVM_CORE_UNINITIALIZED`, then **get** shall call **uvm_init** with *cs* set to *null* and return the implementation defined core service instance (see F.4.2).

F.4.1.4 Methods for core service sub-typing

F.4.1.4.1 get_root

```
pure virtual function uvm_root get_root()
```

Intended to return the **uvm_root** instance (see F.7).

F.4.1.4.2 get_factory

```
pure virtual function uvm_factory get_factory()
```

Intended to return the currently enabled UVM factory.

F.4.1.4.3 set_factory

```
pure virtual function void set_factory(
   uvm_factory f
)
```

Intended to set the current UVM factory.

F.4.1.4.4 get_report_server

```
pure virtual function uvm_report_server get_report_server()
```

Intended to return the current global `report_server`.

F.4.1.4.5 set_report_server

```
pure virtual function void set_report_server(
   uvm_report_server server
)
```

Intended to set the central report server to *server*.

F.4.1.4.6 get_default_tr_database

```
pure virtual function uvm_tr_database get_default_tr_database()
```

Intended to return the current default record database.

F.4.1.4.7 set_default_tr_database

```
pure virtual function void set_default_tr_database(
   uvm_tr_database db
)
```

Intended to set the current default record database to *db*.

F.4.1.4.8 get_component_visitor

```
pure virtual function uvm_visitor#(
   uvm_component
) get_component_visitor()
```

Intended to retrieve the current component visitor. See also F.4.1.4.9.

F.4.1.4.9 set_component_visitor

```
pure virtual function void set_component_visitor(
   uvm_visitor#(uvm_component) v
)
```

Intended to set the component visitor to *v* (this visitor is being used for the traversal at `end_of_elaboration_phase`, e.g., for name checking).

F.4.1.4.10 set_phase_max_ready_to_end

```
pure virtual function void set_phase_max_ready_to_end(int max)
```

Intended to set the default value for maximum iterations of `ready_to_end` for all phases (see 9.3.1.3.5).

F.4.1.4.11 get_phase_max_ready_to_end

```
pure virtual function int get_phase_max_ready_to_end()
```

Intended to get the default value for maximum iterations of `ready_to_end` for all phases.

F.4.1.4.12 set_default_printer

```
pure virtual function void set_default_printer (uvm_printer printer)
```

Sets the default printer policy instance.

F.4.1.4.13 get_default_printer

```
pure virtual function uvm_printer get_default_printer()
```

Retrieves the default printer policy instance, as defined by **set_default_printer** (see F.4.1.4.12). If **set_default_printer** has not been called or has been called with a value of *null*, the implementation returns the implementation's default printer instance.

F.4.1.4.14 set_default_packer

```
pure virtual function void set_default_packer (uvm_packer packer)
```

Sets the default packer policy instance.

F.4.1.4.15 get_default_packer

```
pure virtual function uvm_packer get_default_packer()
```

Retrieves the default packer policy instance, as defined by **set_default_packer** (see F.4.1.4.14). If **set_default_packer** has not been called or has been called with a value of *null*, the implementation returns the implementation's default packer instance.

F.4.1.4.16 set_default_comparer

```
pure virtual function void set_default_comparer (uvm_comparer comparer)
```

Sets the default comparer policy instance.

F.4.1.4.17 get_default_comparer

```
pure virtual function uvm_comparer get_default_comparer()
```

Retrieves the default comparer policy instance, as defined by **set_default_comparer** (see F.4.1.4.16). If **set_default_comparer** has not been called or has been called with a value of *null*, the implementation returns the implementation's default comparer instance.

F.4.1.4.18 set_default_copier

```
pure virtual function void set_default_copier(
    uvm_copier copier
)
```

Sets the default copier policy instance.

F.4.1.4.19 get_default_copier

```
pure virtual function uvm_copier get_default_copier()
```

Retrieves the default copier policy instance, as defined by **set_default_copier** (see F.4.1.4.18). If **set_default_copier** has not been called or has been called with a value of *null*, the implementation shall return the implementation's default copier instance.

F.4.1.4.20 get_global_seed

```
pure virtual function int unsigned get_global_seed()
```

Returns a seed that shall be used by the UVM base class library to initialize the random number generators of objects and/or processes.

The return value is implementation specific, however the return value shall be immutable. Successive calls to **get_global_seed** within a single simulation shall return the same value.

The mechanism for generating seeds based off of the return value of **get_global_seed** is implementation specific. The seeds generated within the UVM base class library shall be deterministic, such that if all interactions with the UVM base class library between two simulations are identical, including the global seed value, then the seeds generated within the UVM base class library shall be identical.

Refer to **use_uvm_seeding** (see 5.3.3.1) and random stability (see 1.3.6) for additional information regarding random stability within the UVM base class library.

F.4.1.4.21 set_resource_pool

```
pure virtual function void set_resource_pool (
   uvm_resource_pool pool
)
```

Intended to set the global resource pool instance to `pool`.

F.4.1.4.22 get_resource_pool

```
pure virtual function uvm_resource_pool get_resource_pool()
```

Intended to return the global resource pool instance.

F.4.1.4.23 set_resource_pool_default_precedence

```
pure virtual function void set_resource_pool_default_precedence(
   int unsigned precedence
)
```

Overrides the current default precedence being used by the resource pool.

Calling this method only changes the value used for future calls to **get_default_precedence** (see C.2.4.5.5). The precedence of any resources already stored within the pool remains unchanged.

F.4.1.4.24 get_resource_pool_default_precedence

```
pure virtual function int unsigned get_resource_pool_default_precedence()
```

Returns the current default precedence being used by the resource pool.

This value shall be `1000`, unless overwritten via a call to **set_default_precedence** (see C.2.4.5.4).

F.4.2 uvm_default_coreservice_t

Default implementation of the UVM core service. **uvm_default_coreservice_t** can be extended; it provides a full implementation of all **uvm_coreservice_t** methods (see F.4.1.4).

F.4.2.1 Class declaration

```
class uvm_default_coreservice_t extends uvm_coreservice_t
```

F.4.2.2 Constructor

```
function new
```

Constructor for the **uvm_default_coreservice_t** type. This constructor takes no arguments.

F.4.3 get_uvm_seeding

```
virtual function bit get_uvm_seeding()
```

Returns the current UVM seeding *enable* value, as set by **set_uvm_seeding** (see F.4.4). If **set_uvm_seeding** has not been called, this returns `1`.

F.4.4 set_uvm_seeding

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

Sets the UVM seeding *enable* value.

This bit enables or disables the UVM seeding mechanism. It globally affects the operation of the **reseed** method (see 5.3.3.3). The default value shall be `1` (enabled).

When enabled, UVM-based objects are seeded based on their type and full hierarchical name rather than allocation order. This improves random stability for objects whose instance names are unique across each type. The **uvm_component** class (see 13.1) is an example of a type that has a unique instance name.

F.5 Traversal

F.5.1 uvm_visitor #(NODE)

The **uvm_visitor** class provides an abstract base class for a visitor. The visitor visits instances of type `NODE`.

F.5.1.1 Class declaration

```
virtual class uvm_visitor#(type NODE=uvm_component) extends uvm_object
```

F.5.1.2 Methods

F.5.1.2.1 begin_v

```
virtual function void begin_v()
```

This method is invoked by the visitor before the first `NODE` is visited. This base version does nothing.

F.5.1.2.2 end_v

```
virtual function void end_v()
```

This method is invoked by the visitor after the last `NODE` is visited. This base version does nothing.

F.5.1.2.3 visit

```
pure virtual function void visit(
    NODE node
)
```

Intended to be invoked by the visitor for every visited node of the provided structure. The user needs to provide the functionality in this function.

F.5.2 uvm_structure_proxy #(STRUCTURE)

The **uvm_structure_proxy** is a wrapper that provides a set of elements of the `STRUCTURE` to the caller on demand. This is to decouple the retrieval of the **STRUCTURE**'s subelements from the actual function being invoked on `STRUCTURE`.

F.5.2.1 Class declaration

```
virtual class uvm_structure_proxy#(
   type STRUCTURE = uvm_component
   ) extends uvm_object
```

F.5.2.2 Methods

F.5.2.2.1 new

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

Initializes a new **uvm_structure_proxy** with the specified *name*. The default value of *name* shall be an *empty string* (**""**).

F.5.2.2.2 get_immediate_children

```
pure virtual function void get_immediate_children(
       STRUCTURE s,
   ref STRUCTURE children[$]
)
```

Intended to return a set of the direct subelements of *s* within *children*. *children* shall be a queue.

F.5.3 uvm_visitor_adapter #(STRUCTURE,uvm_visitor#(STRUCTURE))

This visitor adapter traverses all nodes of the `STRUCTURE` and invokes *visitor.visit* on every node.

F.5.3.1 Class declaration

```
class uvm_visitor_adapter#(
   type STRUCTURE, VISITOR
) extends uvm_object
```

The type of *VISITOR* shall be derived from **uvm_visitor#(*STRUCTURE*)**.

F.5.3.2 Methods

F.5.3.2.1 new

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

Initializes a new **uvm_visitor_adapter** with the specified *name*. The default value of *name* shall be an *empty string* (**""**).

F.5.3.2.2 accept

```
pure virtual function void accept(
   STRUCTURE s,
   VISITOR v,
   uvm_structure_proxy#(STRUCTURE) p,
   bit invoke_begin_end = 1
)
```

Intended to traverse through *s* (and every subnode of *s*), invoking *v.visit(node)* for each node found. The children of *s* are intended to be determined by invoking *p.get_immediate_children*. *invoke_begin_end* determines whether the visitors begin/end functions should be invoked prior to traversal. The default value of *invoke_begin_end* shall be `1`.

F.5.4 uvm_top_down_visitor_adapter

This adapter traverses the `STRUCTURE` *s* (and invokes the visitor) in a hierarchical fashion. During traversal, *s* is visited before any subnodes of *s* are visited.

F.5.4.1 Class declaration

```
class uvm_top_down_visitor_adapter#(
    type STRUCTURE = uvm_component,
           VISITOR = uvm_visitor#(STRUCTURE)
) extends uvm_visitor_adapter#(STRUCTURE,VISITOR)
```

F.5.4.2 Methods

new

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

Initializes a new **uvm_top_down_visitor_adapter** with the specified *name*. The default value of *name* shall be an *empty string* (`""`).

F.5.5 uvm_bottom_up_visitor_adapter

This adapter traverses the `STRUCTURE` *s* (and invokes the visitor) in a hierarchical fashion. During traversal, all the children of *s* are visited before *s* is visited.

F.5.5.1 Class declaration

```
class uvm_bottom_up_visitor_adapter#(
    type STRUCTURE = uvm_component,
           VISITOR = uvm_visitor#(STRUCTURE)
) extends uvm_visitor_adapter#(STRUCTURE,VISITOR)
```

F.5.5.2 Methods

new

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

Initializes a new **uvm_bottom_up_visitor_adapter** with the specified *name*. The default value of *name* shall be an *empty string* (`""`).

F.5.6 uvm_by_level_visitor_adapter

This adapter traverses the `STRUCTURE` *s* (and invokes the visitor) in a hierarchical fashion. During traversal, all the direct children of *s* are visited before any grandchildren are visited.

F.5.6.1 Class declaration

```
class uvm_by_level_visitor_adapter#(
    type STRUCTURE = uvm_component,
```

```
          VISITOR = uvm_visitor#(STRUCTURE)
) extends uvm_visitor_adapter#(STRUCTURE,VISITOR)
```

F.5.6.2 Methods

new

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

Initializes a new **uvm_by_level_visitor_adapter** with the specified *name*. The default value of *name* shall be an *empty string* (`""`).

F.5.7 uvm_component_proxy

The class provides an implementation of **uvm_structure_proxy::get_immediate_children** that returns the subcomponents of the component instance passed into it.

F.5.7.1 Class declaration

```
class uvm_component_proxy extends uvm_structure_proxy#(
   uvm_component
)
```

F.5.7.2 Methods

new

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

Initializes a new **uvm_component_proxy** with the specified *name*. The default value of *name* shall be an *empty string* (`""`).

F.6 uvm_run_test_callback

Callback used for notification of the beginning and ending of a UVM test.

F.6.1 Class declaration

```
virtual class uvm_run_test_callback extends uvm_callback
```

F.6.2 Methods

F.6.2.1 new

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

Creates a new **uvm_run_test_callback** with the given instance *name*. The default value of *name* is `uvm_run_test_callback`.

F.6.2.2 pre_run_test

```
virtual function void pre_run_test()
```

The **pre_run_test** method is called on all registered **uvm_run_test_callback** instances upon entry to **uvm_root::run_test** (see F.7.2.1).

F.6.2.3 post_run_test

```
virtual function void post_run_test()
```

The **post_run_test** method is called on all registered **uvm_run_test_callback** instances immediately prior to the **uvm_root::run_test** (see F.7.2.1) returning or calling `$finish`.

F.6.2.4 pre_abort

```
virtual function void pre_abort()
```

The **pre_abort** method is called on all registered **uvm_run_test_callback** instances prior to **uvm_root::die** (see F.7.2.2) terminating the simulation.

F.6.2.5 add

```
static function bit add(
    uvm_run_test_callback cb
)
```

Adds *cb* to the list of callbacks to be processed. The method returns `1` if *cb* is not already in the list of callbacks; otherwise, a `0` is returned. If *cb* is *null*, `0` is returned.

F.6.2.6 delete

```
static function bit delete(
    uvm_run_test_callback cb
)
```

Removes *cb* from the list of callbacks to be processed. The method returns `1` if *cb* is in the list of callbacks; otherwise, a `0` is returned. If *cb* is *null*, `0` is returned.

F.7 uvm_root

The **uvm_root** class serves as the implicit top-level and phase controller for all UVM components. Users do not directly instantiate **uvm_root**. UVM automatically creates a singleton of **uvm_root** that users can access via `uvm_root::get`.

The root instance of **uvm_root** plays several key roles in UVM.

a) *Implicit top-level*—The root instance serves as an implicit top-level component. Any component whose parent is specified as `NULL` becomes a child of this instance. Thus, all UVM components in simulation are descendants of this instance.
b) *Phase control*—The root instance is responsible for executing the test (see F.7.2.1).
c) *Search*—Use the root instance to search for components based on their hierarchical name. See F.7.3.1.
d) *Report configuration*—Use the root instance to globally configure report verbosity, logfiles, and actions.
e) *Global reporter*—Because the root instance is accessible via the `uvm_pkg` scope, the reporting mechanism can be used from anywhere. See F.3.2.3.

The root instance checks during the `end_of_elaboration` phase if any errors have been generated so far. If errors are found, an `UVM_FATAL` error shall be generated so the simulation does not continue to the **uvm_start_of_simulation_phase** (see 9.8.1.4).

F.7.1 Common methods

F.7.1.1 new

```
protected function new()
```

Constructor.

As root is a singleton at the top of the component hierarchy, it does not support creation via the **uvm_factory** (see 8.3.1). A fatal message shall be generated if multiple instances or extensions of **uvm_root** are constructed.

F.7.1.2 get

```
static function uvm_root get()
```

Static accessor for **uvm_root**.

The static accessor is provided as a convenience wrapper around retrieving the root via the **uvm_coreservice_t::get_root** method (see F.5).

F.7.2 Simulation control

F.7.2.1 run_test

```
virtual task run_test (
    string test_name = ""
)
```

Executes the test. The following operations are performed in order:

a) The **uvm_coreservice_t** instance (see F.4) is retrieved via **uvm_coreservice_t::get** (see F.4.1.3).

b) The UVM core state (see F.3.1.1) is set to `UVM_CORE_PRE_RUN`.

c) The **pre_run_test** (see F.6.2.2) method is called on all registered **uvm_run_test_callback** instances (see F.5.7.2).

d) If the command-line plusarg, `+UVM_TESTNAME=<TEST_NAME>` (see G.2.1), is found, then an implementation shall call **create_component_by_name** (see 8.3.1.5) on the current factory (see F.4.1.4.2) with *requested_type_name* set to the plusarg defined *<TEST_NAME>* and *name* set to `uvm_test_top`.

e) If *test_name* is not an *empty string* (`""`) and no name was provided via the command-line plusarg, then an implementation calls **create_component_by_name** on the current factory with *requested_type_name* set to *test_name* and *name* set to `uvm_test_top`.

f) If no components other than **uvm_root** (see F.7) have been created at this point, either by **run_test** or by the user, then an implementation shall generate a fatal message and **run_test** shall return immediately.

g) The UVM core state is set to `UVM_CORE_RUNNING`.

h) All components are phased through all registered phases (see Clause 9).

i) The UVM core state is set to `UVM_CORE_POST_RUN`.

j) The **post_run_test** method (see F.6.2.3) is called on all registered **uvm_run_test_callback** instances.

k) The **report_summarize** method (see 6.5.1.16) is called on the current report server (see F.4.1.4.4).

l) The UVM core state is set to `UVM_CORE_FINISHED`.

m) If **get_finish_on_completion** (see F.7.2.5) returns `1`, then `$finish` is called; otherwise, **run_test** shall return.

F.7.2.2 die

```
virtual function void die()
```

This method is called by the report server if a report reaches the maximum quit count or has an `UVM_EXIT` action associated with it, e.g., as with fatal errors.

The following operations are performed in order:

a) The UVM core state (see F.3.1.1) is set to `UVM_CORE_PRE_ABORT` and the previous core state value is temporarily stored for use in c).

b) The **uvm_component::pre_abort** method (see 13.1.4.6) is called on the entire **uvm_component** hierarchy (see 13.1) in a bottom-up fashion.

c) The **pre_abort** method (see F.6.2.4) is called on all registered **uvm_run_test_callback** instances (see F.5.7.2).

d) The **report_summarize** method (see 6.5.1.16) is called on the current report server (see F.4.1.4.4).

e) The UVM core state is set to `UVM_CORE_ABORTED`.

f) The simulation is terminated with `$finish`.

F.7.2.3 set_timeout

```
function void set_timeout(
    time timeout,
    bit overridable = 1
)
```

Specifies the global timeout for the simulation, which is applied in **uvm_run_phase** (see 9.8.1.5). If this method is not called, the default timeout value is implementation specific.

The *timeout* is simply the maximum absolute simulation time allowed before a `UVM_FATAL` timeout occurs. If *timeout* is set to `20ns`, the simulation needs to end before 20 ns or a `UVM_FATAL` timeout will occur. This feature is provided so a user can prevent the simulation from potentially consuming too many resources (disk, memory, CPU, etc.) when the testbench is essentially hung.

If *overridable* is `0`, any future calls to **set_timeout** have no effect. If *overridable* is `1`, this call puts no such restrictions on any future calls. The default value of *overridable* shall be `1`.

F.7.2.4 set_finish_on_completion

```
virtual function void set_finish_on_completion(bit f)
```

If this function has never been called or this function is passed a `1`, **run_test** (see F.7.2.1) calls `$finish` after all phases are executed. When this function is passed a `0`, the user needs to implement a mechanism to terminate the simulation.

F.7.2.5 get_finish_on_completion

```
virtual function bit get_finish_on_completion()
```

Returns the latest value set by **set_finish_on_completion** (see F.7.2.4) or `1` if **set_finish_on_completion** has never been called.

F.7.2.6 end_of_elaboration_phase

```
virtual function void end_of_elaboration_phase(
uvm_phase phase
)
```

Extension of **uvm_component::end_of_elaboration_phase** (see 13.1.4.1.3). The root instance checks if any errors have been generated so far in the default report server (see 6.5.2) using **get_severity_count** (see 6.5.1.5). If the count is greater than `0`, a `UVM_FATAL` message is generated so the simulation does not continue to the **uvm_start_of_simulation_phase** (see 9.8.1.4).

F.7.3 Topology

F.7.3.1 find and find_all

```
function uvm_component find (
   string comp_match
)


function void find_all (
          string comp_match,
      ref uvm_component comps[$],
   input uvm_component comp = null
)
```

Returns the component (**find**) or list of components (**find_all**) matching a given *comp_match* string. Matches are determined using **uvm_is_match** (see F.3.3.1), with *comp_match* as *expr*, and the component's full name (see 13.1.3.2) as *str*.

If the *comp* argument is not *null*, the search begins from that component down; otherwise, all component instances are compared.

find does a **find_all** with `comp = null` and returns the first element in the output queue or *null* if there is an empty queue. Any elements in the *comps* queue prior to the call to **find_all** are unaffected by **find_all** and will still be present in the *comps* queue after **find_all**.

F.7.3.2 print_topology

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

Prints the verification environment's component topology. The *printer* argument provides the policy class to be used for this operation. If no printer is provided (or the value provided is `null`), **print_topology** shall use the default printer policy, as returned by **get_default_printer** (see F.4.1.4.13).

Annex G

(normative)

Command line arguments

G.1 Command line processing

The **uvm_cmdline_processor** class provides an interface to the command line arguments that were provided for the given simulation. Users can retrieve the complete arguments using methods such as **get_args** and **get_arg_matches** and also retrieve the suffixes of arguments using **get_arg_values**.

The generation of the data structures that hold the command line argument information happens during construction of the class object. A global variable called `uvm_cmdline_proc` is created at initialization time and may be used to access command line information. Command line arguments that are in uppercase should only have one setting per invocation. Command line arguments that are in lowercase can have multiple settings per invocation.

The **uvm_cmdline_processor** class also provides support for setting various UVM variables from the command line, such as components' verbosities and configuration settings for integral types and strings. Each of these capabilities is described in G.2.

G.1.1 Class declaration

```
class uvm_cmdline_processor extends uvm_report_object
```

G.1.2 Singleton

get_inst

```
static function uvm_cmdline_processor get_inst()
```

This is a convenience mechanism, it returns the singleton instance of the UVM command line processor.

G.1.3 Basic arguments

G.1.3.1 get_args

```
function void get_args (
   output string args[$]
)
```

This function returns a queue with all of the command line arguments that were used to start the simulation. Element `0` of the array is always the name of the executable that started the simulation. *args* shall be a queue.

G.1.3.2 get_plusargs

```
function void get_plusargs (
   output string args[$]
)
```

This function returns a queue with all of the plus arguments that were used to start the simulation. Plus arguments may be used by the simulator, or may be specific to a company or individual user. `plusargs` never use extra arguments (i.e., if there is a `plusarg` as the second argument on the command line, the third argument is unrelated); this is not necessarily the case with application specific dash arguments. *args* shall be a queue.

G.1.3.3 get_uvmargs

```
function void get_uvm_args (
         output string args[$]
)
```

This function loads the queue *args* with all of the *uvm* arguments that were used to start the simulation. A *uvm* argument is taken to be any argument that starts with a - or + and uses the keyword `UVM` (case insensitive) as the first three letters of the argument.

G.1.3.4 get_arg_matches

```
function int get_arg_matches (
       string match,
    ref string args[$]
)
```

This function replaces any contents of the queue *args* with all of the arguments that match the expression in *match*, and it returns the number of items that matched. If *match* is bracketed with `//`, it is taken as an extended regular expression; otherwise, it is taken as the beginning of an argument to match. For example:

```
string myargs[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_matches("+foo",myargs))
      //matches +foo, +foobar
      //doesn't match +barfoo
   void'(uvm_cmdline_proc.get_arg_matches("/foo/",myargs))
      //matches +foo, +foobar,
      //foo.sv, barfoo, etc.
   void'(uvm_cmdline_proc.get_arg_matches("/^foo.*\.sv",myargs))
      //matches foo.sv
      //and foo123.sv,
      //not barfoo.sv.
```

G.1.4 Argument values

G.1.4.1 get_arg_value

```
function int get_arg_value (
       string match,
    ref string value
)
```

This function finds the first argument that matches *match* and assigns the suffix of the argument to *value*. It then returns the number of command line arguments that match *match*. *match* is interpreted as described in G.1.3.4.

G.1.4.2 get_arg_values

```
function int get_arg_values (
       string match,
   ref string args[$]
)
```

This function finds all arguments that match *match* and replaces any contents of the *args* queue with the suffixes of the matching arguments. It then returns the number of command line arguments that match *match*. *match* is interpreted as described in G.1.3.4. For example, if '`+foo=1,yes,on +foo=5,no,off`' was provided on the command line and the following code was executed:

```
string foo_values[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_values("+foo=",foo_values))
```

the `foo_values` queue would contain two entries:

0"1,yes,on"
1"5,no,off"

G.2 Built-in UVM-aware command line arguments

G.2.1 +UVM_TESTNAME

`+UVM_TESTNAME=<`*`class name`*`>` can be used to specify which **uvm_test** (or **uvm_component**) should be created via the factory and cycled through the UVM phases. If multiple settings occur, the first occurrence is used and a warning is issued for subsequent settings. For example:

```
<sim command> +UVM_TESTNAME=read_modify_write_test
```

G.2.2 +UVM_VERBOSITY

`+UVM_VERBOSITY=<`*`verbosity`*`>` can be used to specify the initial verbosity for all components. If multiple settings occur, the first occurrence is used and a warning is issued for subsequent settings. For example:

```
<sim command> +UVM_VERBOSITY=UVM_HIGH
```

G.2.3 +uvm_set_verbosity

`+uvm_set_verbosity=<`*`comp`*`>,<`*`id`*`>,<`*`verbosity`*`>,<`*`phase`*`>` and `+uvm_set_verbosity=<`*`comp`*`>,<`*`id`*`>,<`*`verbosity`*`>,time,<`*`time`*`>` can be used to manipulate the verbosity of specific components at specific phases (and times during the "run" phases) of the simulation. The *id* argument can be either **ALL** for all IDs or a specific message id. Wild carding is not supported for *id* due to performance concerns. Settings for non-"run" phases are executed in order of occurrence on the command line. Settings for "run" phases (times) are sorted by time and then executed in order of occurrence for settings of the same time. For example:

```
<sim command>
+uvm_set_verbosity=uvm_test_top.env0.agent1.*,_ALL_,UVM_FULL,time,800
```

G.2.4 +uvm_set_action

`+uvm_set_action=<comp>,<id>,<severity>,<action>` provides the equivalent of various **uvm_report_object**'s **set_report_*_action** APIs (see 6.3.5.2). The special keyword **ALL** can be provided for the *id* and/or *severity* arguments. The action can be `UVM_NO_ACTION` or a `|` separated list of the other UVM message actions. For example:

```
<sim command>
+uvm_set_action=uvm_test_top.env0.*,_ALL_,UVM_ERROR,UVM_NO_ACTION
```

G.2.5 +uvm_set_severity

`+uvm_set_severity=<comp>,<id>,<current severity>,<new severity>` provides the equivalent of the various **uvm_report_object**'s **set_report_*_severity_override** APIs (see 6.3.7). The special keyword **ALL** can be provided for the *id* and/or *current severity* arguments. For example:

```
<sim command>
+uvm_set_severity=uvm_test_top.env0.*,BAD_CRC,UVM_ERROR,UVM_WARNING
```

G.2.6 +UVM_MAX_QUIT_COUNT

`+UVM_MAX_QUIT_COUNT=<count>,<overridable>` changes the max quit count for the report server. The `<overridable>` argument ('`YES`' or '`NO`') specifies whether user code can subsequently change this value. If set to '`NO`' and the user code tries to change the max quit count value, a warning message is issued and the attempted change is ignored.

```
<sim command> +UVM_MAX_QUIT_COUNT=5,NO
```

G.2.7 +uvm_set_inst_override and +uvm_set_type_override

`+uvm_set_inst_override=<req_type>,<override_type>,<full_inst_path>` and `+uvm_set_type_override=<req_type>,<override_type>[,<replace>]` work like the name-based overrides in the **factory** (see 8.2.3.2): **factory.set_inst_override_by_name** and **factory.set_type_override_by_name**. For **uvm_set_type_override**, the third argument is `0` or `1` (the default is `1` if this argument is left off); this argument specifies whether previous type overrides for the type should be passed to the `replace` field of the factory calls. For example:

```
<sim command> +uvm_set_type_override=eth_packet,short_eth_packet
```

G.2.8 +uvm_set_config_int and +uvm_set_config_string

`+uvm_set_config_int=<comp>,<field>,<value>` and `+uvm_set_config_string=<comp>,<field>,<value>` work like their procedural counterparts: **set_config_int** and **set_config_string**. For the value of the settings, using `'b` (`0b`), `'o`, `'d`, and `'h` (`'x` or `0x`) as the first two characters of the value is treated as a base specifier for interpreting the base of the number. Size specifiers are not used since SystemVerilog does not allow size specifiers in string to value conversions. For example:

```
<sim command> +uvm_set_config_int=uvm_test_top.soc_env,mode,5
```

No equivalent of **set_config_object** exists as an **uvm_object** cannot be passed into the simulation via the command line.

G.2.9 +uvm_set_default_sequence

The `+uvm_set_default_sequence=<seqr>,<phase>,<type>` `plusarg` defines a default sequence from the command line, using the *typename* of that sequence. For example:

```
<sim command>
   +uvm_set_default_sequence=path.to.sequencer,main_phase,seq_type
```

This is functionally equivalent to calling the following in a test:

```
uvm_coreservice_t cs = uvm_coreservice_t::get()
uvm_factory f = cs.get_factory()
uvm_config_db#(uvm_object_wrapper)::set(this,
"path.to.sequencer.main_phase",
"default_sequence",
f.find_wrapper_by_name("seq_type"))
```

Consensus

WE BUILD IT.

Connect with us on:

Facebook: https://www.facebook.com/ieeesa

Twitter: @ieeesa

LinkedIn: http://www.linkedin.com/groups/IEEESA-Official-IEEE-Standards-Association-1791118

IEEE-SA Standards Insight blog: http://standardsinsight.com

YouTube: IEEE-SA Channel

IEEE
standards.ieee.org
Phone: +1 732 981 0060 Fax: +1 732 562 1571