Universal Verification Methodology for SystemC (UVM-SystemC)

Language Reference Manual

Accellera SystemC Verification Working Group

July 2020

Notices

Accellera Systems Initiative Standards documents are developed within Accellera Systems Initiative (Accellera) and its Technical Committee. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

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

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

The existence of an Accellera 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 an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

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

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate reasonable action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committee and Working Groups are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

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

Accellera Systems Initiative
8698 Elk Grove Blvd. Suite 1, #114
Elk Grove, CA 95624
USA

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

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera, provided that permission is obtained from and any required fee, if any, is paid to Accellera. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera. To arrange for authorization please contact Lynn Garibaldi, Executive Director, Accellera Systems Initiative, 8698 Elk Grove Blvd. Suite 1, #114, Elk Grove, CA 95624, phone (916) 760-1056, e-mail lynn@accellera.org.

Suggestions for improvements to the Universal Verification Methodology (UVM) Language Reference Manual for SystemC (UVM-SystemC) are welcome. They can be sent to the Accellera SystemC Verification discussion forum:

forums.accellera.org/forum/38-systemc-verification-uvm-systemc-scv

The current Accellera SystemC Verification Working Group web page is:

accellera.org/activities/working-groups/systemc-verification

Contributors

The development of the Universal Verification Methodology for SystemC (UVM-SystemC) Langauge Reference Manual was sponsored by the Accellera Systems Initiative and was created under the leadership of the following people:

Stephan Gerth, Bosch-Sensortec (*Chair*)
Bas Arts, NXP Semiconductors (*Vice-chair*)
Thilo Vörtler, COSEDA Technologies GmbH (*Technical Editor*)
Martin Barnasconi, NXP Semiconductors (*Technical Editor*)

Acknowledgements

The creation of this standard has been supported by the European Commission as part of the Seventh Framework Programme (FP7) for Research and Technological Development in the project 'Verification for heterogeneous Reliable Design and Integration' (VERDI). The research leading to this result has received funding from the European Commission under Grand agreement ID 287562.

More information on the Seventh Framework Programme (FP7) and VERDI project can be found here:

https://cordis.europa.eu/project/id/287562

The partners in the VERDI consortium wish to thank Cadence Design Systems Inc. for the initial contribution of the UVM-SC library reference and documentation (UVM version 1.0, June 2011). This document contains portions of this work, and has been extended to make it compatible with the UVM 1.2 standard.

Contents

1. Introduction

UVM-SystemC is a SystemC library extension offering features compatible with the Universal Verification Methodology (UVM). This library is built on top of the SystemC language standard and defines the Application Programming Interface aligned with the UVM standard defined in IEEE Std. 1800.2-2017[1,2]. The UVM-SystemC library does not cover the entire UVM standard, nor the existing UVM implementation in SystemVerilog. However, the UVM-SystemC library offers the essential ingredients to create verification environments which are compliant with the UVM standard.

UVM-SystemC is released as reference implementation that works with any IEEE Std. 1666-2011[3] compliant SystemC simulation environment. Note that UVM-SystemC uses certain specialized SystemC features introduced since the revision in 2011, such as process control constructs, which are not implemented in all SystemC simulators. The UVM-SystemC functionality can be used together with the Accellera Systems Initiative SystemC reference implementation[4].

UVM-SystemC uses existing SystemC functionality wherever suitable, and introduces new UVM classes on top of the SystemC base classes to facilitate the creation of modular, configurable and reusable verification environments. Certain UVM in SystemVerilog functionality is available as native SystemC language features, and therefore UVM-SystemC uses the existing SystemC classes as foundations for the UVM extensions. Also the transaction-level modeling (TLM) concepts natively exist in SystemC and IEEE Std. 1666-2011, so UVM-SystemC uses the original SystemC TLM definitions and base classes.

Elements which are part of the UVM-SystemC library and language definition and which are *not* part of the UVM-SystemVerilog standard are marked with the superscript section symbol §. Elements marked with the superscript degree symbol ° are renamed in UVM-SystemC, in contrast to the UVM-SystemVerilog standard, due to their incompatibility due to reserved keywords in C/C++ or an inappropriate name in the context of SystemC base class of member function definitions. The reference to the original UVM-SystemVerilog name is given in brackets and marked with the superscript dagger symbol †. Note that these original names are not defined in UVM-SystemC.

1 The IEEE standards or products referred to in this standard are trademarks of The Institute of Electrical and Electronics Engineers, Inc.
2 IEEE Standard for Universal Verification Methodology Language Reference Manual, https://standards.ieee.org/standard/1800_2-2017.html
3 IEEE Standard for Standard SystemC Language Reference Manual, https://standards.ieee.org/standard/1666-2011.html
4 Accellera Systems Initiative SystemC reference implementation version 2.3.0 or newer is required, https://accellera.org/downloads/standards/systemc

2. Terminology

2.1 Shall, should, may, can

The word *shall* is used to indicate a mandatory requirement.

The word *should* is used to recommend a particular course of action, but it does not impose any obligation.

The word *may* is used to mean shall be permitted (in the sense of being legally allowed).

The word *can* is used to mean shall be able to (in the sense of being technically possible).

In some cases, word usage is qualified to indicate on whom the obligation falls, such as *an application may* or *an implementation shall*.

2.2 Implementation, application

The word *implementation* is used to mean any specific implementation of the full UVM-SystemC class library as defined in this standard, only the public interface of which need be exposed to the application.

The word *application* is used to mean a C++ program, written by an end user, that uses the UVM-SystemC class library, that is, uses classes, functions, or macros defined in this standard.

2.3 Call, called from, derived from

The term *call* is taken to mean call directly or indirectly. Call indirectly means call an intermediate function that in turn calls the function in question, where the chain of function calls may be extended indefinitely.

Similarly, *called from* means called from directly or indirectly.

Except where explicitly qualified, the term *derived from* is taken to mean derived directly or indirectly from. Derived indirectly from means derived from one or more intermediate base classes.

2.4 Implementation-defined

The italicized term *implementation-defined* is used where part of a C++ definition is omitted from this standard. In such cases, an implementation shall provide an appropriate definition that honors the semantics defined in this standard.

3. Overview

3.1 Namespace

All UVM-SystemC classes and functions shall reside inside the namespace **uvm**.

3.2 Header files

An application shall include the C++ header file **uvm** or **uvm.h** to make use of the UVM-SystemC class library functions. The header file named **uvm** shall only add the name **uvm** to the declarative region in which it is included, whereas the header file named **uvm.h** shall add *all* of the names from the namespace **uvm** to the declarative region in which it is included.

NOTE—It is recommended that an application includes the header file **uvm** rather than the header file **uvm.h**. This means the namespace uvm has to be mentioned explicitly when using UVM-SystemC classes and functions. Alternatively, an application may use the C++ *using directive* at the global and local scope to gain access to these classes and functions.

3.3 Global functions

A minimal set of global functionality is defined offering generic UVM capabilities and convenience functions for configuration and printing. The global functions, enums, type defintions, and classes **uvm_coreservice_t** and **uvm_default_coreservice_t** are specified in Chapter 17.

3.4 Base classes

These classes define the base UVM class for all other UVM classes, and the base class for data objects:

— **uvm_void**

— **uvm_object**

— **uvm_root**

— **uvm_port_base**

— **uvm_export_base**§

— **uvm_component_name**§

The base classes are specified in Chapter 4.

3.5 Policy classes

These classes include policy objects for various operations based on class **uvm_object**:

— The class **uvm_printer** provides an interface for printing objects of type **uvm_object** in various formats. Classes derived from class **uvm_printer** implement pre-defined printing formats or policies:
  — The class **uvm_table_printer** prints the object in a tabular form.
  — The class **uvm_tree_printer** prints the object in a tree form.

  — The class **uvm_line_printer** prints the information on a single line, but uses the same object separators as the tree printer.

  These printer classes have 'knobs' that an application may use to control what and how information is printed. These knobs are contained in a separate knob class **uvm_printer_knobs**

— **uvm_comparer**: performs deep comparison of objects derived from **uvm_object**. An application may configure what is compared and how miscompares are reported.

— **uvm_packer**: performs packing (serialization) and unpacking of properties.

The policy classes are specified in Chapter 5.

3.6 Registry and factory classes

The registry and factory classes include the **uvm_factory** and associated classes for object and component registration. The class **uvm_factory** implements a factory pattern. A singleton factory instance is created for a given simulation run. Class types are registered with the factory using the class **uvm_object_wrapper** and its derivatives. The class **uvm_factory** supports type and instance overrides.

The registry and factory classes are:

— **uvm_object_wrapper**

— **uvm_object_registry**

— **uvm_component_registry**

— **uvm_factory**

— **uvm_default_factory**

The registry and factory classes are specified in Chapter 6.

3.7 Component hierarchy classes

These classes define the base class for hierarchical UVM components and the test environment. The class **uvm_component** provides interfaces for:

— Hierarchy—Provides methods for searching and traversing the component hierarchy.

— Configuration—Provides methods for configuring component topology and other parameters before and during component construction.

— Phasing—Defines a phased test flow that all components follow. Methods include the phase callbacks, such as **run_phase** and **report_phase**, overridden by the derived classes. During simulation, these callbacks are executed in precise order.

— Factory—Provides a convenience interface to the **uvm_factory**. The factory is used to create new components and other objects based on type-wide and instance-specific configuration.

All structural component classes **uvm_env**, **uvm_test**, **uvm_agent**, **uvm_driver**, **uvm_monitor**, **uvm_subscriber** and **uvm_scoreboard** are derived from the class **uvm_component**.

The UVM component classes are specified in Chapter 7.

3.8 Sequencer classes

The sequencer classes serve as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of transactions of type **uvm_sequence_item** generated by one or more sequences based on type **uvm_sequence**. The sequencer classes are:

- **uvm_sequencer_base**
- **uvm_sequencer_param_base**
- **uvm_sequencer**

The sequencer classes are specified in Chapter 8.

3.9 Sequence classes

The sequence classes offer the infrastructure to create stimuli descriptions based on transactions, encapsulated as a sequence or sequence item. The following sequence classes are defined:

- **uvm_transaction**
- **uvm_sequence_item**
- **uvm_sequence_base**
- **uvm_sequence**

The sequence classes are specified in Chapter 9.

3.10 Configuration and resource classes

The configuration and resource classes provide access to the configuration and resource database. The configuration database is used to store and retrieve both configuration time and run time properties. The configuration and resource classes are:

- **uvm_config_db**: Configuration database, which acts as interface on top of the resource database.
- **uvm_resource_db**: Resource database.
- **uvm_resource_options**: Provides a namespace for managing options for the resources facility.
- **uvm_resource_base**: Provides a non-parameterized base class for resources.
- **uvm_resource_pool**: Provides the global resource database.
- **uvm_resource**: Defines the parameterized resource.

This configuration and resource classes are specified in Chapter 10.

3.11 Phasing and synchronization classes

The phasing classes define the order of execution of pre-defined callback function and processes, which run either sequentially or concurrently. In addition, dedicated member functions for synchronization are available to coordinate the execution of or status of these processes between all UVM components or objects.

The phasing and synchronization classes are:

- **uvm_phase**: The base class for defining a phase’s behavior, state, context.
- **uvm_domain**: Phasing schedule node representing an independent branch of the schedule.
- **uvm_bottomup_phase**: A phase implementation for bottom up function phases.

— **uvm_topdown_phase**: A phase implementation for top-down function phases.
— **uvm_process_phase**° (**uvm_task_phase**†): A phase implementation for phases which are launched as spawned process.
— **uvm_objection**: Mechanism to synchronize phases based on passing execution status information between running processes.
— **uvm_callbacks**: The base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component base class for user-defined callback classes.
— **uvm_callback_iter**: A class for iterating over callback queues of a specific callback type.
— **uvm_callback**: The base class for user-defined callback classes.

The phasing and synchronization classes are specified in Chapter 11.

3.12 Reporting classes

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. An application can also filter out reports based on their verbosity, identity, or severity.

The following reporting classes are defined:

— **uvm_report_object**: The base class which provides the interface to the UVM reporting mechanism.
— **uvm_report_handler**: The class which acting as implementation for the member functions defined in the class **uvm_report_object**.
— **uvm_report_server** and **uvm_default_report_server**: The class acting as global server that processes all of the reports generated by the class **uvm_report_handler**.
— **uvm_report_catcher**: The class which captures and counts all reports issued by the class **uvm_report_server**.

The reporting classes are specified in Chapter 12.

3.13 Macros

The UVM-SystemC macros make common code easier to write. It is not imperative to use the macros, but in many cases the macros can save a substantial amount of user-written code. The macros defined in UVM-SystemC are:

— Macros for component and object registration:
    — **UVM_OBJECT_UTILS**
    — **UVM_OBJECT_PARAM_UTILS**
    — **UVM_COMPONENT_UTILS**
    — **UVM_COMPONENT_PARAM_UTILS**
— Sequence execution macros:
    — **UVM_DO**, **UVM_DO_ON** and **UVM_DO_ON_PRI**
    — **UVM_CREATE**,**UVM_CREATE_ON**
    — **UVM_DECLARE_P_SEQUENCER**
— Reporting macros:

  — **UVM_INFO**, **UVM_ERROR**, **UVM_WARNING** and **UVM_FATAL**
- Callback macros:
  — **UVM_REGISTER_CB** and **UVM_DO_CALLBACKS**

Detailed information for the macros or the associated member functions are specified in Chapter 13.

3.14 TLM classes

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

The following TLM-1 classes are defined:

- TLM-1 blocking ports **uvm_blocking_put_port**, **uvm_blocking_get_port**, **uvm_blocking_peek_port**, and **uvm_blocking_get_peek_port**.
- TLM-1 non-blocking ports **uvm_nonblocking_put_port**, **uvm_nonblocking_get_port**, **uvm_nonblocking_peek_port**, and **uvm_nonblocking_get_peek_port**.
- TLM analysis ports and exports **uvm_analysis_port**, **uvm_analysis_export**, and **uvm_analysis_imp**.
- The request-response channel class **uvm_tlm_req_rsp_channel**.
- The sequencer interface classes: **uvm_sqr_if_base**, **uvm_seq_item_pull_port**, **uvm_seq_item_pull_export**, and **uvm_seq_item_pull_imp**.

The TLM classes are specified in Chapter 14.

NOTE—UVM-SystemC does not define the TLM-2.0 blocking and non-blocking transport interfaces, direct memory interface (DMI), nor a debug transport interface. An application should use the SystemC TLM-2.0 interfaces instead.

3.15 Register abstraction classes

The register abstraction classes, when properly extended, abstract the read/write operations to registers and memories in a DUT.

The register abstraction classes are specified in Chapter 15 and Chapter 16.

3.16 Existing SystemC functionality used in UVM-SystemC

Because SystemVerilog does not support multiple inheritance, UVM-SystemVerilog is constrained to have only one base class, from which both data elements and hierarchical elements inherit. As SystemC is based on C++, it supports multiple inheritance. As such, UVM-SystemC uses multiple inheritance where suitable.

In UVM-SystemVerilog, the class **uvm_component** inherits from class **uvm_report_object**. In UVM-SystemC, class **uvm_component** applies multiple inheritance and derives from the SystemC class **sc_core**::**sc_module** and from **uvm_report_object**. Note that the class **uvm_object** is not derived from class **sc_core**::**sc_object**, but from class **uvm_void**.

The class **sc_core**::**sc_module** already offer the hierarchical features that **uvm_component** needs, namely parent and children, and a full instance name. Therefore the parent of a component does not need to be explicitly specified as a constructor argument; instead the class **uvm_component_name** keeps track of the component hierarchy.

The class **sc_core**::**sc_module** has natural equivalents to some of the UVM pre-run phases, which can used in a UVM-SystemC **uvm_component**. For example:

— The UVM-SystemC callback **before_end_of_elaboration** is mapped onto the UVM callback **build_phase**. Note that UVM-SystemC also provides the callback **build_phase** as an alternative to **before_end_of_elaboration**. It is recommended to use this UVM member function.
— The UVM-SystemC callback **end_of_elaboration** is mapped onto the UVM callback **end_of_elaboration_phase**. UVM-SystemC also provides the callback **end_of_elaboration_phase** with the argument of type **uvm_phase** as an alternative to the callback **end_of_elaboration**, which does give access to the phase information. It is recommended to use this UVM member function.
— The UVM-SystemC callback **start_of_simulation** is mapped onto the UVM callback **start_of_simulation_phase**. UVM-SystemC also provides the callback **start_of_simulation_phase** with the argument of type **uvm_phase** as an alternative to the callback **start_of_simulation**, which does give access to the phase information. It is recommended to use this UVM member function.

UVM-SystemC also defines the callback **run_phase** as a thread process of a **uvm_component** . This works because **sc_core**::**sc_module** in SystemC already has the ability to own and spawn thread processes.

UVM-SystemVerilog defines the TLM-1 interfaces like **put** and **get**, as well as some predefined TLM-1 channels like **tlm**::**tlm_fifo**. These already natively exist in the SystemC standard. UVM-SystemC supports the original SystemC TLM-1 definitions. The same holds for the analysis interface in UVM. UVM-SystemC offers a compatibility and convenience layer on top of the SystemC TLM interface proper **tlm**::**tlm_analysis_if** and analysis port **tlm**::**tlm_analysis_port**, defining elements such as **uvm_analysis_port**, **uvm_analysis_export** and **uvm_analysis_imp**.

The SystemC fork-join constructs **SC_FORK** and **SC_JOIN** can be used as a pair to bracket a set of calls to function **sc_core**::**sc_spawn** within a UVM component **run_phase**, enabling the creation of concurrent processes.

3.17 Methodology for hierarchy construction

The UVM in SystemVerilog recommends the use of configurations by using the static member function **set** of the **uvm_config_db** in the build phase, followed by hierarchy construction through the factory, in the same phase.

In UVM-SystemVerilog, it is necessary to make the connections (port binding) in the connect phase, which happens after hierarchy construction of components, ports and exports in the build phase. This enables configuration of port/export construction using the configuration database **uvm_config_db**. In that case, if a parent creates a child in the build phase, that child’s port/export does not exist at that point, and it has to wait for the next phase to bind the child’s port/export.

Consistent with UVM in SystemVerilog, UVM-SystemC also recommends configurations using **uvm_config_db** and hierarchy construction through the factory **uvm_factory** in the build phase. This implies that child objects derived from class **uvm_component** should be declared as pointers inside the parent class, and these children should be constructed in the UVM callback **build_phase** through the UVM factory, which does not contradict the SystemC standard, as the SystemC standard allows construction activity in the callback **before_end_of_elaboration**, which is equivalent to the UVM build phase.

In SystemC, the ports/exports are usually becoming members of a **uvm_component** and not pointers. In that case, the ports/exports are automatically created and initialized in the constructor of the parent

uvm_component. This implies that in UVM-SystemC the ports/export construction is *not* configurable through **uvm_config_db**. Because the bulk of the UVM hierarchy construction occurs in the build phase, the port/export bindings that depend on the entire hierarchy being constructed have to be done in a later phase. Similar as in UVM-SystemVerilog, the connect phase is introduced in UVM-SystemC to perform the port bindings using the **sc_core**::**sc_port** member function **bind** or **operator()**. The UVM binding mechanism using the member function **connect** of the ports is made available for compatibility purposes.

4. Base classes

4.1 uvm_void

The class **uvm_void** shall provide the base class for all UVM classes. It shall be an abstract class with no data members or functions, to allow the creation of a generic container of objects.

An application may derive directly from this class and inherits none of the UVM functionality, but such classes may be placed in **uvm_void**-typed containers along with other UVM objects.

4.1.1 Class definition

```
namespace uvm {
  class uvm_void {};
} // namespace uvm
```

4.2 uvm_object

The class **uvm_object** shall provide the base class for all UVM data and hierarchical classes. Its primary role is to define a set of member functions for common operations such as create, copy, compare, print, and record. Classes deriving from **uvm_object** shall implement the member functions such as **create** and **get_type_name**.

4.2.1 Class definition

```
namespace uvm {

  class uvm_object : public uvm_void
  {
   public:

    // Group: Construction
    uvm_object();
    explicit uvm_object( const std::string& name );

    // Group: Identification
    virtual void set_name( const std::string& name );
    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual int get_inst_id() const;
    static int get_inst_count();
    static const uvm_object_wrapper* get_type();
    virtual const uvm_object_wrapper* get_object_type() const;
    virtual const std::string get_type_name() const;

    // Group: Creation
    virtual uvm_object* create( const std::string& name = "" );
    virtual uvm_object* clone();

    // Group: Printing
    void print( uvm_printer* printer = NULL ) const;
    std::string sprint( uvm_printer* printer = NULL ) const;
    virtual void do_print( const uvm_printer& printer ) const;
    virtual std::string convert2string() const;

    // Group: Recording
    void record( uvm_recorder* recorder = NULL );
    virtual void do_record( const uvm_recorder& recorder );

    // Group: Copying
    void copy( const uvm_object& rhs );
    virtual void do_copy( const uvm_object& rhs );

    // Group: Comparing
    bool compare( const uvm_object& rhs,
```

```
                   const uvm_comparer* comparer = NULL ) const;

    virtual bool do_compare( const uvm_object& rhs,
                             const uvm_comparer* comparer = NULL ) const;

    // Group: Packing
    int pack( std::vector<bool>& bitstream, uvm_packer* packer = NULL );
    int pack_bytes( std::vector<unsigned char>& bytestream, uvm_packer* packer = NULL );
    int pack_ints( std::vector<unsigned int>& intstream, uvm_packer* packer = NULL );
    virtual void do_pack( uvm_packer& packer ) const;

    // Group: Unpacking
    int unpack( const std::vector<bool>& v, uvm_packer* packer = NULL );
    int unpack_bytes( const std::vector<unsigned char>& v, uvm_packer* packer = NULL );
    int unpack_ints( const std::vector<unsigned int>& v, uvm_packer* packer = NULL );
    virtual void do_unpack( uvm_packer& packer );

  }; // class uvm_object

} // namespace uvm
```

4.2.2 Constructors

```
uvm_object();
explicit uvm_object( const std::string& name );
```

The constructor shall create a new **uvm_object** with the given instance *name* passed as argument. If no argument is given, the default constructor shall call function **sc_core**::**sc_gen_unique_name** (“object”) to generate a unique string name as instance name of this object.

4.2.3 Identification

4.2.3.1 set_name

```
virtual void set_name( const std::string& name );
```

The member function **set_name** shall set the instance name of this object passed as argument, overwriting any previously given name. It shall be an error if the name is already in use for another object.

4.2.3.2 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the name of the object, as provided by the argument *name* via the constructor or member function **set_name**.

4.2.3.3 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of this object. The default implementation is the same as **get_name**, as objects of type **uvm_object** do not inherently possess hierarchy.

NOTE—Objects possessing hierarchy, such as objects of type **uvm_component**, override the default implementation. Other objects might be associated with component hierarchy, but are not themselves components. For example, sequence classes of type **uvm_sequence** are typically associated with a sequencer class of type **uvm_sequencer**. 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.

4.2.3.4 get_inst_id

```
virtual int get_inst_id() const;
```

The member function **get_inst_id** shall return the object's unique, numeric instance identifier.

4.2.3.5 get_inst_count

```
static int get_inst_count();
```

The member function **get_inst_count** shall return the current value of the instance counter, which represents the total number of objects of type **uvm_object** that have been allocated in simulation. The instance counter is used to form a unique numeric instance identifier.

4.2.3.6 get_type

```
static const uvm_object_wrapper* get_type();
```

The member function **get_type** shall return the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation member functions take arguments of **uvm_object_wrapper**. The default implementation of this member function shall produce an error and return NULL.

4.2.3.7 get_object_type

```
virtual const uvm_object_wrapper* get_object_type() const;
```

The member function **get_object_type** shall the return the type-proxy (wrapper) for this object. The **uvm_factory**'s type-based override and creation member functions take arguments of **uvm_object_wrapper**. The default implementation of this member function does a factory lookup of the proxy using the return value from **get_type_name**. If the type returned by **get_type_name** is not registered with the factory, then the member function shall return NULL.

This member function behaves the same as the static member function **get_type**, but uses an already allocated object to determine the type-proxy to access (instead of using the static object).

4.2.3.8 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object, which is typically the type identifier enclosed in quotes. It is used for various debugging functions in the library, and it is used by the factory for creating objects.

4.2.4 Creation

4.2.4.1 create

```
virtual uvm_object* create( const std::string& name = "" );
```

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

4.2.4.2 clone

```
virtual uvm_object* clone();
```

The member function **clone** shall create and return a pointer to an exact copy of this object.

NOTE—As the member function clone is virtual, derived classes may override this implementation if desired.

4.2.5 Printing

4.2.5.1 print

```
void print( uvm_printer* printer = NULL ) const;
```

The member function **print** shall deep-print this object's properties in a format and manner governed by the given argument *printer*. If the argument *printer* is not provided, the global **uvm_default_printer** shall be used (see Section 5.7.4).

The member function **print** is not virtual and shall not be overloaded. To include custom information in the **print** and **sprint** operations, derived classes shall override the member function **do_print** and can use the provided printer policy class to format the output.

4.2.5.2 sprint

```
std::string sprint( uvm_printer* printer = NULL ) const;
```

The member function **sprint** shall return the object's properties as a string and in a format and manner governed by the given argument *printer*. If the argument *printer* is not provided, the global **uvm_default_printer** shall be used (see Section 5.7.4).

The member function **sprint** is not virtual and shall not be overloaded. To include additional fields in the **print** and **sprint** operation, derived classes shall override the member function **do_print** and use the provided printer policy class to format the output. The printer policy shall manage all string concatenations and provide the string to **sprint** to return to the caller.

4.2.5.3 do_print

```
virtual void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall provide a context called by the member functions **print** and **sprint** that allows an application to customize what gets printed. The argument *printer* is the policy object that governs the format and content of the output. To ensure correct **print** and **sprint** operation, and to ensure a consistent output format, the printer shall be used by all **do_print** implementations.

4.2.5.4 convert2string

```
virtual std::string convert2string() const;
```

The member function **convert2string** shall provide a context which may be called directly by the application, to provide object information in the form of a string. Unlike the member function **sprint**, there is no requirement to use a **uvm_printer** policy object. As such, the format and content of the output is fully customizable, which may be suitable for applications not requiring the consistent formatting offered by the **print**/**sprint**/**do_print** API.

4.2.6 Recording

4.2.6.1 record

```
void record( uvm_recorder* recorder = NULL );
```

The member function **record** shall deep-records this object's properties according to an optional recorder policy. The member function is not virtual and shall not be overloaded. To include additional fields in the record operation, derived classes should override the member function **do_record**.

The optional argument *recorder* specifies the recording policy, which governs how recording takes place. If a recorder policy is not provided explicitly, then the global **uvm_default_recorder** policy is used (see Section 5.7.7).

NOTE—The recording mechanism is implementation-defined. The **uvm_recorder** policy provides a standardized interface to a simulator's recording capabilities.

4.2.6.2 do_record

```
virtual void do_record( const uvm_recorder& recorder );
```

The member function **do_record** shall provide a context called by the member function **record**. A derived class should overload this member function to include its fields in a record operation.

The argument *recorder* is policy object for recording this object. A **do_record** implementation should call the appropriate recorder member function for each of its fields.

NOTE—The actual recording mechanism is implementation defined, thereby insulating the application from the implementation.

4.2.7 Copying

4.2.7.1 copy

```
void copy( const uvm_object& rhs );
```

The member function **copy** shall make a copy of the specified object passed as argument.

The member function is not virtual and shall not be overloaded in derived classes. To copy the fields of a derived class, that class shall overload the member function **do_copy**.

4.2.7.2 do_copy

```
virtual void do_copy( const uvm_object& rhs );
```

The member function **do_copy** shall provide a context called by the member function **copy**. A derived class should overload this member function to include its fields in a copy operation.

4.2.8 Comparing

4.2.8.1 compare

```
bool compare( const uvm_object& rhs,
              const uvm_comparer* comparer = NULL ) const;
```

The member function **compare** shall compare members of this data object with those of the object provided in the rhs (right-hand side) argument. It shall return true on a match; otherwise it shall return false.

The optional argument *comparer* specifies the comparison policy. It allows an application to control some aspects of the comparison operation. It also stores the results of the comparison, such as field-by-field miscompare information and the total number of miscompares. If a comparer policy is not provided or set to NULL, then the global **uvm_default_comparer** policy is used (see Section 5.7.6).

The member function is not virtual and shall not be overloaded in derived classes. To compare the fields of a derived class, that class shall overload the member function **do_compare**.

4.2.8.2 do_compare

```
virtual bool do_compare( const uvm_object& rhs,
                         const uvm_comparer* comparer = NULL ) const;
```

The member function **do_compare** shall provide a context called by the member function **compare**. A derived class should overload this member function to include its fields in a compare operation. The member function shall return true if the comparison succeeds; otherwise it shall return false.

4.2.9 Packing

4.2.9.1 pack

```
int pack( std::vector<bool>& bitstream, uvm_packer* packer = NULL );
```

The member function **pack** shall concatenate the object properties into a vector of bits. The member function shall return the total number of bits packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the pack operation, derived classes shall overload the member function **do_pack**.

4.2.9.2 pack_bytes

```
int pack_bytes( std::vector<char>& bytestream, uvm_packer* packer = NULL );
```

The member function **pack_bytes** shall concatenate the object properties into a vector of bytes. The member function shall return the total number of bytes packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the pack operation, derived classes shall overload the member function **do_pack**.

4.2.9.3 pack_ints

```
int pack_ints( std::vector<int>& intstream, uvm_packer* packer = NULL );
```

The member function **pack_ints** shall concatenate the object properties into a vector of integers. The member function shall return the total number of integers packed into the given vector.

The optional argument *packer* specifies the packing policy, which governs the packing operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the pack operation, derived classes shall overload the member function **do_pack**.

4.2.9.4 do_pack

```
oid do_pack( uvm_packer& packer ) const;
```

The member function **do_pack** shall provide a context called by the member functions **pack**, **pack_bytes** and **pack_ints**. A derived class should overload this member function to include its fields in a packing operation.

The argument *packer* is the policy object for packing and should be used to pack objects.

4.2.10 Unpacking

4.2.10.1 unpack

```
int unpack( const std::vector<bool>& bitstream, uvm_packer* packer = NULL );
```

The member function **unpack** shall extract the values from a vector of bits. The member function shall return the total number of bits unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs both the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes shall overload the member function **do_unpack**.

NOTE—The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector. The behavior is undefined in case a different packer policy or ordering is applied for packing and unpacking.

4.2.10.2 unpack_bytes

```
int unpack_bytes( const std::vector<char>& bytestream, uvm_packer* packer = NULL );
```

The member function **unpack_bytes** shall extract the values from a vector of bytes. The member function shall return the total number of bytes unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes shall overload the member function **do_unpack**.

NOTE—The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector. The behavior is undefined in case a different packer policy or ordering is applied for packing and unpacking.

4.2.10.3 unpack_ints

```
int unpack_ints( const std::vector<int>& intstream, uvm_packer* packer = NULL );
```

The member function **unpack_ints** shall extract the values from a vector of integers. The member function shall return the total number of integers unpacked from the given vector.

The optional argument *packer* specifies the packing policy, which governs the pack and unpack operation. If a packer policy is not provided or set to NULL, the global **uvm_default_packer** policy shall be used (see Section 5.7.5).

The member function is not virtual and shall not be overloaded in derived classes. To include additional fields in the unpack operation, derived classes shall overload the member function do_unpack.

NOTE—The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector. The behavior is undefined in case a different packer policy or ordering is applied for packing and unpacking.

4.2.10.4 do_unpack

```
>virtual void do_unpack( uvm_packer& packer ) const;
```

The member function **do_unpack** shall provide a context called by the member functions **unpack**, **unpack_bytes** and **unpack_ints**. A derived class should overload this member function to include its fields in a unpacking operation. The member function shall return true if the unpacking succeeds; otherwise it shall return false.

The argument *packer* is the policy object for unpacking and should be used to unpack objects.

NOTE—The application of the member function for unpacking shall exactly correspond to the member function for packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input vector. The behavior is undefined in case a different packer policy or ordering is applied for packing and unpacking.

4.2.11 Object macros

UVM-SystemC provides the following macros for a **uvm_object**:

— utility macro **UVM_OBJECT_UTILS**(*classname*) is to be used inside the class definition that expands to:
  — The declaration of the member function **get_type_name,** which returns the type of a class as string.
  — The declaration of the member function **get_type**, which returns a factory proxy object for the type.
  — The declaration of the proxy class **uvm_object_registry<** *classname*> used by the factory.
— Template classes shall use the macro **UVM_OBJECT_PARAM_UTILS**, to guarantee correct registration of one or more parameters passed to the class template. Note that template classes are not evaluated at compile-time, and thus not registered with the factory. Due to this, name-based lookup with the factory for template classes is not possible. Instead, an application shall use the member function **get_type** for factory overrides.

4.3 uvm_root

The class **uvm_root** serves as the implicit top-level and phase controller for all UVM components. An application shall not directly instantiate **uvm_root**. A UVM implementation shall create a single instance of **uvm_root** that an application can access via the global variable **uvm_top**.

4.3.1 Class definition

```
namespace uvm {

  class uvm_root : public uvm_component
  {
   public:
    static uvm_root* get();

    // Group: Simulation control
    virtual void run_test( const std::string& test_name = "" );
    virtual void die();
    void set_timeout( const sc_core::sc_time& timeout, bool overridable = true );
    void set_finish_on_completion( bool enable );
    bool get_finish_on_completion();

    // Group: Topology
    uvm_component* find( const std::string& comp_match );
    void find_all( const std::string& comp_match,
                   std::vector<uvm_component*>& comps,
                   uvm_component* comp = NULL );
    void print_topology( uvm_printer* printer = NULL );
    void enable_print_topology( bool enable = true );

    // Global variable
    const uvm_root* uvm_top;

  }; // class uvm_root
} // namespace uvm
```

4.3.2 Simulation control

4.3.2.1 run_test

```
virtual void run_test( const std::string& test_name = "" );
```

The member function **run_test** shall register the UVM phasing mechanism. If the optional argument *test_name* is provided, then the specified test component is created just prior to phasing, if and only if this component is derived from class **uvm_test**. Otherwise it shall be an error.

The phasing mechanism is used during test execution, where all components are called following a defined set of registered phases. The member function **run_test** shall register both the common phases as well as the UVM run-time phases. (See Chapter 11).

NOTE 1—Selection of the test via the command line interface is not yet available.

NOTE 2—The test execution is started using the SystemC function **sc_core::sc_start**. It is recommended not to specify the simulation stop time, as the end-of-test is automatically managed by the phasing mechanism.

4.3.2.2 die

```
virtual void die();
```

The member function **die** shall be called by the report server if a report reaches the maximum quit count or has a **UVM_EXIT** action associated with it, e.g., as with fatal errors. The member function shall call the member function **uvm_component**::**pre_abort** on the entire UVM component hierarchy in a bottom-up fashion. It then shall call **uvm_report_server**::**report_summarize** and terminate the simulation.

4.3.2.3 set_timeout

```
void set_timeout( const sc_core::sc_time& timeout, bool overridable = true );
```

The member function **set_timeout** shall define the timeout for the run phases. If not called, the default timeout shall be set to **UVM_DEFAULT_TIMEOUT** (see Section 17.2.3).

4.3.2.4 set_finish_on_completion

```
void set_finish_on_completion( bool enable );
```

The member function **set_finish_on_completion** shall define how simulation is finalized. If the application did not call this member function or if the argument *enable* is set to true, it shall terminate the simulation after execution of the UVM phases. If the argument *enable* is set to false, the simulation shall be paused after execution of the UVM phases.

NOTE—An implementation may call the function **sc_core::sc_stop** to terminate the simulation. An implementation may call the function **sc_core::sc_pause** to pause the simulation.

4.3.2.5 get_finish_on_completion

```
bool get_finish_on_completion();
```

The member function **get_finish_on_completion** shall return true if the application has not called member function **set_finish_on_completion** or if the member function was called with the argument *enable* as true; otherwise it shall return false. (See also Section 4.3.2.4.)

4.3.3 Topology

4.3.3.1 find

```
uvm_component* find( const std::string& comp_match );
```

The member function **find** shall return a component handle matching the given string *comp_match*. The string may contain the wildcards '*' and '?'. Strings beginning with character '.' are absolute path names.

4.3.3.2 find_all

```
void find_all( const std::string& comp_match,
               std::vector<uvm_component*>& comps,
               uvm_component* comp = NULL );
```

The member function **find_all** shall return a vector of component handles matching the given string *comp_match*. The string may contain the wildcards '*' and '?'. Strings beginning with character '.' are absolute path names. If the optional component argument *comp* is provided, then the search begins from that component down; otherwise it searches all components.

4.3.3.3 print_topology

```
void print_topology( uvm_printer* printer = NULL );
```

The member function **print_topology** shall print the verification environment's component topology. The argument *printer* shall be an object of class **uvm_printer** that controls the format of the topology printout; a NULL printer prints with the default output.

4.3.3.4 enable_print_topology

```
void enable_print_topology( bool enable = true );
```

The member function **enable_print_topology** shall print the entire testbench topology just after completion of the **end_of_elaboration** phase, if enabled. By default, the testbench topology is not printed, unless enabled by the application by calling this member function.

4.3.4 Global variable

4.3.4.1 uvm_top

```
const uvm_root* uvm_top;
```

The data member **uvm_top** is a handle to the top-level (root) component that governs phase execution and provides the component search interface. By default, this handle is provided by the **uvm_root** singleton.

The **uvm_top** instance of **uvm_root** plays several key roles in the UVM:

— *Implicit top-level*: The **uvm_top** serves as an implicit top-level component. Any UVM component which is not instantiated in another UVM component (e.g. when instantiated in a **sc_core**::**sc_module** or in **sc_main**) becomes a child of **uvm_top**. Thus, all UVM components in simulation are descendants of **uvm_top**.

— *Phase control*: **uvm_top** manages the phasing for all components.

— *Search*: An application may use **uvm_top** to search for components based on their hierarchical name. See member functions **find** (Section 4.3.3.1) and **find_all** (Section 4.3.3.2).

— *Report configuration*: An application may use **uvm_top** to globally configure report verbosity, log files, and actions. For example, **uvm_top.set_report_verbosity_level_hier**(**UVM_FULL**) would set full verbosity for all components in simulation.

— *Global reporter*: Because **uvm_top** is globally accessible, the UVM reporting mechanism is accessible from anywhere outside **uvm_component**, such as in modules and sequences. See **uvm_report_error**, **uvm_report_warning**, and other global methods.

The **uvm_top** instance checks during the **end_of_elaboration_phase** if any errors have been generated so far. If errors are found a **UVM_FATAL** error is generated as result so that the simulation shall not continue to the **start_of_simulation_phase**.

4.4 uvm_port_base

The class **uvm_port_base** shall provide methods to bind ports to interfaces or to other ports, and to forward interface method calls to the channel to which the port is bound, according to the same mechanism as defined in SystemC. Therefore this class shall be derived from the class **sc_core**::**sc_port**.

4.4.1 Class definition

```
namespace uvm {

  template <class IF>
  class uvm_port_base : public sc_core::sc_port<IF>
  {
   public:
    uvm_port_base();
    explicit uvm_port_base( const std::string& name );

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_component* get_parent() const;
    virtual const std::string get_type_name() const;

    virtual void connect( IF& );
    virtual void connect( uvm_port_base<IF>& );

  // class uvm_port_base

} // namespace uvm
```

4.4.2 Template parameter IF

The template parameter IF shall specify the name of the interface type used for the port. The port can only be bound to a channel which is derived from the same type, or to another port which is derived from this type.

4.4.3 Constructor

```
uvm_port_base();
explicit uvm_port_base( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name,* if passed as an argument.

4.4.4 Member functions

4.4.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the leaf name of this port.

4.4.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of this port.

4.4.4.3 get_parent

```
virtual uvm_component* get_parent() const;
```

The member function **get_parent** shall return the handle to this port's parent, or NULL if it has no parent.

4.4.4.4 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name to this port. Derived port classes shall implement this member function to return the concrete type.

4.4.4.5 connect

```
virtual void connect( IF& );
virtual void connect( uvm_port_base<IF>& );
```

The member function **connect** shall bind this port to the interface given as argument.

NOTE—The member function **connect** implements the same functionality as the SystemC member function **bind**.

4.5 uvm_export_base§

The class **uvm_export_base**§ shall provide methods to bind exports to interfaces or to other exports, and to forward interface method calls to the channel to which the export is bound, according to the same mechanism as defined in SystemC. Therefore this class shall be derived from the class **sc_core**::**sc_export**.

4.5.1 Class definition

```
namespace uvm {

  template <class IF>
  class uvm_export_base§ : public sc_core::sc_export<IF>
  {
   public:
    uvm_export_base();
    explicit uvm_export_base( const std::string& name );

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_component* get_parent() const;
    virtual const std::string get_type_name() const;

    virtual void connect( IF& );
    virtual void connect( uvm_export_base<IF>& );

  // class uvm_export_base

} // namespace uvm
```

4.5.2 Template parameter IF

The template parameter IF shall specify the name of the interface type used for the export. The export can only be bound to a channel which is derived from the same type, or to another export which is derived from this type.

4.5.3 Constructor

```
uvm_export_base();
explicit uvm_export_base( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name,* if passed as an argument.

4.5.4 Member functions

4.5.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the leaf name of this export.

4.5.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of this export.

4.5.4.3 get_parent

```
virtual uvm_component* get_parent() const;
```

The member function **get_parent** shall return the handle to this export's parent, or NULL if it has no parent.

4.5.4.4 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name to this export. Derived export classes shall implement this member function to return the concrete type.

4.5.4.5 connect

```
virtual void connect( IF& );
virtual void connect( uvm_export_base<IF>& );
```

The member function **connect** shall bind this export to the interface given as argument.

NOTE—The member function **connect** implements the same functionality as the SystemC member function **bind**.

4.6 uvm_component_name§

The class **uvm_component_name**§ is shall provide the mechanism for building the hierarchical names of component instances and component hierarchy during elaboration.

An implementation shall maintain the UVM component hierarchy, that is, it shall build a list of hierarchical component names, where each component instance is named as if it were a child of another component (its parent). The mechanism to implement such component hierarchy is implementation-defined.

NOTE 1—The hierarchical name of an instance in the component hierarchy is returned from member function get_full_name of class **uvm_component**, which is the base class of all component instances.

NOTE 2—An object of type **uvm_object** may have a hierarchical name and may have a parent in the component hierarchy, but such object is not part of the component hierarchy.

4.6.1 Class definition

```
namespace uvm {

  class uvm_component_name§
  {
   public:
    uvm_component_name( const char* name );
    uvm_component_name( const uvm_component_name& name );
    ~uvm_component_name();
    operator const char*() const;

   private:
    // Disabled
    uvm_component_name();
    uvm_component_name& operator= ( const uvm_component_name& name );
  }; // class uvm_component_name

} // namespace uvm
```

4.6.2 Constraints on usage

The class **uvm_component_name** shall only be used as argument in a constructor of a class derived from class **uvm_component**. Such constructor shall only contain this argument of type **uvm_component_name**.

4.6.3 Constructor

```
uvm_component_name( const char* name );
```

The constructor **uvm_component_name**(const char* *name*) shall store the name in the component hierarchy. The constructor argument *name* shall be used as the string name for that component being instantiated within the component hierarchy.

NOTE—An application shall define for each class derived directly or indirectly from class **uvm_component** a constructor with a single argument of type **uvm_component_name**, where the constructor **uvm_component_name**(const char*) is called as an implicit conversion.

```
uvm_component_name( const uvm_componet_name& name );
```

The constructor **uvm_component_name**(const **uvm_component_name**& *name*) shall copy the constructor argument but shall not modify the component hierarchy.

NOTE—When an application derives a class directly or indirectly from class **uvm_component**, the derived class constructor calls the base class constructor with an argument of class **uvm_component_name** and thus this copy constructor is called.

4.6.4 Destructor

```
~uvm_component_name();
```

The destructor shall remove the object from the component hierarchy if, and only if, the object being destroyed was constructed by using the constructor signature **uvm_component_name**(const char* *name*).

4.6.5 operator const char*

```
operator const char*() const;
```

This conversion function shall return the string name (not the hierarchical name) associated with the **uvm_component_name**.

5. Policy classes

The UVM policy classes provide specific tasks for printing, comparing, recording, packing, and unpacking of objects derived from class **uvm_object**. They are implemented separately from class **uvm_object** so that an application can plug in different ways to print, compare, etc. without modifying the object class being operated on. 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. An application may also customize operations by deriving new policy subtypes from these base types. For example, the UVM provides four different printer policy classes derived from the policy base class **uvm_printer**, each of which print objects in a different format.

The following policy classes are defined:

— **uvm_packer**

— **uvm_printer, uvm_table_printer**, **uvm_tree_printer**, **uvm_line_printer** and **uvm_printer_knobs**.

— **uvm_recorder**

— **uvm_comparer**

5.1 uvm_packer

The class **uvm_packer** provides a policy object for packing and unpacking objects of type **uvm_object**. The policies determine how packing and unpacking should be done. Packing an object causes the object to be placed into a packed array of type byte or int. Unpacking an object causes the object to be filled from the pack array. The logic values X and Z are lost on packing. The maximum size of the packed array is defined by **UVM_PACKER_MAX_BYTES** (see Section 17.2.2).

5.1.1 Class definition

```
namespace uvm {

  class uvm_packer
  {
   public:

    // Group: Packing
    virtual void pack_field( const uvm_bitstream_t& value, int size );
    virtual void pack_field_int( const uvm_integral_t& value, int size );
    virtual void pack_string( const std::string& value );
    virtual void pack_time( const sc_core::sc_time& value );
    virtual void pack_real( double value );
    virtual void pack_real( float value );
    virtual void pack_object( const uvm_object& value );
    virtual uvm_packer& operator<< ( bool value );
    virtual uvm_packer& operator<< ( double& value );
    virtual uvm_packer& operator<< ( float& value );
    virtual uvm_packer& operator<< ( char value );
    virtual uvm_packer& operator<< ( unsigned char value );
    virtual uvm_packer& operator<< ( short value );
    virtual uvm_packer& operator<< ( unsigned short value );
    virtual uvm_packer& operator<< ( int value );
    virtual uvm_packer& operator<< ( unsigned int value );
    virtual uvm_packer& operator<< ( long value );
    virtual uvm_packer& operator<< ( unsigned long value );
    virtual uvm_packer& operator<< ( long long value );
    virtual uvm_packer& operator<< ( unsigned long long value );
    virtual uvm_packer& operator<< ( const std::string& value );
    virtual uvm_packer& operator<< ( const char* value );
    virtual uvm_packer& operator<< ( const uvm_object& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_logic& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_bv_base& value );
```

```
    virtual uvm_packer& operator<< ( const sc_dt::sc_lv_base& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_int_base& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_uint_base& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_signed& value );
    virtual uvm_packer& operator<< ( const sc_dt::sc_unsigned& value );

    template <class T>
    uvm_packer& operator<< ( const std::vector<T>& value );

    // Group: Unpacking
    virtual bool is_null();
    virtual uvm_integral_t unpack_field_int( int size );
    virtual uvm_bitstream_t unpack_field( int size );
    virtual std::string unpack_string( int num_chars = -1 );
    virtual sc_core::sc_time unpack_time();
    virtual double unpack_real();
    virtual float unpack_real();
    virtual void unpack_object( uvm_object& value );
    virtual unsigned int get_packed_size() const;

    virtual uvm_packer& operator>> ( bool& value );
    virtual uvm_packer& operator>> ( double& value );
    virtual uvm_packer& operator>> ( float& value );
    virtual uvm_packer& operator>> ( char& value );
    virtual uvm_packer& operator>> ( unsigned char& value );
    virtual uvm_packer& operator>> ( short& value );
    virtual uvm_packer& operator>> ( unsigned short& value );
    virtual uvm_packer& operator>> ( int& value );
    virtual uvm_packer& operator>> ( unsigned int& value );
    virtual uvm_packer& operator>> ( long& value );
    virtual uvm_packer& operator>> ( unsigned long& value );
    virtual uvm_packer& operator>> ( long long& value );
    virtual uvm_packer& operator>> ( unsigned long long& value );
    virtual uvm_packer& operator>> ( std::string& value );
    virtual uvm_packer& operator>> ( uvm_object& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_logic& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_bv_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_lv_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_int_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_uint_base& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_signed& value );
    virtual uvm_packer& operator>> ( sc_dt::sc_unsigned& value );

    template <class T>
    virtual uvm_packer& operator>> ( std::vector<T>& value );

    // Data members (variables)
    bool physical;
    bool abstract;
    bool use_metadata;
    bool big_endian;

   private:
    // Disabled
    uvm_packer();

  }; // class uvm_packer

} // namespace uvm
```

5.1.2 Constraints on usage

An application shall not explicitly create an instance of the class **uvm_packer**.

5.1.3 Packing

5.1.3.1 pack_field

```
virtual void pack_field( const uvm_bitstream_t& value, int size );
```

The member function **pack_field** shall pack an integral value (less than or equal to **UVM_MAX_STREAMBITS**) into the packed array. The argument *size* is the number of bits of value to pack.

5.1.3.2 pack_field_int

```
virtual void pack_field_int( const uvm_integral_t& value, int size );
```

The member function **pack_field_int** shall pack the integral value (less than or equal to 64 bits) into the packed array. The argument *size* is the number of bits of value to pack.

NOTE—This member function is the optimized version of pack_field is useful for sizes up to 64 bits.

5.1.3.3 pack_string

```
virtual void pack_string( const std::string& value );
```

The member function **pack_string** shall pack a string value into the packed array. When the variable **metadata** is set, the packed string is terminated by a NULL character to mark the end of the string.

5.1.3.4 pack_time

```
virtual void pack_time( const sc_core::sc_time& value );
```

The member function **pack_time** shall pack a time value as 64 bits into the packed array.

5.1.3.5 pack_real

```
virtual void pack_real( double value );
virtual void pack_real( float value );
```

The member function **pack_real** shall pack a real value as binary vector into the packed array. When the argument is a double precision floating point value of type double, a 64 bit binary vector shall be used. When the argument is a single precision floating point value of type float, a 32 bit binary vector shall be used. The convertion of the floating point representation to binary vector shall be according to IEEE Std. 754-2019[5].

5.1.3.6 pack_object

```
virtual void pack_object( const uvm_object& value );
```

The member function **pack_object** shall pack an object value into the packed array. A 4-bit header is inserted ahead of the string to indicate the number of bits that was packed. If a NULL object was packed, then this header shall be 0.

5.1.4 Unpacking

5.1.4.1 is_null

```
virtual bool is_null();
```

[5] IEEE Standard for Floating-Point Arithmetic, https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html

The member function **is_null** shall be used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is zero. If the next four bits are all zero, then the return value is a true; otherwise it returns false.

NOTE—This member function is useful when unpacking objects, to decide whether a new object needs to be allocated or not.

5.1.4.2 unpack_field_int

```
virtual uvm_integral_t unpack_field_int( int size );
```

The member function **unpack_field_int** shall unpack bits from the packed array and returns the bit-stream that was unpacked. The argument *size* the number of bits to unpack; the maximum is 64 bits.

NOTE—This member function is a more efficient variant than **unpack_field** when unpacking into smaller vectors.

5.1.4.3 unpack_field

```
virtual uvm_bitstream_t unpack_field( int size );
```

The member function **unpack_field** shall unpack bits from the packed array and returns the bit-stream that was unpacked. The argument *size* is the number of bits to unpack; the maximum is defined by **UVM_MAX_STREAMBITS**.

5.1.4.4 unpack_string

```
virtual std::string unpack_string( int num_chars = -1 );
```

The member function **unpack_string** shall unpack a string. The argument *num_chars* specifies the number of bytes that are unpacked into a string. If *num_chars* is -1, then unpacking stops on at the first NULL character that is encountered.

5.1.4.5 unpack_time

```
virtual sc_core::sc_time unpack_time();
```

The member function **unpack_time** shall unpack the next 64 bits of the packed array and places them into a time variable.

5.1.4.6 unpack_real

```
virtual double unpack_real();
virtual float unpack_real();
```

The member function **unpack_real** shall unpack the next 64 bits of the packed array and places them into a real variable. The 64 bits of packed data shall be converted to double precision floating point notation according to IEEE Std. 754-2019.

5.1.4.7 unpack_object

```
virtual void unpack_object( uvm_object& value );
```

The member function **unpack_object** shall unpack an object and stores the result into *value*. Argument *value* shall be an allocated object that has enough space for the data being unpacked. The first four bits of packed data are used to determine if a null object was packed into the array. The member function **is_null** can be used to peek at the next four bits in the pack array before calling this member function.

5.1.4.8 get_packed_size

```
virtual unsigned int get_packed_size() const;
```

The member function **get_packed_size** returns the number of bits that were packed.

5.1.5 operator<<, operator>>

The class **uvm_packer** defines **operator<<** for packing, and **operator >>** for unpacking basic C++ types, SystemC types, the type **uvm_object**, and std::vector types. The supported data types are:

- Basic C++ types: bool, double, float, char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, and unsigned long long.
- SystemC types: **sc_dt**::**sc_logic**, **sc_dt**::**sc_bv**, **sc_dt**::**sc_lv**, **sc_dt**::**sc_int**, **sc_dt**::**sc_uint**, **sc_dt**::**sc_signed**, and **sc_dt**::**sc_unsigned**.
- String of type std::string and const char*
  When packing, an additional NULL byte is packed after the string is packed when **use_metadata** is set to true (see Section 5.1.6.3).
- Any type that derives from class **uvm_object**
- Vector types: std::vector<T>, where T is one of the supported data types listed above, and has an **operator<<** defined for it:
  When packing, additional 32 bits are packed indicating the size of the vector, prior to packing individual elements.

An application may use **operator<<** or **operator>>** for the implementation of the member function **do_pack** and **do_unpack** as part of an application-specific object definition derived from class **uvm_object**.

5.1.6 Data members (variables)

5.1.6.1 physical

```
bool physical;
```

The data member **physical** shall provides a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application may, in the member functions **uvm_object**::**do_pack** and **uvm_object**::**do_unpack**, test the setting of this field, to use it as a filter. By default, the data member **physical** is set to true in the constructor of **uvm_packer**.

5.1.6.2 abstract

```
bool abstract;
```

The data member **abstract** shall provides a filtering mechanism for fields. The abstract and physical settings allow an object to distinguish between two different classes of fields. An application may, in the member functions **uvm_object**::**do_pack** and **uvm_object**::**do_unpack**, test the setting of this field, to use it as a filter. By default, the data member **abstract** is set to false in the constructor of **uvm_packer**.

5.1.6.3 use_metadata

```
bool use_metadata;
```

The data member **use_metadata** shall indicate whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of **uvm_object**::**do_pack** and **uvm_object**::**do_unpack** should regard this bit when performing their respective operation. When set to true, metadata should be encoded as follows:

— For strings, pack an additional NULL byte after the string is packed.

— For objects, pack 4 bits prior to packing the object itself. Use 0b0000 to indicate the object being packed is null, otherwise pack 0b0001 (the remaining 3 bits are reserved).

— For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to to packing individual elements.

By default, **use_metadata** is set to false.

5.1.6.4 big_endian

```
bool big_endian;
```

The data member **big_endian** shall determine the order that integral data is packed (using the member functions **pack_field**, **pack_field_int**, **pack_time**, or **pack_real**) and how the data is unpacked from the pack array (using the member functions **unpack_field**, **unpack_field_int**, **unpack_time**, or **unpack_real**). By default, the data member is set to true in the constructor of **uvm_packer**. When the data member is set, data is associated msb to lsb; otherwise, it is associated lsb to msb.

5.2 uvm_printer

The class **uvm_printer** shall provide the basic printer functionality, which shall be overloaded by derived classes to support various pre-defined printing formats.

5.2.1 Class definition

```
namespace uvm {

  class uvm_printer
  {
   public:
    // Group: Printing types
    virtual void print_field( const std::string& name,
                              const uvm_bitstream_t& value,
                              int size = -1,
                              uvm_radix_enum radix = UVM_NORADIX,
                              const char* scope_separator = ".",
                              const std::string& type_name = "" ) const;

    virtual void print_field_int( const std::string& name,
                                  const uvm_integral_t& value,
                                  int size = -1,
                                  uvm_radix_enum radix = UVM_NORADIX,
                                  const char* scope_separator = ".",
                                  const std::string& type_name = "" ) const;

    virtual void print_real( const std::string& name,
                             double value,
```

```
                              const char* scope_separator = "." ) const;

    virtual void print_real( const std::string& name,
                             float value,
                             const char* scope_separator = "." ) const;

    virtual void print_object( const std::string& name,
                               uvm_object* value,
                               const char* scope_separator = "." ) const;

    virtual void print_object_header( const std::string& name,
                                      uvm_object* value,
                                      const char* scope_separator = "." ) const;

    virtual void print_string( const std::string& name,
                               const std::string& value,
                               const char* scope_separator = "." ) const;

    virtual void print_time( const std::string& name,
                             const sc_core::sc_time& value,
                             const char* scope_separator = "." ) const;

    virtual void print_generic( const std::string& name,
                                const std::string& type_name,
                                int size,
                                const std::string& value,
                                const char* scope_separator = "." ) const;

    // Group: Printer subtyping
    virtual std::string emit();
    virtual std::string format_row( const uvm_printer_row_info& row );
    virtual std::string format_header();
    virtual std::string format_footer();

    std::string adjust_name( const std::string& id,
                             const char* scope_separator = "." ) const;

    virtual void print_array_header( const std::string& name,
                                     int size,
                                     const std::string& arraytype = "array",
                                     const char* scope_separator = "." ) const;

    void print_array_range( int min, int max ) const;
    void print_array_footer( int size = 0 ) const;

    // Data members
    uvm_printer_knobs knobs;

   protected:
    // Disabled
    uvm_printer();

  }; // class uvm_printer

} // namespace uvm
```

5.2.2 Constraints on usage

An application shall not explicitly create an instance of the class **uvm_printer**.

5.2.3 Printing types

5.2.3.1 print_field

```
virtual void print_field( const std::string& name,
                          const uvm_bitstream_t& value,
                          int size = -1,
                          uvm_radix_enum radix = UVM_NORADIX,
                          const char* scope_separator = ".",
                          const std::string& type_name = "" );
```

The member function **print_field** shall print a field of type **uvm_bitstream_t**. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *size* defines the number of bits of the field. The argument radix defined radix to use for printing. The printer knob for radix is used if no radix is specified. The argument *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 "[" (open bracket).

5.2.3.2 print_field_int

```
virtual void print_field_int( const std::string& name,
                              const uvm_integral_t& value,
                              int size = -1,
                              uvm_radix_enum radix = UVM_NORADIX,
                              const char* scope_separator = ".",
                              const std::string& type_name = "" );
```

The member function **print_field_int** shall print an integer field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *size* defines the number of bits of the field. The argument radix defined radix to use for printing. The printer knob for radix is used if no radix is specified. The argument *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 "[" (open bracket).

5.2.3.3 print_real

```
virtual void print_real( const std::string& name,
                         double value,
                         const char* scope_separator = "." );
```

The member function **print_real** shall print a real (double) field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.3.4 print_double

```
virtual void print_double( const std::string& name,
                           double value,
                           const char* scope_separator = "." );
```

The member function **print_double** shall print a real (double) field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

NOTE—This member function has been introduced to be more compatible with C++/SystemC coding styles and types. The member function has identical functionality to **print_real**.

5.2.3.5 print_object

```
virtual void print_object( const std::string& name,
                           const uvm_object& value,
                           const char* scope_separator = "." ) const;
```

The member function **print_object** shall print an object. The argument *name* defines the name of the object. The argument *value* contains the reference to the object. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of the object.

Whether the object is recursed depends on a variety of knobs, such as the depth knob; if the current depth is at or below the depth setting, then the object is not recursed. By default, the children of objects of type

uvm_component are printed. To disable automatic printing of these objects, an application can set the member function **uvm_component::print_enabled** to false for the specific children to be excluded from printing.

5.2.3.6 print_object_header

```
virtual void print_object_header( const std::string& name,
                                  const uvm_object& value,
                                  const char* scope_separator = "." ) const;
```

The member function **print_object_header** shall print an object header. The argument *name* defines the name of the object. The argument *value* contains the reference to the object. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.3.7 print_string

```
virtual void print_string( const std::string& name,
                           const std::string& value,
                           const char* scope_separator = "." );
```

The member function **print_string** shall print a string field. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.3.8 print_time

```
virtual void print_time( const std::string& name,
                         const sc_core::sc_time& value,
                         const char* scope_separator = "." );
```

The member function **print_time** shall print the time. The argument *name* defines the name of the field. The argument *value* contains the value of the field. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.3.9 print_generic

```
virtual void print_generic( const std::string& name,
                            const std::string& type_name,
                            int size,
                            const std::string& value,
                            const char* scope_separator = "." );
```

The member function **print_generic** shall print a field using the arguments *name*, *type_name*, *size*, and *value*. The argument *scope_separator* is used to find the leaf name since many printers only print the leaf name of a field.

5.2.4 Printer subtyping

5.2.4.1 emit

```
virtual std::string emit();
```

The member **emit** shall return a string representing the contents of an object in a format defined by an extension of this object.

5.2.4.2 format_row

```
virtual std::string format_row( const uvm_printer_row_info& row );
```

The member **format_row** shall offer a hook for producing custom output of a single field (row).

5.2.4.3 format_header

```
virtual std::string format_header();
```

The member function **format_header** shall offer a hook to override the base header with a custom header.

5.2.4.4 format_footer

```
virtual std::string format_footer();
```

The member **format_footer** shall offer a hook to override the base footer with a custom footer.

5.2.4.5 adjust_name

```
std::string adjust_name( const std::string& id,
                         const char* scope_separator = "." ) const;
```

The member function **adjust_name** shall print a field's name, or *id* , which is the full instance name. The intent of the separator is to mark where the leaf name starts if the printer is configured to print only the leaf name of the identifier.

5.2.4.6 print_array_header

```
virtual void print_array_header( const std::string& name,
                                 int size,
                                 const std::string& arraytype = "array",
                                 const char* scope_separator = "." ) const;
```

The member function **print_array_header** shall print the header of an array. This member function shall be called before each individual element is printed. The member function **print_array_footer** shall be called to mark the completion of array printing.

5.2.4.7 print_array_range

```
void print_array_range( int min, int max ) const;
```

The member function **print_array_range** shall print a range using ellipses for values. This member function is used when honoring the array knobs for partial printing of large arrays, **uvm_printer_knobs**::**begin_elements** and **uvm_printer_knobs**::**end_elements**. This member function should be called after **uvm_printer_knobs**::**begin_elements** have been printed and before **uvm_printer_knobs**::**end_elements** have been printed.

5.2.4.8 print_array_footer

```
void print_array_footer( int size = 0 ) const;
```

The member function **print_array_footer** shall print the footer of an array. This member function marks the end of an array print. Generally, there is no output associated with the array footer, but this member function lets the printer know that the array printing is complete.

5.2.5 Data members

5.2.5.1 knobs

```
uvm_printer_knobs knobs;
```

The data member **knobs** shall provide access to the variety of knobs associated with a specific printer instance.

5.3 uvm_table_printer

The class **uvm_table_printer** shall provide a pre-defined printing output in a tabular format.

5.3.1 Class definition

```
namespace uvm {

  class uvm_table_printer : public uvm_printer
  {
   public:
    // Constructor
    uvm_table_printer();

    // Member function
    virtual std::string emit();

  }; // class uvm_table_printer

} // namespace uvm
```

5.3.2 Constructor

```
uvm_table_printer();
```

The constructor shall create a new instance of type **uvm_table_printer**.

5.3.3 emit

The member function **emit** shall format the collected information for printing into a table format.

5.4 uvm_tree_printer

The class **uvm_tree_printer** shall provide a pre-defined printing output in a tree format.

5.4.1 Class definition

```
namespace uvm {

  class uvm_tree_printer : public uvm_printer
  {
   public:
    // Constructor
    uvm_tree_printer();
```

```
    // Member function
    virtual std::string emit();

  }; // class uvm_tree_printer

} // namespace uvm
```

5.4.2 Constructor

```
uvm_tree_printer();
```

The constructor shall create a new instance of type **uvm_tree_printer**.

5.4.3 emit

The member function **emit** shall format the collected information for printing into a hierarchical tree format.

5.5 uvm_line_printer

The class **uvm_line_printer** shall provide a pre-defined printing output in a line format.

5.5.1 Class definition

```
namespace uvm {

  class uvm_line_printer : public uvm_printer
  {
   public:
    // Constructor

    uvm_line_printer();

    // Member function
    virtual std::string emit();

  }; // class uvm_line_printer

} // namespace uvm
```

5.5.2 Constructor

```
uvm_line_printer();
```

The constructor shall create a new instance of type **uvm_line_printer**.

5.5.3 emit

The member function **emit** shall format the collected information for printing into a line format, which contains no line-feeds and indentation.

5.6 uvm_comparer

The class **uvm_comparer** shall provide 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 member functions **uvm_object::compare** and **uvm_object::do_compare** are passed a **uvm_comparer** policy object.

5.6.1 Class definition

```
namespace uvm {

  class uvm_comparer
  {
   public:
    // Member functions
    virtual bool compare_field( const std::string& name,
                                const uvm_bitstream_t& lhs,
                                const uvm_bitstream_t& rhs,
                                int size,
                                uvm_radix_enum radix = UVM_NORADIX ) const;

    virtual bool compare_field_int( const std::string& name,
                                    const uvm_integral_t& lhs,
                                    const uvm_integral_t& rhs,
                                    int size,
                                    uvm_radix_enum radix = UVM_NORADIX ) const;

    virtual bool compare_field_real( const std::string& name,
                                     double lhs,
                                     double rhs ) const;

    virtual bool compare_field_real( const std::string& name,
                                     float lhs,
                                     float rhs ) const;

    virtual bool compare_object( const std::string& name,
                                 const uvm_object& lhs,
                                 const uvm_object& rhs ) const;

    virtual bool compare_string( const std::string& name,
                                 const std::string& lhs,
                                 const std::string& rhs ) const;

    void print_msg( const std::string& msg ) const;

    // Group: Comparer settings
    void set_policy( uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY );
    uvm_recursion_policy_enum get_policy() const;
    void set_max_messages( unsigned int num = 1 );
    unsigned int get_max_messages() const;
    void set_verbosity( unsigned int verbosity = UVM_LOW );
    unsigned int get_verbosity() const;
    void set_severity( uvm_severity sev = UVM_INFO );
    uvm_severity get_severity () const;
    void set_miscompare_string( const std::string& miscompares = "" );
    std::string get_miscompare_string() const;
    void set_field_attribute( uvm_field_enum attr = UVM_PHYSICAL );
    uvm_field_enum get_field_attribute() const;
    void compare_type( bool enable = true );
    unsigned int get_result() const;

   private:
    // Disabled
    uvm_comparer();

  }; // class uvm_comparer

} // namespace uvm
```

5.6.2 Constraints on usage

An application shall not explicitly create an instance of the class **uvm_comparer**.

5.6.3 Member functions

5.6.3.1 compare_field

```
virtual bool compare_field( const std::string& name,
                            const uvm_bitstream_t& lhs,
                            const uvm_bitstream_t& rhs,
                            int size,
                            uvm_radix_enum radix = UVM_NORADIX ) const;
```

The member function **compare_field** shall compare two integral values. The argument *name* is used for purposes of storing and printing a miscompare. The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison. The argument *size* indicates the number of bits to compare. *size* shall be less than or equal to **UVM_MAX_STREAMBITS**. The argument *radix* is used for reporting purposes, the default radix is hex.

5.6.3.2 compare_field_int

```
virtual bool compare_field_int( const std::string& name,
                                const uvm_integral_t& lhs,
                                const uvm_integral_t& rhs,
                                int size,
                                uvm_radix_enum radix = UVM_NORADIX ) const;
```

The member function **compare_field_int** shall compare two integral values. This member function is same as **compare_field** except that the arguments are small integers, less than or equal to 64 bits. It is automatically called by **compare_field** if the operand size is less than or equal to 64.

The argument *name* is used for purposes of storing and printing a miscompare. The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison. The argument *size* indicates the number of bits to compare. *size* shall be less than or equal to 64. The argument *radix* is used for reporting purposes, the default radix is hex.

5.6.3.3 compare_field_real

```
virtual bool compare_field_real( const std::string& name,
                                 double lhs,
                                 double rhs ) const;

virtual bool compare_field_real( const std::string& name,
                                 float lhs,
                                 float rhs ) const;
```

The member function **compare_field_real** shall compare two real numbers, represented by type double or float, respetively. The left-hand-side *lhs* and right-hand-side *rhs* arguments are used for comparison.

5.6.3.4 compare_object

```
virtual bool compare_object( const std::string& name,
                             const uvm_object& lhs,
                             const uvm_object& rhs ) const;
```

The member function **compare_object** shall compare two class objects using the data member *policy* to determine whether the comparison should be deep, shallow, or reference. The argument *name* is used for purposes of storing and printing a miscompare. The *lhs* and *rhs* objects are the two objects used for comparison. The data member *check_type* determines whether or not to verify the object types match (the return from *lhs*.**get_type_name**() matches *rhs*.**get_type_name**()).

5.6.3.5 compare_string

```
virtual bool compare_string( const std::string& name,
                             const std::string& lhs,
                             const std::string& rhs ) const;
```

The member function **compare_string** shall compare two two string variables. The argument *name* is used for purposes of storing and printing a miscompare. The *lhs* and *rhs* objects are the two objects used for comparison.

5.6.3.6 print_msg

```
void print_msg( const std::string& msg ) const;
```

The member function **print_msg** shall cause the error count to be incremented and the message passed as argument to be appended to the miscompares string (a newline is used to separate messages). If the message count is less than the data member *show_max* setting, then the message is printed to standard-out using the current verbosity (see Section 5.6.4.5) and severity (see Section 5.6.4.7) settings.

5.6.4 Comparer settings

5.6.4.1 set_policy

```
void set_policy( uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY );
```

The member function **set_policy** shall set the comparison policy. The following arguments are valid: **UVM_DEEP**, **UVM_REFERENCE**, or **UVM_SHALLOW**. The default policy shall be set to **UVM_DEFAULT_POLICY**.

5.6.4.2 get_policy

```
uvm_recursion_policy_enum get_policy() const;
```

The member function **get_policy** shall return the comparison policy.

5.6.4.3 set_max_messages

```
void set_max_messages( unsigned int num = 1 );
```

The member function **set_max_messages** sets the maximum number of messages to send to the printer for miscompares of an object. The default number of messages shall be set to one.

5.6.4.4 get_max_messages

```
unsigned int get_max_messages() const;
```

The member function **get_max_messages** shall return the maximum number of messages to send to the printer for miscompares of an object.

5.6.4.5 set_verbosity

```
void set_verbosity( unsigned int verbosity = UVM_LOW );
```

The member function **set_verbosity** shall set the verbosity for printed messages. The verbosity setting is used by the messaging mechanism to determine whether messages should be suppressed or shown. The default verbosity shall be set to **UVM_LOW**.

5.6.4.6 get_verbosity

```
unsigned int get_verbosity() const;
```

The member function **get_verbosity** shall return the verbosity for printed messages.

5.6.4.7 set_severity

```
void set_severity( uvm_severity sev = UVM_INFO);
```

The member function **set_severity** shall set the severity for printed messages. The severity setting is used by the messaging mechanism for printing and filtering messages. The default severity shall be set to **UVM_INFO**.

5.6.4.8 get_severity

```
uvm_severity get_severity() const;
```

The member function **get_severity** shall return the severity for printed messages.

5.6.4.9 set_miscompare_string

```
void set_miscompare_string( const std::string& miscompares = "" );
```

The member function **set_miscompare_string** shall set the miscompare string. This string is reset to an empty string when a comparison is started. The string holds the last set of miscompares that occurred during a comparison. The default miscompare string shall be empty.

5.6.4.10 get_miscompare_string

```
std::string get_miscompare_string() const;
```

The member function **get_miscompare_string** shall return the last set of miscompares that occurred during a comparison.

5.6.4.11 set_field_attribute

```
void set_field_attribute( uvm_field_enum attr = UVM_PHYSICAL );
```

The member function **set_field_attribute** shall set the field attribute to **UVM_PHYSICAL** or **UVM_ABSTRACT**. The physical and abstract settings allow an object to distinguish between these two different classes of fields.

NOTE—An application can use the callback **uvm_object::do_compare** to check the field attribute if it wants to use it as a filter.

5.6.4.12 get_field_attribute

```
uvm_field_enum get_field_attribute() const;
```

The member function **get_field_attribute** shall return the field attribute being **UVM_PHYSICAL** or **UVM_ABSTRACT**.

5.6.4.13 compare_type

```
void compare_type( bool enable = true );
```

The member function **compare_type** shall determine whether the type, given by **uvm_object**::**get_type_name**, is used to verify that the types of two objects are the same. If enabled, the member function **compare_object** is called. By default, type checking shall be enabled.

NOTE—In some cases an application may disable type checking, when the two operands are related by inheritance but are of different types.

5.6.4.14 get_result

```
unsigned int get_result() const;
```

The member function **get_result** shall return the number of miscompares for a given compare operation. An application can use the result to determine the number of miscompares that were found.

5.7 Default policy objects

5.7.1 uvm_default_table_printer

```
extern uvm_table_printer* uvm_default_table_printer;
```

The global object **uvm_default_table_printer** shall define a handle to an object of type **uvm_table_printer**, which can be used with **uvm_object**::**do_print** to get tabular style printing.

5.7.2 uvm_default_tree_printer

```
extern uvm_tree_printer* uvm_default_tree_printer;
```

The global object **uvm_default_tree_printer** shall define a handle to an object of type **uvm_tree_printer**, which can be used with **uvm_object**::**do_print** to get a multi-line tree style printing.

5.7.3 uvm_default_line_printer

```
extern uvm_line_printer* uvm_default_line_printer;
```

The global object **uvm_default_line_printer** shall define a handle to an object of type **uvm_line_printer**, which can be used with **uvm_object**::**do_print** to get a single-line style printing.

5.7.4 uvm_default_printer

```
extern uvm_printer* uvm_default_printer;
```

The global object **uvm_default_printer** shall define the default printer policy, which shall be set to **uvm_default_table_printer.** An application can redefine the default printer, by setting it to any legal **uvm_printer** derived type, including the global line, tree, and table printers in the previous sections.

5.7.5 uvm_default_packer

```
extern uvm_printer* uvm_default_packer;
```

The global object **uvm_default_packer** shall define the default packer policy. It shall be used when calls to **uvm_object**::**pack** and **uvm_object**::**unpack** do not specify a packer policy.

5.7.6 uvm_default_comparer

```
extern uvm_comparer* uvm_default_comparer;
```

The global object **uvm_default_comparer** shall define the default comparer policy. It shall be used when calls to **uvm_object**::**compare** do not specify a comparer policy.

5.7.7 uvm_default_recorder

```
extern uvm_recorder* uvm_default_recorder;
```

The global object **uvm_default_recorder** shall define the default recorder policy. It shall be used when calls to **uvm_object**::**record** do not specify a recorder policy.

6. Registry and factory classes

The registry and factory classes offer the interface to register and use UVM objects and components via the factory.

The following classes are defined:

— **uvm_object_wrapper**
— **uvm_object_registry**
— **uvm_component_registry**
— **uvm_factory**
— **uvm_default_factory**

The class **uvm_object_wrapper** forms the base class for the registry classes **uvm_object_registry** and **uvm_component_registry**, which act as lightweight proxies for UVM objects and components, respectively.

UVM object and component types are registered with the factory via typedef or macro invocation. When the application requests a new object or component from the factory, the factory determines what type of object to create based on its configuration, and asks that type's proxy to create an instance of the type, which is returned to the application.

6.1 uvm_object_wrapper

The class **uvm_object_wrapper** shall provide an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every object or component derived from **uvm_object** or **uvm_component** respectively in the test environment, are registered with the **uvm_factory**. When the factory is called upon to create an object or component, it shall find and delegate the request to the appropriate proxy.

6.1.1 Class definition

```
namespace uvm {

  class uvm_object_wrapper
  {
   public:
    virtual uvm_object* create_object( const std::string& name = "" );
    virtual uvm_component* create_component( const std::string& name,
                                             uvm_component* parent );
    virtual const std::string get_type_name() const = 0;
  };

} // namespace uvm
```

6.1.2 Member functions

6.1.2.1 create_object

```
virtual uvm_object* create_object( const std::string& name = "" );
```

The member function **create_object** shall create a new object with the optional name passed as argument. An object proxy (e.g., **uvm_object_registry**<T>) implements this member function to create an object of a specific type, T (see Section 6.2).

6.1.2.2 create_component

```
virtual uvm_component* create_component( const std::string& name,
                                         uvm_component* parent );
```

The member function **create_component** shall create a new component, by passing to its constructor the given name and parent. The component proxy (e.g. **uvm_component_registry**<T>) implements this member function to create a component of a specific type, T (see Section 6.3).

6.1.2.3 get_type_name

```
virtual const std::string get_type_name() const = 0;
```

The implementation of the pure virtual member function **get_type_name** shall return the type name of the object created by **create_component** or **create_object**. The factory uses this name when matching against the requested type in name-based lookups.

6.2 uvm_object_registry

The class **uvm_object_registry** shall provide a lightweight proxy for a **uvm_object** of type T. The proxy enables efficient registration with the **uvm_factory**. Without it, registration would require an instance of the object itself.

The macros **UVM_OBJECT_UTILS** or **UVM_OBJECT_PARAM_UTILS** shall create the appropriate class **uvm_object_registry** necessary to register that particular object with the factory.

6.2.1 Class definition

```
namespace uvm {

  template <typename T = uvm_object>
  class uvm_object_registry<T> : public uvm_object_wrapper
  {
   public:
    virtual uvm_object* create_object( const std::string& name = "" );
    virtual const std::string get_type_name() const;
    static uvm_object_registry<T>* get();

    static T* create( const std::string& name = "",
                      uvm_component* parent = NULL,
                      const std::string& contxt = "" );

    static void destroy$( T* obj );

    static void set_type_override( uvm_object_wrapper* override_type,
                                   bool replace = true );

    static void set_inst_override( uvm_object_wrapper* override_type,
                                   const std::string& inst_path,
                                   uvm_component* parent = NULL );
  }; // class uvm_object_registry

} // namespace uvm
```

6.2.2 Template parameter T

The template parameter T specifies the object type of the objects being registered. The object type shall be a derivative of class **uvm_object**.

6.2.3 Member functions

6.2.3.1 create_object

```
virtual uvm_object* create_object( const std::string& name = "" );
```

The member function **create_object** shall create an object of type T and returns it as a handle to a **uvm_object**. This is an overload of the member function in **uvm_object_wrapper**. It is called by the factory after determining the type of object to create. An application shall not call this member function directly. Instead, an application shall call the static member function **create**.

6.2.3.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object. This member function overloads the member function in **uvm_object_wrapper**.

6.2.3.3 get

```
static uvm_object_registry<T>* get();
```

The member function **get** shall return the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

6.2.3.4 create

```
static T* create( const std::string& name = "",
                  uvm_component* parent = NULL,
                  const std::string& contxt = "" );
```

The member function **create** shall return a new 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 new instance shall have the given leaf name *name*, if provided as argument. The argument *contxt*, if supplied, supersedes the parent's context.

6.2.3.5 destroy§

```
static void destroy§( T* obj );
```

The member function **destroy** shall remove the object given as argument from the UVM object registry and deallocates its memory location. A warning shall be generated if the object does not exist in the registry.

NOTE—An application should always call the static member function **destroy** when using the static member function **create** to avoid memory leakage.

6.2.3.6 set_type_override

```
static void set_type_override( uvm_object_wrapper* override_type,
                               bool replace = true );
```

The member function **set_type_override** shall configure 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.

When argument *replace* is set to true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.2.3.7 set_inst_override

```
static void set_inst_override( uvm_object_wrapper* override_type,
                               const std::string& inst_path,
                               uvm_component* parent = NULL );
```

The member function **set_inst_override** shall configure the factory to create an object of the type represented by argument *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 argument *parent* is not specified, argument *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If argument *parent* is specified, argument *inst_path* is interpreted as being relative to the parent's hierarchical instance path. The argument *inst_path* may contain wildcards for matching against multiple contexts.

6.3 uvm_component_registry

The class **uvm_component_registry** shall provide a lightweight proxy for a **uvm_component** of type T. The proxy enables efficient registration with the **uvm_factory**. Without it, registration would require an instance of the component itself.

The macros **UVM_COMPONENT_UTILS** and **UVM_COMPONENT_PARAM_UTILS** shall create the appropriate class **uvm_component_registry** necessary to register that particular component with the factory.

6.3.1 Class definition

```
namespace uvm {

  template <typename T = uvm_component>
  class uvm_component_registry : public uvm_object_wrapper
  {
   public:
    virtual uvm_component* create_component( const std::string& name,
                                             uvm_component* parent );

    virtual const std::string get_type_name() const;
    static uvm_component_registry<T>* get();

    static T* create( const std::string& name = "",
                      uvm_component* parent = NULL,
                      const std::string& contxt = "" );

    static void destroy§( T* obj );

    static void set_type_override( uvm_object_wrapper* override_type,
                                   bool replace = true );

    static void set_inst_override( uvm_object_wrapper* override_type,
                                   const std::string& inst_path,
                                   uvm_component* parent = NULL );
  }; // class uvm_component_registry

} // namespace uvm
```

6.3.2 Template parameter T

The template parameter T specifies the object type of the components being registered. The object type shall be a derivative of class **uvm_component**.

6.3.3 Member functions

6.3.3.1 create_component

```
virtual uvm_component* create_component( const std::string& name,
                                          uvm_component* parent );
```

The member function **create_component** shall create an object of type T having the provided *name* and *parent*, and returns it as a handle to a **uvm_component**. This is an overload of the member function in **uvm_object_wrapper**. It is called by the factory after determining the type of component to create. An application shall not call this member function directly. Instead, an application shall call the static member function **create**.

6.3.3.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component. This member function overloads the member function in **uvm_object_wrapper**.

6.3.3.3 get

```
static uvm_component_registry<T>* get();
```

The member function **get** shall return the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

6.3.3.4 create

```
static T* create( const std::string& name = "",
                  uvm_component* parent = NULL,
                  const std::string& contxt = "" );
```

The member function **create** shall return a new instance of the component type, T, represented by this proxy, subject to any factory overrides based on the context provided by the parent’s full name. The new instance shall have the given leaf name *name*, if provided as argument. The argument *contxt*, if supplied, supersedes the parent’s context.

6.3.3.5 destroy§

```
static void destroy§( T* obj );
```

The member function **destroy** shall remove the object given as argument from the UVM component registry and deallocates its memory location. A warning shall be generated if the component does not exist in the registry.

NOTE—An application should always call the static member function **destroy** when using the static member function **create** to avoid memory leakage.

6.3.3.6 set_type_override

```
static void set_type_override( uvm_object_wrapper* override_type,
                               bool replace = true );
```

The member function **set_type_override** shall configure the factory to create a component of the type represented by argument *override_type* whenever a request is made to create a component of the type represented by this proxy, provided no instance override applies. The override type shall be derived from the original type, T.

When argument *replace* is set to true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.3.3.7 set_inst_override

```
static void set_inst_override( uvm_object_wrapper* override_type,
                               const std::string& inst_path,
                               uvm_component* parent = NULL );
```

The member function **set_inst_override** shall configure the factory to create a component of the type represented by argument *override_type* whenever a request is made to create a component of the type represented by this proxy, with matching instance paths. The override type shall be derived from the original type, T.

If argument *parent* is not specified, argument *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If argument *parent* is specified, argument *inst_path* is interpreted as being relative to the parent's hierarchical instance path. The argument *inst_path* may contain wildcards for matching against multiple contexts.

6.4 uvm_factory

The class **uvm_factory** implements a factory pattern. A singleton factory instance is created for a given simulation run. Object and component types are registered with the factory using proxies to the actual objects and components being created. The classes **uvm_object_registry**<T> and **uvm_component_registry**<T> are used to proxy objects of type **uvm_object** and **uvm_component** respectively. These registry classes both use the **uvm_object_wrapper** as abstract base class.

6.4.1 Class definition

```
namespace uvm {

  class uvm_factory {
   public:

    // Group: Access and registration

    static uvm_factory* get();

    void do_register°( uvm_object_wrapper* obj ) = 0;

    // Group: Type & instance overrides

    virtual void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                            uvm_object_wrapper* override_type,
                                            const std::string& full_inst_path ) = 0;

    virtual void set_inst_override_by_name( const std::string& original_type_name,
                                            const std::string& override_type_name,
                                            const std::string& full_inst_path ) = 0;
```

```
    virtual void set_type_override_by_type( uvm_object_wrapper* original_type,
                                            uvm_object_wrapper* override_type,
                                            bool replace = true ) = 0;

    virtual void set_type_override_by_name( const std::string& original_type_name,
                                            const std::string& override_type_name,
                                            bool replace = true ) = 0;

    // Group: Creation

    virtual uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
                                               const std::string& parent_inst_path = "",
                                               const std::string& name = "" ) = 0;

    virtual uvm_object* create_object_by_name( const std::string& requested_type_name,
                                               const std::string& parent_inst_path = "",
                                               const std::string& name = "" ) = 0;

    virtual uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                                     const std::string& parent_inst_path = "",
                                                     const std::string& name = "",
                                                     uvm_component* parent = NULL ) = 0;

    virtual uvm_component* create_component_by_name( const std::string& requested_type_name,
                                                     const std::string& parent_inst_path = "",
                                                     const std::string& name = "",
                                                     uvm_component* parent = NULL ) = 0;

    // Group: Debug

    virtual void debug_create_by_type( uvm_object_wrapper* requested_type,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" ) = 0;

    virtual void debug_create_by_name( const std::string& requested_type_name,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" ) = 0;

    virtual uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                                       const std::string& full_inst_path ) = 0;

    virtual uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                                       const std::string& full_inst_path ) = 0;

    virtual void print( int all_types = 1 ) = 0;

  }; // class uvm_factory

} // namespace uvm
```

6.4.2 Access and registration

6.4.2.1 get

```
static uvm_factory* get();
```

The member function **get** shall return this**uvm_factory**.

6.4.2.2 do_register° (register†)

```
virtual void do_register°( uvm_object_wrapper* obj ) = 0;
```

The member function **do_register°** shall 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 member function **create_object** or **create_component** to do so.

When doing name-based operations, the factory calls the proxy's member function **get_type_name** to match against the argument **requested_type_name** in subsequent calls to **create_component_by_name** and **create_object_by_name**. If the proxy object's member function **get_type_name** returns the empty string, name-based lookup is effectively disabled.

NOTE—An application needs to invoke the macros **UVM_OBJECT_UTILS**, **UVM_OBJECT_PARAM_UTILS**, **UVM_COMPONENT_UTILS**, or **UVM_COMPONENT_PARAM_UTILS** to register a particular object or component respectively with the factory.

6.4.3 Type and instance overrides

6.4.3.1 set_inst_override_by_type

```
virtual void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                        uvm_object_wrapper* override_type,
                                        const std::string& full_inst_path ) = 0;
```

The member function **set_inst_override_by_type** shall 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*. The override type shall be derived from the original type, T.

Both the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

The argument *full_inst_path* is matched against the concatenation of parent instance path and name (*parent_inst_path.name*) provided in future create requests. The argument *full_inst_path* may include wildcards ('*' and '?') such that a single instance override can be applied in multiple contexts. An argument *full_inst_path* of '*' is effectively a type override, as it matches all contexts.

When the factory processes instance overrides, the instance queue is processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides.

6.4.3.2 set_inst_override_by_name

```
virtual void set_inst_override_by_name( const std::string& original_type_name,
                                        const std::string& override_type_name,
                                        const std::string& full_inst_path ) = 0;
```

The member function **set_inst_override_by_name** shall 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*. The original type is typically a super class of the override type.

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 member functions **create_object_by_type**, **create_object_by_name**, **create_component_by_type** or **create_component_by_name** with the same string and matching instance path shall produce the type represented by *override_type_name*, which shall be preregistered with the factory.

The argument *full_inst_path* is matched against the concatenation of parent instance path and name (*parent_inst_path.name*) provided in future create requests. The argument *full_inst_path* may include wildcards ('*' and '?') such that a single instance override can be applied in multiple contexts. An argument *full_inst_path* of '*' is effectively a type override, as it matches all contexts.

When the factory processes instance overrides, the instance queue is processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides.

6.4.3.3 set_type_override_by_type

```
virtual void set_type_override_by_type( uvm_object_wrapper* original_type,
                                        uvm_object_wrapper* override_type,
                                        bool replace = true ) = 0;
```

The member function **set_type_override_by_type** shall 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 override type shall be derived from the original type, T.

Both the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When argument *replace* is set to true, a previous override on *original_type* is replaced, otherwise a previous override, if any, remains intact.

6.4.3.4 set_type_override_by_name

```
virtual void set_type_override_by_name( const std::string& original_type_name,
                                        const std::string& override_type_name,
                                        bool replace = true ) = 0;
```

The member function **set_type_override_by_name** shall 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 override type shall be derived from the original type, T.

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 member functions **create_object_by_type**, **create_object_by_name**, **create_component_by_type** or **create_component_by_name** with the same string and matching instance path shall produce the type represented by *override_type_name*, which shall be preregistered with the factory.

When argument *replace* is set to true, a previous override on *original_type_name* is replaced, otherwise a previous override, if any, remains intact.

6.4.4 Creation

6.4.4.1 create_object_by_type

```
virtual uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
                                           const std::string& parent_inst_path = "",
                                           const std::string& name = "" ) = 0;
```

The member function **create_object_by_type** shall create and return an object of the requested type, which is specified by argument *requested_type*. A requested object shall be derived from the base class **uvm_object**.

The argument *parent_inst_path* is an optional hierarchical anchor for the object being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. Newly created object shall have the given *name*, if provided.

6.4.4.2 create_object_by_name

```
virtual uvm_object* create_object_by_name( const std::string& requested_type_name,
                                           const std::string& parent_inst_path = "",
```

```
                                          const std::string& name = "" ) = 0;
```

The member function **create_object_by_name** shall create and return an object of the requested type, which is specified by argument *requested_type_name*. The requested type shall be registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error is produced and the member function shall return NULL. A requested object shall be derived from the base class **uvm_object**.

The argument *parent_inst_path* is an optional hierarchical anchor for the object being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. If no instance override is found, the factory then searches for a type override. Newly created object shall have the given *name*, if provided.

NOTE—The convenience function **create_object** is available in the class **uvm_component** for the creation of an object (see Section 7.1.8.2). Alternatively, an application can create an object by using the static member function **create** via the **uvm_object_registry**, which is made available via the macro **UVM_OBJECT_UTILS** or **UVM_OBJECT_PARAM_UTILS**.

6.4.4.3 create_component_by_type

```
virtual uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                                  const std::string& parent_inst_path = "",
                                                  const std::string& name = "",
                                                  uvm_component* parent = NULL ) = 0;
```

The member function **create_component_by_type** shall create and return a component of the requested type, which is specified by argument *requested_type*. A requested component shall be derived from the base class **uvm_component**.

The argument *parent_inst_path* is an optional hierarchical anchor for the component being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. Newly created components shall have the given *name* and *parent*.

6.4.4.4 create_component_by_name

```
virtual uvm_component* create_component_by_name( const std::string& requested_type_name,
                                                  const std::string& parent_inst_path = "",
                                                  const std::string& name = "",
                                                  uvm_component* parent = NULL ) = 0;
```

The member function **create_component_by_name** shall create and return a component of the requested type, which is specified by argument *requested_type_name*. The requested type shall be registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error is produced and the member function shall return NULL. A requested component shall be derived from the base class **uvm_component**.

The argument *parent_inst_path* is an optional hierarchical anchor for the component being created. If this argument is provided, then the concatenation, *parent_inst_path.name*, forms the instance path (context) that is used to search for an instance override. If no instance override is found, the factory then searches for a type override. Newly created components shall have the given *name* and *parent*.

NOTE—The convenience function **create_component** is available in the class **uvm_component** for the creation of a component (see Section 7.1.8.1). Alternatively, an application can create an object by using the static member function **create** via the **uvm_component_registry** which is made available via the macro **UVM_COMPONENT_UTILS** or **UVM_COMPONENT_PARAM_UTILS**.

6.4.5 Debug

6.4.5.1 debug_create_by_type

```
virtual void debug_create_by_type( uvm_object_wrapper* requested_type,
                                   const std::string& parent_inst_path = "",
                                   const std::string& name = "" ) = 0;
```

The member function **debug_create_by_type** shall perform the same search algorithm as the member function **create_object_by_type**, but it shall not create a new object. Instead, it provides detailed information about what type of object it would return, listing each override that was applied to arrive at the result. Interpretation of the arguments are exactly as with the member function **create_object_by_type**.

6.4.5.2 debug_create_by_name

```
virtual void debug_create_by_name( const std::string& requested_type_name,
                                   const std::string& parent_inst_path = "",
                                   const std::string& name = "" ) = 0;
```

The member function **debug_create_by_name** shall perform the same search algorithm as the member function **create_object_by_name**, but it shall not create a new object. Instead, it provides detailed information about what type of object it would return, listing each override that was applied to arrive at the result. Interpretation of the arguments are exactly as with the member function **create_object_by_name**.

6.4.5.3 find_override_by_type

```
virtual uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                                   const std::string& full_inst_path ) = 0;
```

The member function **find_override_by_type** shall return the proxy to the object that would be created given the arguments. The argument *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created.

6.4.5.4 find_override_by_name

```
virtual uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                                   const std::string& full_inst_path ) = 0;
```

The member function **find_override_by_name** shall return the proxy to the object that would be created given the arguments. The argument *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created.

6.4.5.5 print

```
virtual void print( int all_types = 1 ) = 0;
```

The member function **print** shall print the state of the **uvm_factory**, including registered types, instance overrides, and type overrides.

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

6.5 uvm_default_factory

The class **uvm_default_factory** shall provide the default implementation of the UVM factory.

6.5.1 Class definition

```
namespace uvm {

  class uvm_default_factory : public uvm_factory
  {
   public:

    // Group: Registration

    virtual void do_register°( uvm_object_wrapper* obj );

    // Group: Type & instance overrides

    virtual void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                            uvm_object_wrapper* override_type,
                                            const std::string& full_inst_path );

    virtual void set_inst_override_by_name( const std::string& original_type_name,
                                            const std::string& override_type_name,
                                            const std::string& full_inst_path );

    virtual void set_type_override_by_type( uvm_object_wrapper* original_type,
                                            uvm_object_wrapper* override_type,
                                            bool replace = true );

    virtual void set_type_override_by_name( const std::string& original_type_name,
                                            const std::string& override_type_name,
                                            bool replace = true );

    // Group: Creation

    virtual uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
                                               const std::string& parent_inst_path = "",
                                               const std::string& name = "" );

    virtual uvm_object* create_object_by_name( const std::string& requested_type_name,
                                               const std::string& parent_inst_path = "",
                                               const std::string& name = "" );

    virtual uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                                     const std::string& parent_inst_path = "",
                                                     const std::string& name = "",
                                                     uvm_component* parent = NULL );

    virtual uvm_component* create_component_by_name( const std::string& requested_type_name,
                                                     const std::string& parent_inst_path = "",
                                                     const std::string& name = "",
                                                     uvm_component* parent = NULL );

    // Group: Debug

    virtual void debug_create_by_type( uvm_object_wrapper* requested_type,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" );

    virtual void debug_create_by_name( const std::string& requested_type_name,
                                       const std::string& parent_inst_path = "",
                                       const std::string& name = "" );

    virtual uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                                       const std::string& full_inst_path );

    virtual uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                                       const std::string& full_inst_path );

    virtual void print( int all_types = 1 );

  }; // class uvm_default_factory
```

```
} // namespace uvm
```

6.5.2 Registration

6.5.2.1 do_register° (register†)

```
virtual void do_register°( uvm_object_wrapper* obj );
```

The member function **do_register°** shall register the given proxy object, *obj*, with the factory.

6.5.3 Type and instance overrides

6.5.3.1 set_inst_override_by_type

```
virtual void set_inst_override_by_type( uvm_object_wrapper* original_type,
                                        uvm_object_wrapper* override_type,
                                        const std::string& full_inst_path );
```

The member function **set_inst_override_by_type** shall 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*.

6.5.3.2 set_inst_override_by_name

```
virtual void set_inst_override_by_name( const std::string& original_type_name,
                                        const std::string& override_type_name,
                                        const std::string& full_inst_path );
```

The member function **set_inst_override_by_name** shall 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*.

6.5.3.3 set_type_override_by_type

```
virtual void set_type_override_by_type( uvm_object_wrapper* original_type,
                                        uvm_object_wrapper* override_type,
                                        bool replace = true );
```

The member function **set_type_override_by_type** shall 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.

6.5.3.4 set_type_override_by_name

```
virtual void set_type_override_by_name( const std::string& original_type_name,
                                        const std::string& override_type_name,
                                        bool replace = true );
```

The member function **set_type_override_by_name** shall 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.

6.5.4 Creation

6.5.4.1 create_object_by_type

```
virtual uvm_object* create_object_by_type( uvm_object_wrapper* requested_type,
                                           const std::string& parent_inst_path = "",
                                           const std::string& name = "" );
```

The member function **create_object_by_type** shall create and return an object of the requested type, specified by type.

6.5.4.2 create_object_by_name

```
virtual uvm_object* create_object_by_name( const std::string& requested_type_name,
                                           const std::string& parent_inst_path = "",
                                           const std::string& name = "" );
```

The member function **create_object_by_name** shall create and return an object of the requested type, specified by name.

6.5.4.3 create_component_by_type

```
virtual uvm_component* create_component_by_type( uvm_object_wrapper* requested_type,
                                                 const std::string& parent_inst_path = "",
                                                 const std::string& name = "",
                                                 uvm_component* parent = NULL );
```

The member function **create_component_by_type** shall create and return a component of the requested type, specified by type.

6.5.4.4 create_component_by_name

```
virtual uvm_component* create_component_by_name( const std::string& requested_type_name,
                                                 const std::string& parent_inst_path = "",
                                                 const std::string& name = "",
                                                 uvm_component* parent = NULL );
```

The member function **create_component_by_name** shall create and return a component of the requested type, specified by name.

6.5.5 Debug

6.5.5.1 debug_create_by_type

```
virtual void debug_create_by_type( uvm_object_wrapper* requested_type,
                                   const std::string& parent_inst_path = "",
                                   const std::string& name = "" );
```

The member function **debug_create_by_type** shall perform the same search algorithm as the member function **create_object_by_type**, but it shall not create a new object.

6.5.5.2 debug_create_by_name

```
virtual void debug_create_by_name( const std::string& requested_type_name,
                                   const std::string& parent_inst_path = "",
                                   const std::string& name = "" );
```

The member function **debug_create_by_name** shall perform the same search algorithm as the member function **create_object_by_name**, but it shall not create a new object.

6.5.5.3 find_override_by_type

```
virtual uvm_object_wrapper* find_override_by_type( uvm_object_wrapper* requested_type,
                                                   const std::string& full_inst_path );
```

The member function **find_override_by_type** shall return the proxy to the object that would be created given the arguments.

6.5.5.4 find_override_by_name

```
virtual uvm_object_wrapper* find_override_by_name( const std::string& requested_type_name,
                                                   const std::string& full_inst_path );
```

The member function **find_override_by_name** shall return the proxy to the object that would be created given the arguments.

6.5.5.5 print

```
virtual void print( int all_types = 1 );
```

The member function **print** shall print the state of the **uvm_factory**, including registered types, instance overrides, and type overrides.

7. Component hierarchy classes

The UVM components form the foundation of the UVM. They are used to assemble the actual verification environment in a hierarchical and modular fashion, offering a basic set of building blocks such as sequencers, drivers, monitors, scoreboards, and other components. The UVM class library provides a set of predefined component types, all derived directly or indirectly from class **uvm_component**. The following classes are defined:

— **uvm_component**

— **uvm_agent**

— **uvm_driver**

— **uvm_monitor**

— **uvm_env**

— **uvm_scoreboard**

— **uvm_subscriber**

— **uvm_test**

— **uvm_sequencer** (see Chapter 8)

7.1 uvm_component

The class **uvm_component** is the root base class for all structural elements. It provides interfaces for:

— *Hierarchy*: lookup child components

— *Phasing*: pre-run phases, run phase, and post-run phases

— *Factory*: convenience interface to **uvm_factory**

— *Process control*: to suspend and resume processes

— *Objection*: to handle raised and dropped objections

— *Reporting*: hierarchical reporting of messages

— *Recording*: transaction recording

7.1.1 Class definition

```
namespace uvm {

  class uvm_component : public sc_core::sc_module,
                        public uvm_report_object
  {
   public:

    // Constructor
    explicit uvm_component( uvm_component_name name );

    // Group: Hierarchy Interface
    virtual uvm_component* get_parent() const;
    virtual const std::string get_full_name() const;
    void get_children( std::vector<uvm_component*>& children ) const;
    uvm_component* get_child( const std::string& name ) const;
    int get_next_child( std::string& name ) const;
    int get_first_child( std::string& name ) const;
    int get_num_children() const;
    bool has_child( const std::string& name ) const;
    uvm_component* lookup( const std::string& name ) const;
    unsigned int get_depth() const;

    // Group: Phasing Interface
    virtual void build_phase( uvm_phase& phase );
    virtual void connect_phase( uvm_phase& phase );
```

```
virtual void end_of_elaboration_phase( uvm_phase& phase );
virtual void start_of_simulation_phase( uvm_phase& phase );
virtual void run_phase( uvm_phase& phase );
virtual void pre_reset_phase( uvm_phase& phase );
virtual void reset_phase( uvm_phase& phase );
virtual void post_reset_phase( uvm_phase& phase );
virtual void pre_configure_phase( uvm_phase& phase );
virtual void configure_phase( uvm_phase& phase );
virtual void post_configure_phase( uvm_phase& phase );
virtual void pre_main_phase( uvm_phase& phase );
virtual void main_phase( uvm_phase& phase );
virtual void post_main_phase( uvm_phase& phase );
virtual void pre_shutdown_phase( uvm_phase& phase );
virtual void shutdown_phase( uvm_phase& phase );
virtual void post_shutdown_phase( uvm_phase& phase );
virtual void extract_phase( uvm_phase& phase );
virtual void check_phase( uvm_phase& phase );
virtual void report_phase( uvm_phase& phase );
virtual void final_phase( uvm_phase& phase );
virtual void phase_started( uvm_phase& phase );
virtual void phase_ready_to_end( uvm_phase& phase );
virtual void phase_ended( uvm_phase& phase );
void set_domain( uvm_domain* domain, int hier = 1 );
uvm_domain* get_domain() const;
void define_domain( uvm_domain* domain );
void set_phase_imp( uvm_phase* phase, uvm_phase* imp, int hier = 1 );

// Group: Process control interface
virtual bool suspend();
virtual bool resume();

// Group: Configuration Interface
void print_config( bool recurse = false, bool audit = false ) const;
void print_config_with_audit( bool recurse = false ) const;
void print_config_matches( bool enable = true );

// Group: Objection Interface
virtual void raised( uvm_objection* objection,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );

virtual void dropped( uvm_objection* objection,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );

virtual void all_dropped( uvm_objection* objection,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );

// Group: Factory Interface
uvm_component* create_component( const std::string& requested_type_name,
                                 const std::string& name );

uvm_object* create_object( const std::string& requested_type_name,
                           const std::string& name );

static void set_type_override_by_type( uvm_object_wrapper* original_type,
                                       uvm_object_wrapper* override_type,
                                       bool replace = true );

void set_inst_override_by_type( const std::string& relative_inst_path,
                                uvm_object_wrapper* original_type,
                                uvm_object_wrapper* override_type );

static void set_type_override( const std::string& original_type_name,
                               const std::string& override_type_name,
                               bool replace = true );

void set_inst_override( const std::string& relative_inst_path,
                        const std::string& original_type_name,
                        const std::string& override_type_name );

void print_override_info( const std::string& requested_type_name = "",
                          const std::string& name = "" );
```

```
    // Group: Hierarchical reporting interface
    void set_report_id_verbosity_hier( const std::string& id,
                                       int verbosity );

    void set_report_severity_id_verbosity_hier( uvm_severity severity,
                                                const std::string& id,
                                                int verbosity );

    void set_report_severity_action_hier( uvm_severity severity,
                                          uvm_action action );

    void set_report_id_action_hier( const std::string& id,
                                    uvm_action action );

    void set_report_severity_id_action_hier( uvm_severity severity,
                                             const std::string& id,
                                             uvm_action action );

    void set_report_default_file_hier( UVM_FILE file );
    void set_report_severity_file_hier( uvm_severity severity,
                                        UVM_FILE file );

    void set_report_id_file_hier( const std::string& id,
                                  UVM_FILE file );

    void set_report_severity_id_file_hier( uvm_severity severity,
                                           const std::string& id,
                                           UVM_FILE file );

    void set_report_verbosity_level_hier( int verbosity );
    virtual void pre_abort();

  }; // class uvm_component

} // namespace uvm
```

7.1.2 Construction interface

When creating a new UVM component, an application should always provide a local leaf name. The parent is traced from the current **uvm_component** at top of the hierarchy stack. The **uvm_component** hierarchy stack is built during module construction, in the pre-run phases **build_phase** and **connect_phase**. If the parent component is not derived from **uvm_component**, the leaf object becomes part of the object **uvm_root**. The full hierarchical name shall be unique; if it is not unique, a warning message is generated, and a number is appended at the end of the hierarchical name to make it unique.

Compatible with SystemC, it is illegal to create a component after the **before_end_of_elaboration** phase or UVM pre-run phases **build_phase** and **connect_phase**. The constructor for **uvm_component** spawns off the member function **run_phase** of that component.

7.1.2.1 Constructor

```
explicit uvm_component( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.1.3 Hierarchy interface

The following member functions provide user access to information about the component hierarchy, for example, topology.

7.1.3.1 get_parent

```
virtual uvm_component* get_parent() const;
```

The member function **get_parent** shall return a pointer to the component's parent, or NULL if it has no parent.

7.1.3.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full hierarchical name of the component. It shall concatenate the hierarchical name of the parent, if any, with the leaf name of the component, as returned by member function **uvm_object**::**get_name** (see Section 4.2.3.2).

7.1.3.3 get_children

```
void get_children( std::vector<uvm_component*>& children ) const;
```

The member function **get_children** shall return a vector of type std::vector containing a pointer to every instance of the component's children of class **uvm_component**.

7.1.3.4 get_child

```
uvm_component* get_child( const std::string& name ) const;
```

The member function **get_child** shall return a pointer to the component's child which matches the argument string *name*.

7.1.3.5 get_first_child

```
int get_first_child( std::string& name ) const;
```

The member function **get_first_child** shall pass the name of the first child of a component to the argument *name*. The member function returns true of the first child has been found; otherwise it shall return false.

7.1.3.6 get_first_child

```
int get_next_child( std::string& name ) const;
```

The member function **get_next_child** shall pass the name of the next child of a component, followed after a call to member function **get_first_child**, to the argument *name*. The member function returns true of the next child has been found; otherwise it shall return false.

7.1.3.7 get_num_children

```
int get_num_children() const;
```

The member function **get_num_children** shall return the number of the component's children.

7.1.3.8 has_child

```
bool has_child( const std::string& name ) const;
```

The member function **has_child** shall return true if this component has a child with the given name; otherwise it shall return false;

7.1.3.9 lookup

```
uvm_component* lookup( const std::string& name ) const;
```

The member function **lookup** shall return a pointer to a component with the passed hierarchical name *name* relative to the component. If the argument *name* is preceded with a '.' (dot), then the search shall begin relative to the top level (absolute lookup). The member function shall return NULL if no component has been found. The argument *name* shall not contain wildcards.

7.1.3.10 get_depth

```
unsigned int get_depth() const;
```

The member function **get_depth** shall return the component's depth from the root level. **uvm_top** has a depth of zero. The test and any other top level components have a depth of 1, and so on.

7.1.4 Phasing interface

UVM components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own member function, which derived components can override to incorporate component-specific behavior. During simulation, the phases are executed one by one, where one phase shall complete before the next phase begins.

The phases can be grouped in three main categories:

— Pre-run phases
— Run-time phases
— Post-run phases

7.1.4.1 Pre-run phases

The pre-run phases are responsible for the construction, connection and elaboration of the structural composition. In the pre-run phases, there is neither notion nor progress of time. It consists of the following phases:

— **build_phase**: The component constructs its children in this phase. It may use the static member function **uvm_config_db**::**get** to obtain any configuration for itself, the member function **uvm_config_db**::**set** to define any configuration for its own children, and the factory interface for actually creating the children and other objects it might need. An application shall declare child objects derived from **uvm_component** as pointers, instead of member fields of a component, such that they can be created via the factory in this phase.

— **connect_phase**: After creating the children in the **build_phase**, the component makes connections (binding of (TLM) ports and exports) from child-to-child or from child-to-self (that is, to promote a child or export up the hierarchy for external access).

— **end_of_elaboration_phase**: At this point, the entire testbench environment has been built and connected. No new components and connections shall be created from this point forward. Components do final checks for proper connectivity.

— **start_of_simulation_phase**: The simulation is about to begin, and this phase is used to perform any pre-run activity such as displaying banners, printing final testbench topology and configuration information.

As UVM components are derived from class **sc_module**, the inherited callbacks **before_end_of_elaboration**, **end_of_elaboration**, and **start_of_simulation** are available. It is recommended *not* to use these member functions for the construction of testbenches, but to use the UVM pre-run phases. Main reason is to support maximum reusability and flexibility for building, configuration and connecting various verification components using the same construction mechanism.

7.1.4.2 Run-time phases

The run-time phases are used to perform the actual verification. These phases are exclusively designed only for objects derived from class **uvm_component**. Run-time phases consume time.

A component's primary function is implemented in the member function **run_phase**. The component should not declare 'run_phase' as a thread process. The UVM-SystemC library spawns **run_phase** as a thread process. Other processes may be spawned from the run phase, if desired. When a component returns from executing its member function **run_phase**, it does not signify completion of its run phase. Any processes that it may have spawned still continue to run.

The run phase executes along with the other processes in the SystemC language: no special status is provided to the **run_phase** processes; for example, there is no guarantee that the **run_phase** processes is the first on the runable queue at time 0s, and hence there is no guarantee that the **run_phase** processes execute ahead of the other SystemC processes.

Concurrently to the execution of the **run_phase**, UVM defines a pre-defined schedule which consists of four groups of phases which are executed sequentially:

— Reset phases: Phases to apply reset signals for the DUT. Consists of three phases called **pre_reset_phase**, **reset_phase**, and **post_reset_phase**.
— Configure phases: Phases which can be used for the configuration of the DUT. Consists of three phases called **pre_configure_phase**, **configure_phase**, and **post_configure_phase**.
— Main phases: Phases which are used to apply the primary test stimulus to DUT. Consists of three phases called **pre_main_phase**, **main_phase**, and **post_main_phase**.
— Shutdown phase: Phases to wait for all data to be drained out of the DUT and to disable DUT. Consists of three phases called **pre_shutdown_phase**, **shutdown_phase**, and **post_shutdown_phase**.

7.1.4.3 Post-run phases

The post-run phases are:

— **extract_phase**: This phase occurs after the run phase is over. This phase is specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**. It is used to extract simulation results from coverage collectors and scoreboards, collect status/error counts, statistics, and other information from components in bottom-up order. Being a separate phase, the extract phase ensures all relevant data from potentially independent sources (that is, other components) are collected before being checked in the next phase.
— **check_phase**: This phase is specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**. Having extracted vital simulation results in the previous phase, the check phase is used to validate such data and determine the overall simulation outcome. It executes bottom-up.
— **report_phase**: Finally, the report phase is used to output results to files and/or the screen. This phase is also be specific to objects derived from class **uvm_component** and does not apply to objects derived from class **sc_module**.
— **final_phase**: This phase is called as soon as all tests have been executed and completed. This phase is used to close created or used files before the simulation exits.

7.1.4.4 build_phase

```
virtual void build_phase( uvm_phase& phase );
```

The member function **build_phase** shall provide a context to implement functionality as part of the build phase. The application shall not call this member function directly.

7.1.4.5 connect_phase

```
virtual void connect_phase( uvm_phase& phase );
```

The member function **connect_phase** shall provide a context to implement functionality as part of the connect phase. The application shall not call this member function directly.

7.1.4.6 end_of_elaboration_phase

```
virtual void end_of_elaboration_phase( uvm_phase& phase );
```

The member function **end_of_elaboration_phase** shall provide a context to implement functionality as part of the end of elaboration phase. The application shall not call this member function directly.

7.1.4.7 start_of_simulation_phase

```
virtual void start_of_simulation_phase( uvm_phase& phase );
```

The member function **start_of_simulation_phase** shall provide a context to implement functionality as part of the start of simulation phase. The application shall not call this member function directly.

7.1.4.8 run_phase

```
virtual void run_phase( uvm_phase& phase );
```

The member function **run_phase** shall provide a context to implement functionality as part of the run phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.9 pre_reset_phase

```
virtual void pre_reset_phase( uvm_phase& phase );
```

The member function **pre_reset_phase** shall provide a context to implement functionality as part of the pre-reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.10 reset_phase

```
virtual void reset_phase( uvm_phase& phase );
```

The member function **reset_phase** shall provide a context to implement functionality as part of the reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.11 post_reset_phase

```
virtual void post_reset_phase( uvm_phase& phase );
```

The member function **post_reset_phase** shall provide a context to implement functionality as part of the post-reset phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.12 pre_configuration_phase

```
virtual void pre_configuration_phase( uvm_phase& phase );
```

The member function **pre_configuration_phase** shall provide a context to implement functionality as part of the pre-configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.13 configuration_phase

```
virtual void configuration_phase( uvm_phase& phase );
```

The member function **configuration_phase** shall provide a context to implement functionality as part of the configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.14 post_configuration_phase

```
virtual void post_configuration_phase( uvm_phase& phase );
```

The member function **post_configuration_phase** shall provide a context to implement functionality as part of the post-configuration phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.15 pre_main_phase

```
virtual void pre_main_phase( uvm_phase& phase );
```

The member function **pre_main_phase** shall provide a context to implement functionality as part of the pre-main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.16 main_phase

```
virtual void main_phase( uvm_phase& phase );
```

The member function **main_phase** shall provide a context to implement functionality as part of the main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.17 post_main_phase

```
virtual void post_main_phase( uvm_phase& phase );
```

The member function **post_main_phase** shall provide a context to implement functionality as part of the post-main phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.18 pre_shutdown_phase

```
virtual void pre_shutdown_phase( uvm_phase& phase );
```

The member function **pre_shutdown_phase** shall provide a context to implement functionality as part of the pre-shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.19 shutdown_phase

```
virtual void shutdown_phase( uvm_phase& phase );
```

The member function **shutdown_phase** shall provide a context to implement functionality as part of the shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.20 post_shutdown_phase

```
virtual void post_shutdown_phase( uvm_phase& phase );
```

The member function **post_shutdown_phase** shall provide a context to implement functionality as part of the post-shutdown phase. An objection shall be raised, using the member function *phase*.**raise_objection**, to cause the phase to persist. Once all components have dropped their respective objection using *phase*.**drop_objection**, or if no components raise an objection, the phase shall be ended. Any processes spawned by this member function continue to run after the member function returns, but they shall be killed once the phase ends. The application shall not call this member function directly.

7.1.4.21 extract_phase

```
virtual void extract_phase( uvm_phase& phase );
```

The member function **extract_phase** shall provide a context to implement functionality as part of the extract phase. The application shall not call this member function directly.

7.1.4.22 check_phase

```
virtual void check_phase( uvm_phase& phase );
```

The member function **check_phase** shall provide a context to implement functionality as part of the check phase. The application shall not call this member function directly.

7.1.4.23 report_phase

```
virtual void report_phase( uvm_phase& phase );
```

The member function **report_phase** shall provide a context to implement functionality as part of the report phase. The application shall not call this member function directly.

7.1.4.24 final_phase

```
virtual void final_phase( uvm_phase& phase );
```

The member function **final_phase** shall provide a context to implement functionality as part of the final phase. The application shall not call this member function directly.

7.1.4.25 phase_started

```
virtual void phase_started( uvm_phase& phase );
```

The member function **phase_started** shall provide a context to implement functionality as part of the start of each phase. The argument *phase* specifies the phase being started. Any threads spawned in this callback are not affected when the phase ends.

7.1.4.26 phase_ready_to_end

```
virtual void phase_ready_to_end( uvm_phase& phase );
```

The member function **phase_ready_to_end** shall provide a context to implement functionality as part of the ending of each phase. The argument *phase* specifies the phase being ended. The member function shall be invoked when all objections to ending the given phase have been dropped, thus indicating that phase is ready to end. All this component's threads spawned for the given phase shall be killed upon return from this member function. Components needing to consume delta cycles or advance time to perform a clean exit from the phase may raise the phase's objection.

7.1.4.27 phase_ended

```
virtual void phase_ended( uvm_phase& phase );
```

The member function **phase_ended** shall provide a context to implement functionality at the end of each phase. The argument *phase* specifies the phase that has ended. Any threads spawned in this callback are not affected when the phase ends.

7.1.4.28 set_domain

```
void set_domain( uvm_domain* domain, int hier = 1 );
```

The member function **set_domain** shall set the phase domain to this component and, if *hier* is set, recursively to all its children.

7.1.4.29 get_domain

```
uvm_domain* get_domain() const;
```

The member function **get_domain** shall return a pointer to the phase domain set on this component.

7.1.4.30 define_domain

```
void define_domain( uvm_domain* domain );
```

The member function **define_domain** shall build a custom phase schedules into the provided domain passed as pointer.

7.1.4.31 set_phase_imp

```
void set_phase_imp( uvm_phase* phase, uvm_phase* imp, int hier = 1 );
```

The member function **set_phase_imp** shall provide a context for an application-specific phase implementation, which shall be created as a singleton object extending the default one and implementing required behavior for the member functions **execute** and **traverse**.

The optional argument *hier* specifies whether to apply the custom functor to the whole tree or just this component.

7.1.5 Process control interface

The class **uvm_component** has the following member functions to support process control constructs on the run process handle:

- **suspend**
- **resume**

The default implementation of these member functions is to invoke the corresponding process control construct on the component's run process handle, if the run process is active (that is, not already terminated), for those simulators that support process control constructs. Each of these member functions return true if the simulator supports process control constructs. For those simulators that do not support process control constructs, these member functions do nothing and return false.

NOTE—The process control interface requires at least Accellera Systems Initiative SystemC reference implementation version 2.3.0.

7.1.5.1 suspend

```
virtual bool suspend();
```

The member function **suspend** shall suspend operation of this component. It shall return true if suspending succeeds; otherwise it shall return false.

NOTE—This member function shall be implemented by the application to suspend the component according to the protocol and functionality it implements. A suspended component can be subsequently resumed by calling the member function resume.

7.1.5.2 resume

```
virtual bool resume();
```

The member function **resume** shall resume operation of this component. It shall return true if resuming succeeds; otherwise it shall return false.

NOTE—This member function shall be implemented by the application to resume a component that was previously suspended using member function suspend. Some components may start in the suspended state and may need to be explicitly resumed.

7.1.6 Configuration interface

The configuration interface accommodates additional printing and debug facilities for user-defined configurations using the configuration database **uvm_config_db**.

7.1.6.1 print_config

```
void print_config( bool recurse = false, bool audit = false ) const;
```

The member function **print_config** shall print all configuration information for this component, as set by previous calls to **uvm_config_db**<T>::**set** and exports to the resources pool. The settings are printing in the order of their precedence. If argument *recurse* is set, then configuration information for all children and below are printed as well. If argument *audit* is set, then the audit trail for each resource is printed along with the resource name and value.

7.1.6.2 print_config_with_audit

```
void print_config_with_audit( bool recurse = false ) const;
```

The member function **print_config_with_audit** shall print all configuration information for this component, as set by previous calls to **uvm_config_db**<T>::**set** and exports to the resources pool. The settings are printing in the order of their precedence, and without the audit trail. If argument *recurse* is set, then configuration information for all children and below are printed as well.

7.1.6.3 print_config_matches

```
void print_config_matches( bool enable = true );
```

The member function **print_config_matches** shall print all information about the matching configuration settings as they are being applied for each call of **uvm_config_db**<T>::**get**. By default, this information is not printed.

7.1.7 Objection interface

These member functions provide object level access into the **uvm_objection** mechanism.

7.1.7.1 raised

```
virtual void raised( uvm_objection* objection,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );
```

The member function **raised** shall be called when this or a descendant of this component instance raises the specified objection. The argument *source_obj* is the object that originally raised the objection. The argument *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The argument *count* indicates the number of objections raised by the *source_obj*.

7.1.7.2 dropped

```
virtual void dropped( uvm_objection* objection,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );
```

The member function **dropped** shall be called when this or a descendant of this component instance drops the specified objection. The argument *source_obj* is the object that originally dropped the objection. The argument *description* is optionally provided by the *source_obj* to give a reason for dropping the objection. The argument *count* indicates the number of objections dropped by the *source_obj*.

7.1.7.3 all_dropped

```
virtual void all_dropped( uvm_objection* objection,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );
```

The member function **all_dropped** shall be called when all objections have been dropped by this component and all its descendants. The argument *source_obj* is the object that dropped the last objection. The argument *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The argument *count* indicates the number of objections dropped by the *source_obj*.

7.1.8 Factory interface

The factory interface provides components with convenient access to the UVM's central **uvm_factory** object. The member functions defined in this section shall call the corresponding member functions in **uvm_factory**, passing whatever arguments it can to reduce the number of arguments required of the user.

7.1.8.1 create_component

```
uvm_component* create_component( const std::string& requested_type_name,
                                 const std::string& name );
```

The member function **create_component** shall provide a convenience layer to the member function **uvm_factory**::**create_component_by_name**, which 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* (see Section 6.4.4.4).

7.1.8.2 create_object

```
uvm_object* create_object( const std::string& requested_type_name,
                           const std::string& name );
```

The member function **create_object** shall provide a convenience layer to the member function **uvm_factory**::**create_object_by_name**, which calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name* (see Section 6.4.4.2).

7.1.8.3 set_type_override_by_type

```
static void set_type_override_by_type( uvm_object_wrapper* original_type,
                                       uvm_object_wrapper* override_type,
                                       bool replace = true );
```

The member function **set_type_override_by_type** shall provide a convenience layer to the member function **uvm_factory**::**set_type_override_by_type**, which registers a factory override for components and objects created at this level of hierarchy or below (see Section 6.4.3.3).

The argument *original_type* represents the type that is being overridden. In subsequent calls to **uvm_factory**::**create_object_by_type** or **uvm_factory**::**create_component_by_type**, if the argument *requested_type* matches the *original_type* and the instance paths match, the factory shall produce the *override_type*.

7.1.8.4 set_inst_override_by_type

```
void set_inst_override_by_type( const std::string& relative_inst_path,
                                uvm_object_wrapper* original_type,
                                uvm_object_wrapper* override_type );
```

The member function **set_inst_override_by_type** shall provide a convenience layer to the member function **uvm_factory**::**set_inst_override_by_type**, which registers a factory override for components and objects created at this level of hierarchy or below (see Section 6.4.3.1).

The argument *relative_inst_path* is relative to this component and may include wildcards. The argument *original_type* represents the type that is being overridden. In subsequent calls to **uvm_factory**::**create_object_by_type** or **uvm_factory**::**create_component_by_type**, if the *requested_type* matches the *original_type* and the instance paths match, the factory shall produce the *override_type*.

7.1.8.5 set_type_override

```
static void set_type_override( const std::string& original_type_name,
                               const std::string& override_type_name,
                               bool replace = true );
```

The member function **set_type_override** shall provide a convenience layer to the member function **uvm_factory**::**set_type_override_by_name**, which 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* (see Section 6.4.3.4).

The argument *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** or **create_object** with the same string and matching instance path shall produce the type represented by *override_type_name*. The argument *override_type_name* shall refer to a preregistered type in the factory.

7.1.8.6 set_inst_override

```
void set_inst_override( const std::string& relative_inst_path,
                        const std::string& original_type_name,
                        const std::string& override_type_name );
```

The member function **set_inst_override** shall provide a convenience layer to the member function **uvm_factory**::**set_inst_override_by_name**, which registers a factory override for components created at this level of hierarchy or below (see Section 6.4.3.2).

The argument *relative_inst_path* is relative to this component and may include wildcards. The argument *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to **create_component** or **create_object** with the same string and matching instance path shall produce the type represented by *override_type_name*. The *override_type_name* shall refer to a preregistered type in the factory.

7.1.8.7 print_override_info

```
void print_override_info( const std::string& requested_type_name = "",
                          const std::string& name = "" );
```

The member function **print_override_info** shall provide the same lookup process as **create_object** and **create_component**, but instead of creating an object, it prints information about what type of object would be created given the provided arguments.

7.1.9 Hierarchical reporting interface

This interface provides versions of the member function **set_report_*** in the base class **uvm_report_object** that are applied recursively to this component and all its children. When a report is issued and its associated action **UVM_LOG** is set, the report shall be sent to its associated file descriptor.

7.1.9.1 set_report_id_verbosity_hier

```
void set_report_id_verbosity_hier( const std::string& id,
                                   int verbosity );
```

The member function **set_report_id_verbosity_hier** shall recursively associate the specified verbosity with reports of the given *id*. A verbosity associated with a particular severity-id pair, using member function **set_report_severity_id_verbosity_hier**, shall take precedence over a verbosity associated by this member function.

7.1.9.2 set_report_severity_id_verbosity_hier

```
void set_report_severity_id_verbosity_hier( uvm_severity severity,
                                            const std::string& id,
                                            int verbosity );
```

The member function **set_report_severity_id_verbosity_hier** shall recursively associate the specified verbosity with reports of the given *severity* with *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.

7.1.9.3 set_report_severity_action_hier

```
void set_report_severity_action_hier( uvm_severity severity,
                                      uvm_action action );
```

The member function **set_report_severity_action_hier** shall recursively associate the specified action with reports of the given *severity*. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which shall take precedence over an action associated with a severity as defined in this member function.

7.1.9.4 set_report_id_action_hier

```
void set_report_id_action_hier( const std::string& id,
                                uvm_action action );
```

The member function **set_report_id_action_hier** shall recursively associate the specified action with reports of the given *id*. An action associated with a particular severity-id pair shall take precedence over an action associated with id as defined in this member function.

7.1.9.5 set_report_severity_id_action_hier

```
void set_report_severity_id_action_hier( uvm_severity severity,
                                         const std::string& id,
                                         uvm_action action );
```

The member function **set_report_severity_id_action_hier** shall recursively associate the specified action with reports of the given *severity* with *id* pair. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which shall take precedence over an action associated with a severity.

7.1.9.6 set_report_default_file_hier

```
void set_report_default_file_hier( UVM_FILE file );
```

The member function **set_report_default_file_hier** shall recursively associate the report to the default *file* descriptor. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity, which shall take precedence over the default file descriptor as defined in this member function.

7.1.9.7 set_report_severity_file_hier

```
void set_report_severity_file_hier( uvm_severity severity,
                                    UVM_FILE file );
```

The member function **set_report_severity_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *severity*. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity as defined in this member function.

7.1.9.8 set_report_id_file_hier

```
void set_report_id_file_hier( const std::string& id,
                              UVM_FILE file );
```

The member function **set_report_id_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *id*. A file associated with a particular severity-id pair shall take precedence over a file associated with id as defined in this member function.

7.1.9.9 set_report_severity_id_file_hier

```
void set_report_severity_id_file_hier( uvm_severity severity,
                                       const std::string& id,
                                       UVM_FILE file );
```

The member function **set_report_severity_id_file_hier** shall recursively associate the specified *file* descriptor with reports of the given *severity* and *id* pair. A file associated with a particular severity-id pair shall take precedence over a file associated with id, which shall take precedence over a file associated with a severity, which shall take precedence over the default file descriptor.

7.1.9.10 set_report_verbosity_level_hier

```
void set_report_verbosity_level_hier( int verbosity );
```

The member function **set_report_verbosity_level_hier** shall recursively set the maximum verbosity level for reports for this component and all those below it. Any report from this component sub-tree whose verbosity exceeds this maximum are ignored.

7.1.9.11 pre_abort

```
virtual void pre_abort();
```

The member function **pre_abort** shall be executed when the message system is executing a **UVM_EXIT** action. 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 application before the termination happens. For example, a test may want to execute the report function of a particular component even when an error condition has happened to force a premature termination. The member function **pre_abort** shall be called for all UVM components in the hierarchy in a bottom-up fashion.

7.1.10 Macros

UVM-SystemC defines the following macros for class **uvm_component**:

Utility macro **UVM_COMPONENT_UTILS** (*classname*) to be used inside the Class definition, that expands to:

— The declaration of the member function **get_type_name**, which returns the type of the class as string

— The declaration of the member function **get_type**, which returns a factory proxy object for the type

— The class **uvm_component_registry<***classname***>** used by the factory.

Template classes shall use the macro **UVM_COMPONENT_PARAM_UTILS**, to guarantee correct registration of one or more parameters passed to the class template. Note that template classes are not evaluated at compile-time, and thus not registered with the factory. Due to this, name-based lookup with the factory for template classes is not possible. Instead, an application shall use the member function **get_type** for factory overrides.

7.2 uvm_driver

The class **uvm_driver** is the base class for drivers that initiate requests for new transactions. The ports are typically connected to the exports of an appropriate sequencer component of class **uvm_sequencer**.

7.2.1 Class definition

```
namespace uvm {

  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_driver : public uvm_component
  {
   public:

    // Ports
    uvm_seq_item_pull_port<REQ, RSP> seq_item_port;
    uvm_analysis_port<RSP> rsp_port;

    // Constuctor
    explicit uvm_driver( uvm_component_name name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_driver

} // namespace uvm
```

7.2.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types shall be a derivative of class **uvm_sequence_item**.

7.2.3 Ports

7.2.3.1 seq_item_port

```
uvm_seq_item_pull_port<REQ, RSP> seq_item_port;
```

The port **seq_item_port** of type **uvm_seq_item_pull_port** shall be defined to connect (bind) the driver to the corresponding export in the sequencer.

NOTE—In line with the UVM-SystemVerilog syntax, the member function connect can be used to establish the binding between the driver and the sequencer. The UVM-SystemC implementation also supports the SystemC syntax using the member function **bind** or using **operator()** to perform the binding.

7.2.3.2 rsp_port

```
uvm_analysis_port<RSP> rsp_port;
```

The port **rsp_port** shall provide a way of sending responses back to the connected sequencer.

7.2.4 Member functions

7.2.4.1 Constructor

```
explicit uvm_driver( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.2.4.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.3 uvm_monitor

The class **uvm_monitor** is the base class for monitors. Deriving from **uvm_monitor** allows an application to distinguish monitors from generic component types inheriting from uvm_component. Such monitors shall automatically inherit features that may be added to **uvm_monitor** in the future.

7.3.1 Class definition

```
namespace uvm {

  class uvm_monitor : public uvm_component
  {
   public:

    // Constructor
    explicit uvm_monitor( uvm_component_name name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_monitor
```

```
} // namespace uvm
```

7.3.2 Member functions

7.3.2.1 Constructor

```
explicit uvm_monitor( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.3.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.4 uvm_agent

The class **uvm_agent** is the base class for the creation of agents. Deriving from **uvm_agent** shall enable an application to distinguish agents from other component types also using its inheritance. Such agents shall automatically inherit features that may be added to **uvm_agent** in the future.

While an agent's build function, inherited from **uvm_component**, can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

7.4.1 Class definition

```
namespace uvm {

  class uvm_agent : public uvm_component
  {
   public:

    // Constructor
    explicit uvm_agent( uvm_component_name name );

    // Member functions
    virtual const std::string get_type_name() const;
    uvm_active_passive_enum get_is_active() const;

  }; // class uvm_agent

} // namespace uvm
```

7.4.2 Member functions

7.4.2.1 Constructor

```
explicit uvm_agent( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.4.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.4.2.3 get_is_active

```
uvm_active_passive_enum get_is_active();
```

The member function **get_is_active** shall return **UVM_ACTIVE** if the agent is acting as an active agent and **UVM_PASSIVE** if it is acting as a passive agent (see Section 17.4.4). An application may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

7.5 uvm_env

The class **uvm_env** is the base class for the creation of a self-containing verification environment, such as a verification component which contains multiple agents.

7.5.1 Class definition

```
namespace uvm {

  class uvm_env : public uvm_component
  {
   public:

    // Constructor
    explicit uvm_env( uvm_component_name name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_env

} // namespace uvm
```

7.5.2 Member functions

7.5.2.1 Constructor

```
explicit uvm_env( uvm_component_name name );
```

Constructor

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.5.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.6 uvm_test

The class **uvm_test** is the base class for the test environment.

7.6.1 Class definition

```
namespace uvm {

  class uvm_test : public uvm_component
  {
   public:

    // Constructor
    explicit uvm_test( uvm_component_name name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_test

} // namespace uvm
```

7.6.2 Member functions

7.6.2.1 Constructor

```
explicit uvm_test( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.6.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the object derived from this class as an object of type std::string.

7.7 uvm_scoreboard

The class **uvm_scoreboard** is the base class for the creation of a scoreboard. Deriving from **uvm_scoreboard** shall enable an application to distinguish scoreboards from other component types inheriting directly from **uvm_component**. Such scoreboards shall automatically inherit and benefit from features that may be added to **uvm_scoreboard** in the future.

7.7.1 Class definition

```
namespace uvm {

  class uvm_scoreboard : public uvm_component
  {
   public:
    explicit uvm_scoreboard( uvm_component_name name );
    virtual const std::string get_type_name() const;
  }; // class uvm_scoreboard

} // namespace uvm
```

7.7.2 Member functions

7.7.2.1 Constructor

```
explicit uvm_scoreboard( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.7.2.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component derived from this class as an object of type std::string.

7.8 uvm_subscriber

The class **uvm_subscriber** is the base class for the creation of a subscriber. It 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 member function **write** to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

7.8.1 Class definition

```
namespace uvm {

  template <typename T = int>
  class uvm_subscriber : public uvm_component
  {
   public:

    // Export
    uvm_analysis_export<T> analysis_export;

    // Constructor
    explicit uvm_subscriber( uvm_component_name name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_subscriber

} // namespace uvm
```

7.8.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis export.

7.8.3 Export

7.8.3.1 analysis_export

```
uvm_analysis_export<T> analysis_export;
```

The export **analysis_export** shall provide access to the member function **write**, which derived subscribers shall implement.

7.8.4 Member functions

7.8.4.1 Constructor

```
explicit uvm_subscriber( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

7.8.4.2 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of the component derived from this class as an object of type std::string.

8. Sequencer classes

The sequencer classes offer the interface between the stimuli generators (by means of sequences) and the structural composition of the test infrastructure using verification components. The sequencer is integral part of a verification component, which can be enabled in case the verification component is marked as 'active' (driving) element.

The sequencer processes the transactions, defined as objects derived from class **uvm_sequence_item** or class **uvm_sequence** and passes these transactions to the driver (object derived from class **uvm_driver**).

The following sequencer classes are defined:

— **uvm_sequencer_base**

— **uvm_sequencer_param_base**

— **uvm_sequencer**

8.1 uvm_sequencer_base

The class **uvm_sequencer_base** is the root base class for all sequencer classes.

8.1.1 Class definition

```
namespace uvm {

  class uvm_sequencer_base : public uvm_component
  {
   public:
    // Constructor
    explicit uvm_sequencer_base( uvm_component_name name );

    // Member functions
    bool is_child ( uvm_sequence_base* parent, const uvm_sequence_base* child ) const;

    virtual int user_priority_arbitration(
                                 std::vector< uvm_sequence_request* > avail_sequences );

    virtual void execute_item( uvm_sequence_item* item );
    virtual void start_phase_sequence( uvm_phase& phase );

    virtual void wait_for_grant( uvm_sequence_base* sequence_ptr,
                                 int item_priority = -1,
                                 bool lock_request = false);

    virtual void wait_for_item_done( uvm_sequence_base* sequence_ptr,
                                     int transaction_id = -1 );

    bool is_blocked( const uvm_sequence_base* sequence_ptr ) const;
    bool has_lock( uvm_sequence_base* sequence_ptr );
    virtual void lock( uvm_sequence_base* sequence_ptr );
    virtual void grab( uvm_sequence_base* sequence_ptr );
    virtual void unlock( uvm_sequence_base* sequence_ptr );
    virtual void ungrab( uvm_sequence_base* sequence_ptr );
    virtual void stop_sequences();
    virtual bool is_grabbed() const;
    virtual uvm_sequence_base* current_grabber() const;
    virtual bool has_do_available();
    void set_arbitration( SEQ_ARB_TYPE mode );
    SEQ_ARB_TYPE get_arbitration() const;
    virtual void wait_for_sequences();

    virtual void send_request( uvm_sequence_base* sequence_ptr,
                               uvm_sequence_item* seq_item,
                               bool rerandomize = false);

  }; // class uvm_sequencer_base
```

```
} // namespace uvm
```

8.1.2 Constructor

```
explicit uvm_sequencer_base( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.1.3 Member functions

8.1.3.1 is_child

```
bool is_child( uvm_sequence_base* parent, const uvm_sequence_base* child ) const;
```

The member function **is_child** shall return true if the child sequence is a child of the parent sequence and false otherwise.

8.1.3.2 user_priority_arbitration

```
virtual int user_priority_arbitration(
            std::vector< uvm_sequence_request* > avail_sequences );
```

The member function **user_priority_arbitration** shall be called by an application when the sequencer arbitration mode is set to **SEQ_ARB_USER** (via the member function **set_arbitration**) each time that it needs to arbitrate among sequences. Derived sequencers may override this member function to perform a custom arbitration policy. The override shall return one of the entries from the *avail_sequences* queue, which are indexes into an internal queue of type std::vector< **uvm_sequence_request*** >. The default implementation shall behave similar as **SEQ_ARB_FIFO**, which returns the first entry of *avail_sequences*.

8.1.3.3 execute_item

```
virtual void execute_item( uvm_sequence_item* item );
```

The member function **execute_item** shall execute the given transaction item given as argument directly on this sequencer. A temporary parent sequence is automatically created for the item. There is no capability to retrieve responses. If the driver returns responses, it accumulates in the sequencer, eventually causing response overflow unless member function **uvm_sequence_base**::**set_response_queue_error_report_disabled** is called.

8.1.3.4 start_phase_sequence

```
virtual void start_phase_sequence( uvm_phase phase );
```

The member function **start_phase_sequence** shall start the default sequence for the phase given as argument. The default sequence is configured via resources using either a sequence instance or sequence type (object wrapper). If both are used, the sequence instance takes precedence. When attempting to override a previous default sequence setting, an application shall override both the instance and type (wrapper) resources, else the override may not take effect.

8.1.3.5 wait_for_grant

```
virtual void wait_for_grant( uvm_sequence_base* sequence_ptr,
                             int item_priority = -1,
```

```
                              bool lock_request = false);
```

The member function **wait_for_grant** shall issue a request for the specified sequence. If *item_priority* is not specified, then the current sequence priority shall be used by the arbiter. If a *lock_request* is made, then the sequencer shall issue a lock immediately before granting the sequence. The lock may be granted without the sequence being granted if the member function **is_relevant** of the sequence instance is not asserted.

When this member function returns, the sequencer has granted the sequence, and the sequence shall call **send_request** without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the **send_request** call.

8.1.3.6 wait_for_item_done

```
virtual void wait_for_item_done( uvm_sequence_base* sequence_ptr,
                                 int transaction_id = -1 );
```

The member function **wait_for_item_done** shall block the sequence until the driver calls **item_done** or **put** on a transaction issued by the specified sequence. If no *transaction_id* parameter is specified, then the call shall return the next time that the driver calls **item_done** or **put**. If a specific *transaction_id* is specified, then the call shall only return when the driver indicates that it has completed that specific item.

8.1.3.7 is_blocked

```
bool is_blocked( const uvm_sequence_base* sequence_ptr ) const;
```

The member function **is_blocked** shall return true if the sequence referred to by *sequence_ptr* is currently locked out of the sequencer. It shall return false if the sequence is currently allowed to issue operations.

Even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock.

8.1.3.8 has_lock

```
bool has_lock( uvm_sequence_base* sequence_ptr );
```

The member function **has_lock** shall return true if the sequence referred to in the parameter currently has a lock on the sequencer; otherwise it shall return false. 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.

8.1.3.9 lock

```
virtual void lock( uvm_sequence_base* sequence_ptr );
```

The member function **lock** shall request a lock for the sequence specified by the specified argument *sequence_ptr*. A lock request shall be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. The lock call shall return when the lock has been granted.

8.1.3.10 grab

```
virtual void grab( uvm_sequence_base* sequence_ptr );
```

The member function **grab** shall request a grab for the sequence specified by the specified argument *sequence_ptr*. A grab request is put in front of the arbitration queue. It shall be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence. The grab call shall return when the grab has been granted.

8.1.3.11 unlock

```
virtual void unlock( uvm_sequence_base* sequence_ptr );
```

The member function **unlock** shall remove any locks and grabs obtained by the specified argument *sequence_ptr*.

8.1.3.12 ungrab

```
virtual void ungrab( uvm_sequence_base* sequence_ptr );
```

The member function **ungrab** shall remove any locks and grabs obtained by the specified argument *sequence_ptr*.

8.1.3.13 stop_sequences

```
virtual void stop_sequences();
```

The member function **stop_sequences** shall inform 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.

8.1.3.14 is_grabbed

```
virtual bool is_grabbed() const;
```

The member function **is_grabbed** shall return true if any sequence currently has a lock or grab on this sequencer; otherwise it shall return false.

8.1.3.15 current_grabber

```
virtual uvm_sequence_base* current_grabber() const;
```

The member function **current_grabber** shall return a pointer to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, it returns the child that is currently allowed to perform operations on the sequencer.

8.1.3.16 has_do_available

```
virtual bool has_do_available();
```

The member function **has_do_available** shall return true if any sequence running on this sequencer is ready to supply a transaction, otherwise it shall return false.

8.1.3.17 set_arbitration

```
void set_arbitration( SEQ_ARB_TYPE mode );
```

The member function **set_arbitration** shall set the arbitration mode for the sequencer. The argument *mode* shall be of type **SEQ_ARB_TYPE** and set to:

— **SEQ_ARB_FIFO**: Requests are granted in FIFO order (default).

— **SEQ_ARB_WEIGHTED**: Requests are granted randomly by weight.

— **SEQ_ARB_RANDOM**: Requests are granted randomly.

— **SEQ_ARB_STRICT_FIFO**: Requests at highest priority granted in FIFO order.

— **SEQ_ARB_STRICT_RANDOM**: Requests at highest priority granted in randomly.

— **SEQ_ARB_USER**: Arbitration is delegated to the user-defined member function, **user_priority_arbitration**, which specifies the next sequence to grant.

The default arbitration mechanism shall be set to **SEQ_ARB_FIFO**.

8.1.3.18 get_arbitration

```
SEQ_ARB_TYPE get_arbitration() const;
```

The member function **get_arbitration** shall return the current arbitration mode set for the sequencer (see Section 8.1.3.20).

8.1.3.19 wait_for_sequences

```
virtual void wait_for_sequences();
```

The member function **wait_for_sequences** shall wait for a sequence to have a new item available.

8.1.3.20 send_request

```
virtual void send_request( uvm_sequence_base* sequence_ptr,
                           uvm_sequence_item* seq_item,
                           bool rerandomize = false);
```

Derived classes shall implement the member function **send_request** to send a request item to the sequencer, which shall forward it to the driver.

This member function shall only be called after a **wait_for_grant** call.

NOTE—Randomization is not yet supported in UVM-SystemC.

8.2 uvm_sequencer_param_base

The class **uvm_sequencer_param_base** extends the base class **uvm_sequencer_base** for specific request (REQ) and response (RSP) types, which are specified as template arguments.

8.2.1 Class definition

```
namespace uvm {
```

```
  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequencer_param_base : public uvm_sequencer_base
  {
   public:
    // Constructor
    explicit uvm_sequencer_param_base( uvm_component_name name );

    // Group: Requests
    void send_request( uvm_sequence_base* sequence_ptr,
                       uvm_sequence_item* seq_item,
                       bool rerandomize = false );

    REQ get_current_item() const;

  }; // class uvm_sequencer_param_base

} // namespace uvm
```

8.2.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types shall be a derivative of class **uvm_sequence_item**.

8.2.3 Constructor

```
explicit uvm_sequencer_param_base( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.2.4 Requests

8.2.4.1 send_request

```
virtual void send_request( uvm_sequence_base* sequence_ptr,
                           uvm_sequence_item* seq_item,
                           bool rerandomize = false );
```

The member function **send_request** sends a request item pointed to by *seq_item* to the sequencer pointed to by *sequence_ptr*. The sequencer shall forward it to the driver. This member function shall only be called after a call to member function **wait_for_grant**.

NOTE—Randomization is not yet supported in UVM-SystemC.

8.2.4.2 get_current_item

```
REQ get_current_item() const;
```

The member function **get_current_item** shall return the requested item of type REQ, which is currently being executed by the sequencer. If the sequencer is not currently executing an item, this member function shall return NULL.

The sequencer is executing an item from the time that **get_next_item** or **peek** is called by the driver until the time that member function **get** or **item_done** is called by the driver. In case a driver calls member function **get**, the current item cannot be shown, since the item is completed at the same time as it is requested.

8.3 uvm_sequencer

The class **uvm_sequencer** defines the interface for the TLM communication of sequences or sequence-items by providing access via an export object of class **sc_export**.

8.3.1 Class definition

```
namespace uvm {

  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequencer : public uvm_sequencer_param_base<REQ,RSP>,
                        public uvm_sqr_if_base<REQ, RSP>
  {
   public:
    // Constructor
    explicit uvm_sequencer( uvm_component_name name );

    // Group: Exports
    uvm_seq_item_pull_imp<REQ, RSP, this> seq_item_export;

    // Group: Sequencer interface
    virtual REQ get_next_item( REQ* req = NULL );
    virtual bool try_next_item( REQ& req );
    virtual void item_done( const RSP& item, bool use_item = true );
    virtual void item_done();
    virtual REQ get( REQ* req = NULL );
    virtual void get( REQ& req );
    virtual REQ peek( REQ* req = NULL );
    virtual void put( const RSP& rsp );
    virtual void stop_sequences();

  }; // class uvm_sequencer

} // namespace uvm
```

8.3.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types shall be a derivative of class **uvm_sequence_item**.

8.3.3 Constructor

```
explicit uvm_sequencer( uvm_component_name name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

8.3.4 Exports

8.3.4.1 seq_item_export

```
uvm_seq_item_pull_imp<REQ, RSP, this > seq_item_export;
```

The export **seq_item_export** shall provide access to the sequencer's implementation **uvm_seq_item_pull_imp** via the sequencer interface **uvm_sqr_if_base**<REQ, RSP> (see Section 14.13).

8.3.5 Sequencer interface

8.3.5.1 get_next_item

```
virtual REQ get_next_item( REQ* req = NULL );
```

The member function **get_next_item** shall retrieve the next available item from a sequence (see also Section 14.13.3.1).

8.3.5.2 try_next_item

```
virtual bool try_next_item( REQ& req );
```

The member function **try_next_item** shall retrieve the next available item from a sequence if one is available (see also Section 14.13.3.2).

8.3.5.3 item_done

```
virtual void item_done( const RSP& item, bool use_item = true );
virtual void item_done();
```

The member function **item_done** shall indicate that the request is completed (see also Section 14.13.3.3).

8.3.5.4 get

```
virtual REQ get( REQ* req = NULL );
virtual void get( REQ& req );
```

The member function **get** shall retrieve the next available item from a sequence (see also Section 14.13.3.4).

8.3.5.5 peek

```
virtual REQ peek( REQ* req = NULL );
```

The member function **peek** shall return the current request item if one is in the FIFO (see also Section 14.13.3.5).

8.3.5.6 put

```
virtual void put( const RSP& rsp );
```

The member function **put** shall send a response back to the sequence that issued the request (see also Section 14.13.3.6).

8.3.5.7 stop_sequences

```
virtual void stop_sequences();
```

The member function **stop_sequences** shall tell 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.

8.3.6 Macros

8.3.6.1 UVM_DECLARE_P_SEQUENCER

```
UVM_DECLARE_P_SEQUENCER(SEQUENCER);
```

The macro **UVM_DECLARE_P_SEQUENCER** shall declare a variable **p_sequencer** whose type is specified by the argument *SEQUENCER*.

9. Sequence classes

The sequence classes offer the infrastructure to create stimuli descriptions based on transactions, encapsulated as a sequence or sequence item. As the sequences and sequence items only describe stimuli, they are independent and thus not part of the structural hierarchy of a UVM agent (in which sequencer, driver and monitor resides). Instead, they are included at a higher functional layer defined within the UVM environment (e.g. encapsulated within a verification component derived from class **uvm_env**) or as part of a UVM test environment (component derived from class **uvm_test**).

The following sequence classes are defined:

— **uvm_transaction**

— **uvm_sequence_item**

— **uvm_sequence_base**

— **uvm_sequence**

When sequences are executed parallel, the sequencer shall arbitrate among the parallel sequences. By default, requests are granted in a first-in-first-out (FIFO) order (see Section 8.1.3.17).

9.1 uvm_transaction

The class **uvm_transaction** is the root base class for all UVM transactions. As such, the class **uvm_sequence_item** shall be derived from this class. The main purpose of this class is to provide timestamp properties, notification events, and transaction recording.

9.1.1 Class definition

```
namespace uvm {

  class uvm_transaction : public uvm_object
  {
   public:
    // Constructors
    uvm_transaction();
    explicit uvm_transaction( const std::string& name );

    // Member functions
    void set_transaction_id( int id );
    int get_transaction_id() const;

  }; // class uvm_transaction

} // namespace uvm
```

9.1.2 Constructors

```
uvm_transaction();
explicit uvm_transaction( const std::string& name );
```

The constructor shall create and initialize an instance of the class, which is derived from class **uvm_object**, with the name *name* passed as an argument.

9.1.3 Constraints on usage

An application shall not create transactions based on this base class. Instead, it shall use the class **uvm_sequence_item** or class **uvm_sequence**.

9.1.4 Member functions

9.1.4.1 set_transaction_id

```
void set_transaction_id( int id );
```

The member function **set_transaction_id** shall set the transaction's numeric identifier (ID), passed as argument *id*. If the transaction ID is not set via this member function, the transaction ID defaults to -1.

When using sequences to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

9.1.4.2 get_transaction_id

```
int get_transaction_id() const;
```

The member function **get_transaction_id** shall return the transaction's numeric identifier (ID), which is -1 if not set explicitly by **set_transaction_id**.

When using an object derived from class **uvm_sequence**<REQ, RSP> to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

9.2 uvm_sequence_item

The class **uvm_sequence_item** is the base class for application-defined sequence items and also serves as the base class for class **uvm_sequence**. The class **uvm_sequence_item** provides basic functionality for transactional objects, both sequence items and sequences, to operate in the sequence mechanism.

9.2.1 Class definition

```
namespace uvm {

  class uvm_sequence_item : public uvm_transaction
  {
   public:
    / Constructors
    uvm_sequence_item();
    explicit uvm_sequence_item( const std::string& name );

    // Member functions
    void set_use_sequence_info( bool value );
    bool get_use_sequence_info() const;
    void set_id_info( uvm_sequence_item& item );
    virtual void set_sequencer( uvm_sequencer_base* sequencer );
    uvm_sequencer_base* get_sequencer() const;
    void set_parent_sequence( uvm_sequence_base* parent );
    uvm_sequence_base* get_parent_sequence() const;
    void set_depth( int value );
    int get_depth() const;
    virtual bool is_item() const;
    const std::string get_root_sequence_name() const;
    const uvm_sequence_base* get_root_sequence() const;
    const std::string get_sequence_path() const;

  }; // class uvm_sequence_item

} // namespace uvm
```

9.2.2 Constructors

```
uvm_sequence_item();
explicit uvm_sequence_item( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name*passed as an argument.

9.2.3 Member functions

9.2.3.1 set_use_sequence_info

```
void set_use_sequence_info( bool value );
```

The member function **set_use_sequence_info** shall enable or disable printing, copying, or recording of sequence information (sequencer, parent_sequence, sequence_id, etc.). When the argument of this member function is set to false, then the usage of sequence information shall be disabled. When the argument of this member function is set to true, the printing and copying of sequence information shall be enabled.

9.2.3.2 get_use_sequence_info

```
bool get_use_sequence_info() const;
```

The member function **get_use_sequence_info** shall return true if the usage of sequence information, such as printing and copying of sequence information, has been enabled. The member function shall return false if the usage of sequence information has been disabled.

9.2.3.3 set_id_info

```
void set_id_info( uvm_sequence_item& item );
```

The member function **set_id_info** shall copy the sequence ID and transaction ID from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

9.2.3.4 set_sequencer

```
virtual void set_sequencer( uvm_sequencer_base* sequencer );
```

The member function **set_sequencer** shall set the default sequencer, passed as argument, to be used for the sequence or sequence item for which this member function is called. It shall take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

9.2.3.5 get_sequencer

```
uvm_sequencer_base* get_sequencer() const;
```

The member function **get_sequencer** shall return a pointer to the default sequencer used by the sequence or sequence item for which this member function is called.

9.2.3.6 set_parent_sequence

```
void set_parent_sequence( uvm_sequence_base* parent );
```

The member function **set_parent_sequence** shall set the parent sequence, passed as an argument, of the sequence or sequence item.

9.2.3.7 get_parent_sequence

```
uvm_sequence_base* get_parent_sequence() const;
```

The member function **get_parent_sequence** shall return a pointer to the parent sequence of any sequence for which this member function was called. If this is a parent sequence, the member function shall return NULL.

9.2.3.8 set_depth

```
void set_depth( int value );
```

The member function **set_depth** shall set the depth of a particular sequence. If this member function is not called, the depth of any sequence shall be calculated automatically. When called, the member function shall override the automatically calculated depth, even if it is incorrect.

9.2.3.9 get_depth

```
int get_depth() const;
```

The member function **get_depth** shall return the depth of sequence from its parent. A parent sequence has a depth of 1, its child has a depth of 2, and its grandchild has a depth of 3.

9.2.3.10 is_item

```
virtual bool is_item() const;
```

The member function **is_item** shall return true when the object for which the member function is called is derived from **uvm_sequence_item**. It shall return false if the object is derived from class **uvm_sequence**.

9.2.3.11 get_root_sequence_name

```
const std::string get_root_sequence_name() const;
```

The member function **get_root_sequence_name** shall provide the name of the root sequence (the top-most parent sequence).

9.2.3.12 get_root_sequence

```
const uvm_sequence_base* get_root_sequence() const;
```

The member function **get_root_sequence** shall provide a reference to the root sequence (the top-most parent sequence).

9.2.3.13 get_sequence_path

```
const std::string get_sequence_path() const;
```

The member function **get_sequence_path** shall provide a string of names of each sequence in the full hierarchical path. The dot character '.' is used as the separator between each sequence.

9.3 uvm_sequence_base

The class **uvm_sequence_base** defines the primary interface member functions to create, control and execute the sequences.

9.3.1 Class definition

```
namespace uvm {

  class uvm_sequence_base : public uvm_sequence_item
  {
   public:
    // Constructor
    explicit uvm_sequence_base( const std::string& name );

    // Group: Sequence state
    uvm_sequence_state_enum get_sequence_state() const;
    void wait_for_sequence_state( unsigned int state_mask );

    // Group: Sequence execution
    virtual void start( uvm_sequencer_base* sqr,
                        uvm_sequence_base* parent_sequence = NULL,
                        int this_priority = -1,
                        bool call_pre_post = true );

    virtual void pre_start();
    virtual void pre_body();
    virtual void pre_do( bool is_item );
    virtual void mid_do( uvm_sequence_item* this_item );
    virtual void body();
    virtual void post_do( uvm_sequence_item* this_item );
    virtual void post_body();
    virtual void post_start();

    // Group: Run-time phasing
    uvm_phase* get_starting_phase() const;
    void set_starting_phase( uvm_phase* phase );
    bool get_automatic_phase_objection() const;
    void set_automatic_phase_objection( bool value );

    // Group: Sequence control
    void set_priority( int value );
    int get_priority() const;
    virtual bool is_relevant() const;
    virtual void wait_for_relevant() const;
    void lock( uvm_sequencer_base* sequencer = NULL );
    void grab( uvm_sequencer_base* sequencer = NULL );
    void unlock( uvm_sequencer_base* sequencer = NULL );
    void ungrab( uvm_sequencer_base* sequencer = NULL );
    bool is_blocked() const;
    bool has_lock();
    void kill();
    virtual void do_kill();

    // Group: Sequence item execution
    uvm_sequence_item* create_item( uvm_object_wrapper* type_var,
                                    uvm_sequencer_base* l_sequencer,
                                    const std::string& name );

    virtual void start_item( uvm_sequence_item* item,
                             int set_priority = -1,
                             uvm_sequencer_base* sequencer = NULL );

    virtual void finish_item( uvm_sequence_item* item,
                              int set_priority = -1 );

    virtual void wait_for_grant( int item_priority = -1,
                                 bool lock_request = false );
```

```
    virtual void send_request( uvm_sequence_item* request,
                               bool rerandomize = false );

    virtual void wait_for_item_done( int transaction_id = -1 );

    // Group: Response interface
    void use_response_handler( bool enable );
    bool get_use_response_handler() const;
    virtual void response_handler( const uvm_sequence_item* response );
    void set_response_queue_error_report_disabled( bool value );
    bool get_response_queue_error_report_disabled() const;
    void set_response_queue_depth( int value );
    int get_response_queue_depth() const;
    virtual void clear_response_queue();

  }; // class uvm_sequence_base

} // namespace uvm
```

9.3.2 Constructor

```
explicit uvm_sequence_base( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

9.3.3 Sequence state

9.3.3.1 get_sequence_state

```
uvm_sequence_state_enum get_sequence_state() const;
```

The member function **get_sequence_state** shall return the sequence state as an enumerated value of type **uvm_sequence_state_enum** (see Section 17.4.5). This member function can be used to wait on the sequence reaching or changing from one or more states.

9.3.3.2 wait_for_sequence_state

```
void wait_for_sequence_state( unsigned int state_mask );
```

The member function **wait_for_sequence_state** shall wait until the sequence reaches one of the given states. If the sequence is already in one of these states, the member function shall return immediately.

9.3.4 Sequence execution

9.3.4.1 start

```
virtual void start( uvm_sequencer_base* sqr,
                    uvm_sequence_base* parent_sequence = NULL,
                    int this_priority = -1,
                    bool call_pre_post = true );
```

The member function **start** shall execute the sequence. The argument *sequencer* specifies the sequencer on which to run this sequence. The sequencer shall be compatible with the sequence, that is, the sequencer shall recognize the communicated request and response types.

If *parent_sequence* is not passed as argument or set to NULL, then the sequence is treated as a root sequence, otherwise it is a child of a parent sequence. In the latter case, the parent sequence's member functions **pre_do**, **mid_do**, and **post_do** shall be called during the execution of this sequence.

If *this_priority* is not passed as argument or set to -1, the priority of a sequence is set to priority of its parent sequence. If it is a root (parent) sequence, its default priority is 100. A different priority greater than zero may be specified using this argument. Higher numbers indicate higher priority.

If argument *call_pre_post* is not passed or set to true, then the member functions **pre_body** and **post_body** shall be called before and after calling the member function **body** of the sequence.

9.3.4.2 pre_start

```
virtual void pre_start();
```

The member function **pre_start** shall be provided as a callback for the application that is called before the optional execution of member function **pre_body**. The application shall not call this member function.

9.3.4.3 pre_body

```
virtual void pre_body();
```

The member function **pre_body** shall be provided as a callback for the application that is called before the execution of member function **body**, but only when the sequence is started by using member function **start**. If **start** is called with argument *call_pre_post* set to false, the member function **pre_body** shall not be called. The application shall not call this member function.

9.3.4.4 pre_do

```
virtual void pre_do( bool is_item );
```

The member function **pre_do** shall be provided as a callback for the application that is called on the parent sequence, if the sequence has issued a **wait_for_grant** call and after the sequencer has selected this sequence, and before the item is randomized. The application shall not call this member function.

9.3.4.5 mid_do

```
virtual void mid_do( uvm_sequence_item* this_item );
```

The member function **mid_do** shall be provided as a callback for the application that is called after the sequence item has been randomized, and just before the item is sent to the driver. The application shall not call this member function.

9.3.4.6 body

```
virtual void body();
```

The member function **body** shall be provided as a callback for the application that is called before the optional execution of member function **post_body**. The application shall not call this member function.

NOTE—In an application, the implementation of the sequence resides in this member function.

9.3.4.7 post_do

```
virtual void post_do( uvm_sequence_item* this_item );
```

The member function **post_do** shall be provided as a callback for the application that is called after the driver has indicated that it has completed the sequence item, calling either the member function **item_done** or **put**. The application shall not call this member function.

9.3.4.8 post_body

```
virtual void post_body();
```

The member function **post_body** shall be provided as a callback for the application that is called before the execution of member function **post_start**, but only when the sequence is started by using member function **start**. If **start** is called with argument *call_pre_post* set to false, the member function **post_body** shall not be called. The application shall not call this member function.

9.3.4.9 post_start

```
virtual void post_start();
```

The member function **post_start** shall be provided as a callback for the application that is called after the optional execution of member function **post_body**. The application shall not call this member function.

9.3.5 Run-time phasing

9.3.5.1 get_starting_phase

```
uvm_phase* get_starting_phase() const;
```

The member function **get_starting_phase** shall return 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.

9.3.5.2 set_starting_phase

```
void set_starting_phase( uvm_phase* phase );
```

The member function **set_starting_phase** shall specify the starting phase.

9.3.5.3 get_automatic_phase_objection

```
bool get_automatic_phase_objection() const;
```

The member function **get_automatic_phase_objection** shall return and lock the automatically objection state of the starting phase.

If the member functions returns true, the sequence automatically raises an objection to the starting phase (if the starting phase is not NULL) immediately prior to **pre_start** (see Section 9.3.4.2) being called. The objection is dropped after **post_start** (see Section 9.3.4.9) has executed, or **kill** (see Section 9.3.6.11) has been called.

9.3.5.4 set_automatic_phase_objection

```
void set_automatic_phase_objection( bool value );
```

The member function **set_automatic_phase_objection** shall set the automatically objection state of the starting phase.

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**. In order to simplify this interaction for an application, the implementation shall provide the ability to perform this functionality automatically.

NOTE—An application should not call the member function **set_automatic_phase_objection**(true) if a sequence runs with a forever loop inside of the body, as the objection will never get dropped.

9.3.6 Sequence control

9.3.6.1 set_priority

```
void set_priority( int value );
```

The member function **set_priority** shall set the priority of a sequence. The default priority value for a sequence is 100. Higher values result in higher priorities. When the priority of a sequence is changed, the new priority shall be used by the sequencer the next time that it arbitrates between sequences.

9.3.6.2 get_priority

```
int get_priority() const;
```

The member function **get_priority** shall return the current priority of the sequence.

9.3.6.3 is_relevant

```
virtual bool is_relevant() const;
```

The member function **is_relevant** shall mark a sequence as being relevant or not. By default, the member function **is_relevant** shall return true, indicating that the sequence is always relevant.

An application may choose to overload this member function to indicate to the sequencer that the sequence is not currently relevant after a request has been made. Any sequence that implements the member function **is_relevant** shall also implement **wait_for_relevant**, to enable a sequencer to wait for a sequence to become relevant.

When the sequencer arbitrates, it shall call the member function **is_relevant** on each requesting, unblocked sequence to see if it is relevant. If this member function returns false, then the sequence is not used.

If all requesting sequences are not relevant, then the sequencer shall call **wait_for_relevant** on all sequences and re-arbitrate upon its return.

9.3.6.4 wait_for_relevant

```
virtual void wait_for_relevant() const;
```

The member function shall be called by the sequencer when all available sequences are not relevant. When **wait_for_relevant** returns, the sequencer attempts to re-arbitrate.

Returning from this call does not guarantee that a sequence is relevant, although that would be the ideal. This member function shall provide some delay to prevent an infinite loop.

If a sequence defines **is_relevant** so that it is not always relevant (by default, a sequence is always relevant), then the sequence shall also implement the member function **wait_for_relevant**.

9.3.6.5 lock

```
void lock( uvm_sequencer_base* sequencer = NULL );
```

The member function **lock** shall request a lock on the specified sequencer. If sequencer is NULL, the lock is requested on the current default sequencer. A lock request shall be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence. The lock call shall return when the lock has been granted.

9.3.6.6 grab

```
void grab( uvm_sequencer_base* sequencer = NULL );
```

The member function **grab** shall request a lock on the specified sequencer. If sequencer is NULL, the grab is requested on the current default sequencer. A grab request is put in front of the arbitration queue. It shall be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence. The grab call shall return when the grab has been granted.

9.3.6.7 unlock

```
void unlock( uvm_sequencer_base* sequencer = NULL );
```

The member function **unlock** shall remove any locks or grabs obtained by this sequence on the specified sequencer. If the sequencer is NULL, then the unlock is done on the current default sequencer.

9.3.6.8 ungrab

```
void ungrab( uvm_sequencer_base* sequencer = NULL );
```

The member function **ungrab** shall remove any locks or grabs obtained by this sequence on the specified sequencer. If the sequencer is NULL, then the ungrab is done on the current default sequencer.

9.3.6.9 is_blocked

```
bool is_blocked() const;
```

The member function **is_blocked** shall return a Boolean type indicating whether this sequence is currently prevented from running due to another lock or grab. A true is returned if the sequence is currently blocked. A false is returned if no lock or grab prevents this sequence from executing. Even if a sequence is not blocked, it is possible for another sequence to issue a lock or grab before this sequence can issue a request.

9.3.6.10 has_lock

```
bool has_lock();
```

The member function **has_lock** shall return true if this sequence has a lock; otherwise it shall return false. 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.

9.3.6.11 kill

```
void kill();
```

The member function **kill** shall shall kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed. The sequence state shall be changed to **UVM_STOPPED** and the callback functions **post_body** and **post_start** are not being executed.

9.3.6.12 do_kill

```
virtual void do_kill();
```

The member function **do_kill** shall provide a callback for an application that is called whenever a sequence is terminated by using either **kill** or **stop_sequences**.

9.3.7 Sequence item execution

9.3.7.1 create_item

```
uvm_sequence_item* create_item( uvm_object_wrapper* type_var,
                                uvm_sequencer_base* l_sequencer,
                                const std::string& name );
```

The member function **create_item** shall create and initialize a sequence item of class **uvm_sequence_item** or sequence of class **uvm_sequence** using the factory. The type of the created object, being a sequence item or sequence, is defined by the first argument *type_var*, which shall be of type **uvm_sequence_item** or **uvm_sequence** only. The sequence item or sequence shall be initialized to communicate with the specified sequencer *l_sequencer* passed as second argument. The *name* of the created item shall be passed as third argument.

9.3.7.2 start_item

```
virtual void start_item( uvm_sequence_item* item,
                         int set_priority = -1,
                         uvm_sequencer_base* sequencer = NULL );
```

The member function **start_item** shall initiate execution of a sequence item specified as argument *item*. If the item has not already been initialized using member function **create_item**, then it is initialized here by using the sequencer specified by argument *sequencer*. If argument *sequencer* is not specified or set to NULL, the default sequencer shall be used (see also Section 9.2.3.4). The argument *set_priority* can be used to specify the priority for the execution. If argument *set_priority* is not specified or set to -1, the default priority shall be 100. Randomization, or other member functions, may be done between **start_item** and **finish_item** to ensure late generation.

9.3.7.3 finish_item

```
virtual void finish_item( uvm_sequence_item* item,
                          int set_priority = -1 );
```

The member function **finish_item** shall finalize execution of execution of a sequence item specified as argument *item*. The member function shall be called after **start_item** with no delays or delta-cycles. The argument *set_priority* can be used to specify the priority for the execution. If argument *set_priority* is not specified or set to -1, the default priority shall be 100. Randomization, or other member functions, may be called between **start_item** and **finish_item**.

9.3.7.4 wait_for_grant

```
virtual void wait_for_grant( int item_priority = -1,
                             bool lock_request = false );
```

The member function **wait_for_grant** shall issue a request to the current sequencer. If argument *item_priority* is not specified or set to -1, then the current sequence priority is used by the arbiter. If the argument *lock_request* is set to true, then the sequencer shall issue a lock immediately before granting the sequence.

NOTE—The lock may be granted without the sequence being granted if member function **is_relevant** is not asserted.

9.3.7.5 send_request

```
virtual void send_request( uvm_sequence_item* request,
                           bool rerandomize = false );
```

The member function **send_request** shall send the request item, passed as an argument, to the sequencer, which shall forward it to the driver. If argument *rerandomize* is set to true, the item is randomized before being sent to the driver.

NOTE 1—In an application, the member function **send_request** shall only be called after a call to **wait_for_grant**.

NOTE 2—Randomization is not yet supported in UVM-SystemC.

9.3.7.6 wait_for_item_done

```
virtual void wait_for_item_done( int transaction_id = -1 );
```

The member function **wait_for_item_done** shall block until the driver calls **item_done** or **put**. If no *transaction_id* argument is specified, then the call shall return the next time that the driver calls **item_done** or **put**. If a specific *transaction_id* is specified, then the call shall return when the driver indicates completion of that specific item.

NOTE—If a specific *transaction_id* has been specified, and the driver has already issued an item_done or put for that transaction, then the call hangs, having missed the earlier notification.

9.3.8 Response interface

9.3.8.1 use_response_handler

```
void use_response_handler( bool enable );
```

The member function **use_response_handler** shall send responses to the response handler when argument *enable* is set to true. By default, responses from the driver are retrieved in the sequence by calling member function **get_response**.

9.3.8.2 get_use_response_handler

```
bool get_use_response_handler() const;
```

The member function **get_use_response_handler** shall return the state set by **use_response_handler**. If this member function returns false, the response handler is disabled.

9.3.8.3 response_handler

```
virtual void response_handler( const uvm_sequence_item* response );
```

The member function **response_handler** shall be provided to enable the sequencer, in case returns true, to call this member function for each response that arrives for this sequence.

9.3.8.4 set_response_queue_error_report_disabled

```
void set_response_queue_error_report_disabled( bool value );
```

The member function **set_response_queue_error_report_disabled** shall enable error reporting of overflows of the reponse queue. The response queue shall overflow if more responses are sent to this sequence from the driver than calls to member function **get_response** are made. If argument *value* is set to false, error reporting is disabled. If argument *value* is set to true, error reporting is enabled. By default, if the response queue overflows, an error is reported.

9.3.8.5 get_response_queue_error_report_disabled

```
bool get_response_queue_error_report_disabled() const;
```

The member function **get_response_queue_error_report_disabled** shall return the reporting status of an overflow of the response queue. It returns false when error reports are generated and returns true if no such error reports are generated.

9.3.8.6 set_response_queue_depth

```
void set_response_queue_depth( int value );
```

The member function **set_response_queue_depth** shall set the depth of the reponse queue. The default maximum depth of the response queue is 8. An argument *value* of -1 defines an unbound response queue.

9.3.8.7 get_response_queue_depth

```
int get_response_queue_depth() const;
```

The member function **get_response_queue_depth** shall return the current depth for the response queue. An unbound response queue returns the value -1.

9.3.8.8 clear_response_queue

```
virtual void clear_response_queue();
```

The member function **clear_response_queue** shall empty the response queue for the sequence.

9.4 uvm_sequence

The class **uvm_sequence** extends the base class **uvm_sequence_base** for specific request (REQ) and response (RSP) types, which are specified as template arguments.

9.4.1 Class definition

```
namespace uvm {

  template <typename REQ = uvm_sequence_item, typename RSP = REQ>
  class uvm_sequence : public uvm_sequence_base
  {
   public:
    // Constructor
    explicit uvm_sequence( const std::string& name );

    // Member functions
    void send_request( uvm_sequence_item* request,
                       bool rerandomize = false );

    REQ get_current_item() const;

    virtual void get_response( RSP* response,
                               int transaction_id = -1 );

  }; // class uvm_sequence

} // namespace uvm
```

9.4.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types shall be a derivative of class **uvm_sequence_item**.

9.4.3 Constructor

```
explicit uvm_sequence( const std::string& name );
```

The constructor shall create and initialize an instance of the class with the name *name* passed as an argument.

9.4.4 Member functions

9.4.4.1 send_request

```
void send_request( uvm_sequence_item* request,
                   bool rerandomize = false );
```

The member function **send_request** shall send the request item, passed as an argument, to the sequencer, which shall forward it to the driver. If argument *rerandomize* is set to true, the item is randomized before being sent to the driver.

NOTE 1—In an application, the member function **send_request** shall only be called after a call to wait_for_grant.

NOTE 2—Randomization is not yet supported in UVM-SystemC.

9.4.4.2 get_current_item

```
REQ get_current_item() const;
```

The member function **get_current_item** shall return the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this member function shall return NULL. The sequencer is executing an item from the time that **get_next_item** or **peek** is called until the time that get or **item_done** is called.

NOTE—A driver that only calls **get** will never show a current item, since the item is completed at the same time as it is requested.

9.4.4.3 get_response

```
virtual void get_response( RSP* response,
                           int transaction_id = -1 );
```

The member function **get_response** shall retrieve a response via the response queue. If no response is available in the response queue, the member function blocks until a response is received.

If no *transaction_id* is passed as an argument, this member function shall return the next response sent to this sequence. If a *transaction_id* is specified, the member function shall block until a response with that transaction ID is received in the response queue.

10. Configuration and resource classes

The configuration and resource classes provide access to a centralized database where type specific information can be stored and retrieved. A configuration or resource item may be associated with a specific hierarchical scope of an object derived from class **uvm_component** or it may be visible to all components regardless of their hierarchical position.

The following configuration and resource classes are defined:

- **uvm_config_db**
- **uvm_resource_db**
- **uvm_resource_db_options**
- **uvm_resource_options**
- **uvm_resource_base**
- **uvm_resource_pool**
- **uvm_resource**
- **uvm_resource_types**

10.1 uvm_config_db

The class **uvm_config_db** provides a typed interface for object-centric configuration. It is consistent with the configuration mechanism as defined for the class **uvm_component**. Information can be read from or written to the database at any time during simulation.

10.1.1 Class definition

```
namespace uvm {

  template <class T>
  class uvm_config_db
  {
   public:

    // Constructor
    uvm_config_db();

    // Member functions
    static void set( uvm_component* cntxt,
                     const std::string& inst_name,
                     const std::string& field_name,
                     const T& value );

    static bool get( uvm_component* cntxt,
                     const std::string& inst_name,
                     const std::string& field_name,
                     T& value );

    static bool exists( uvm_component* cntxt,
                        const std::string& inst_name,
                        const std::string& field_name,
                        bool spell_chk = false );

    static void wait_modified( uvm_component* cntxt,
                               const std::string& inst_name,
                               const std::string& field_name );

  }; // class uvm_config_db

} // namespace uvm
```

10.1.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the configuration database.

10.1.3 Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm_config_db** are static, so these are called using the **operator::**.

10.1.4 Member functions

10.1.4.1 set

```
static void set( uvm_component* cntxt,
                 const std::string& inst_name,
                 const std::string& field_name,
                 const T& value );
```

The member function **set** shall create a new or update an existing configuration setting using target field *field_name* in instance with name *inst_name* from the context *cntxt* in which it is defined. If argument *cntxt* is set to NULL, then *inst_name* defines the complete scope for the configuration setting; otherwise, the full name of the component referenced to by *cntxt* shall be added to the instance name. An application may define *inst_name* and *field_name* to be glob-style or regular expression style expressions.

10.1.4.2 get

```
static bool get( uvm_component* cntxt,
                 const std::string& inst_name,
                 const std::string& field_name,
                 T& value );
```

The member function **get** shall retrieve a configuration setting via arguments *inst_name* and *field_name*, using a component pointer *cntxt* as the starting search point. The argument *inst_name* shall be an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. The argument *field_name* is the specific field in the scope that is being searched for.

The member function returns true if the value is being found; otherwise, false is returned.

10.1.4.3 exists

```
static bool exists( uvm_component* cntxt,
                    const std::string& inst_name,
                    const std::string& field_name,
                    bool spell_chk = false );
```

The member function **exists** shall check if a value for *field_name* is available in *inst_name*, using 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 that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for. The argument *spell_chk* can be set to true to turn spell checking on if it is expected that the field should exist in the database. The function returns true if a config parameter exists and false if it does not exist.

10.1.4.4 wait_modified

```
static void wait_modified( uvm_component* cntxt,
                           const std::string& inst_name,
                           const std::string& field_name );
```

The member function **wait_modified** shall wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*. The member function blocks until a new configuration setting is applied that effects the specified field.

10.2 uvm_resource_db

The class **uvm_resource_db** provides a convenience interface for the resources facility. In many cases basic operations such as creating and setting a resource or getting a resource could take multiple lines of code using the interfaces in class **uvm_resource_base** or class **uvm_resource**. The convenience layer in class **uvm_resource_db** reduces many of those operations to a single line of code.

10.2.1 Class definition

```
namespace uvm {

  template < typename T = uvm_object* >
  class uvm_resource_db
  {
   public:

    // Member functions
    static uvm_resource<T>* get_by_type( const std::string& scope );

    static uvm_resource<T>* get_by_name( const std::string& scope,
                                         const std::string& name,
                                         bool rpterr = true );

    static uvm_resource<T>* set_default( const std::string& scope,
                                         const std::string& name );

    static void set( const std::string& scope,
                     const std::string& name,
                     const T& val,
                     uvm_object* accessor = NULL );

    static void set_anonymous( const std::string& scope,
                               const T& val,
                               uvm_object* accessor = NULL );

    static bool read_by_name( const std::string& scope,
                              const std::string& name,
                              T val,
                              uvm_object* accessor = NULL );

    static bool read_by_type( const std::string& scope,
                              T val,
                              uvm_object* accessor = NULL );

    static bool write_by_name( const std::string& scope,
                               const std::string& name,
                               const T& val,
                               uvm_object* accessor = NULL );

    static bool write_by_type( const std::string& scope,
                               const T& val,
                               uvm_object* accessor = NULL );

    static void dump();

   private:
    // disabled
    uvm_resource_db();
```

```
  }; // class uvm_config_db

} // namespace uvm
```

10.2.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

10.2.3 Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm_resource_db** are static, so these shall be called using the **operator::**. An application shall not instantiate this class, but shall call the static member functions directly.

10.2.4 Member functions

10.2.4.1 get_by_type

```
static uvm_resource<T>* get_by_type( const std::string& scope );
```

The member function **get_by_type** shall return the resource by type. The type is specified in the database class parameter so the only argument to this member function is the scope.

10.2.4.2 get_by_name

```
static uvm_resource<T>* get_by_name( const std::string& scope,
                                     const std::string& name,
                                     bool rpterr = true );
```

The member function **get_by_name** shall return the resource by name. The first argument is the current scope and the second argument is the name of the resource to be retrieved. If the argument *rpterr* is set to true, a warning shall be generated if no matching resource is found.

10.2.4.3 set_default

```
static uvm_resource<T>* set_default( const std::string& scope,
                                     const std::string& name );
```

The member function **set_default** shall create a new resource with a default value and add it to the resource database using arguments *name* and *scope* as the lookup parameters.

10.2.4.4 set

```
static void set( const std::string& scope,
                 const std::string& name,
                 const T& val,
                 uvm_object* accessor = NULL );
```

The member function **set** shall create a new resource, write a value *val* to it, and add it to the resource database using arguments *name* and *scope* as the lookup parameters. The argument *accessor* is used for auditing.

10.2.4.5 set_anonymous

```
static void set_anonymous( const std::string& scope,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **set_anonymous** shall create a new resource, write a value *val* to it, and add it to the resource database. As the resource has no argument *name*, it is not added to the name map. But is does have an argument *scope* for lookup purposes. The argument *accessor* is used for auditing.

10.2.4.6 read_by_name

```
static bool read_by_name( const std::string& scope,
                          const std::string& name,
                          T val,
                          uvm_object* accessor = NULL );
```

The member function **read_by_name** shall locate a resource by arguments *name* and *scope* and returns the value through argument *val*. The member function shall return true if the read was successful; otherwise it shall return false. The argument *accessor* is used for auditing.

10.2.4.7 read_by_type

```
static bool read_by_type( const std::string& scope,
                          T val,
                          uvm_object* accessor = NULL );
```

The member function **read_by_type** shall read a value by type. The value is returned through the argument *val*. The argument *scope* is used for the lookup. The member function shall return true if the read was successful; otherwise it shall return false. The argument *accessor* is used for auditing.

10.2.4.8 write_by_name

```
static bool write_by_name( const std::string& scope,
                           const std::string& name,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **write_by_name** shall write the argument *val* into the resources database. First, look up the resource by using arguments *name* and *scope*. If it is not located then add a new resource to the database and then write its value.

10.2.4.9 write_by_type

```
static bool write_by_type( const std::string& scope,
                           const T& val,
                           uvm_object* accessor = NULL );
```

The member function **write_by_type** shall write the argument *val* into the resources database. First, look up the resource by type. If it is not located then add a new resource to the database and then write its value.

Because the scope is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the scope argument. If a **get_by_name** match is found for name and scope then *val* shall be written to that matching resource and thus may impact other scopes which also match the resource.

10.2.4.10 dump

```
static void dump();
```

The member function **dump** shall dump all the resources in the resource pool. This is useful for debugging purposes. This member function does not use the parameter T, so it shall dump the same thing (the entire database) no matter the value of the parameter.

10.3 uvm_resource_db_options

The class **uvm_resource_db_options** shall provide a namespace for managing options for the resources database facility. The class shall define static member functions for manipulating and retrieving the value of the data members. The static data members represent options and settings that control the behavior of the resources database facility.

10.3.1 Class definition

```
namespace uvm {

  class uvm_resource_db_options
  {
   public:

    // Member functions
    static void turn_on_tracing();
    static void turn_off_tracing();
    static bool is_tracing();

   private:
    // Disabled
    uvm_resource_db_options();

  }; // class uvm_resource_db_options

} // namespace uvm
```

10.3.2 Member functions

10.3.2.1 turn_on_tracing

```
static void turn_on_tracing();
```

The member function **turn_on_tracing** shall enable tracing for the resource database. This causes all reads and writes to the database to display information about the accesses.

10.3.2.2 turn_off_tracing

```
static void turn_off_tracing();
```

The member function **turn_off_tracing** shall disable tracing for the resource database.

10.3.2.3 is_tracing

```
static bool is_tracing();
```

The member function **is_tracing** shall return true if the tracing facility is enabled; otherwise it shall return false.

10.4 uvm_resource_options

The class **uvm_resource_options** shall provide a namespace for managing options for the resources facility. The class shall only provide static member functions for manipulating and retrieving the value of its data members.

10.4.1 Class definition

```
namespace uvm {

  class uvm_resource_options
  {
   public:

    // Member functions
    static void turn_on_auditing();
    static void turn_off_auditing();
    static bool is_auditing();

   private:
    // disabled
    uvm_resource_options();

  }; // class uvm_resource_options

} // namespace uvm
```

10.4.2 Member functions

10.4.2.1 turn_on_auditing

```
static void turn_on_auditing();
```

The member function **turn_on_auditing** shall enable auditing for the resource database. This causes all reads and writes to the database to store information about the accesses. Auditing is enabled by default.

10.4.2.2 turn_off_auditing

```
static void turn_off_auditing();
```

The member function **turn_off_auditing** shall disable auditing for the resource database. If auditing is disabled, it is not possible to get extra information about resource database accesses.

10.4.2.3 is_auditing

```
static bool is_auditing();
```

The member function **is_auditing** shall return true if auditing is enabled; otherwise it shall return false.

10.5 uvm_resource_base

The class **uvm_resource_base** shall provide a non-parameterized base class for resources. It supports interfaces for scope matching and virtual member functions for printing the resource and accessors list.

10.5.1 Class definition

```
namespace uvm {
```

```
  class uvm_resource_base : public uvm_object
  {
   public:

    // Constructor
    uvm_resource_base( const std::string& name = "",
                       const std::string& scope = "*" );

    // Group: Resource database interface
    virtual uvm_resource_base* get_type_handle() const = 0;

    // Group: Read-only interface
    void set_read_only();
    bool is_read_only() const;

    // Group: Notification
    void wait_modified();

    // Group: Scope interface
    void set_scope( const std::string* scope );
    std::string get_scope() const;
    bool match_scope( const std::string& scope );

    // Group: Priority
    virtual void set_priority( uvm_resource_types::priority_e pri ) = 0;

    // Group: Utility functions
    void do_print( const uvm_printer& printer ) const;

    // Group: Audit trail
    void record_read_access( uvm_object* accessor = NULL );
    void record_write_access( uvm_object* accessor = NULL );
    virtual void print_accessors() const;
    void init_access_record( uvm_resource_types::access_t access_record );

    // Data members
    unsigned int precedence;
    static int unsigned default_precedence;

  }; // class uvm_resource_base

} // namespace uvm
```

10.5.2 Constructor

```
uvm_resource_base( const std::string& name = "",
                   const std::string& scope = "*" );
```

The constructor takes two arguments, the name of the resource *name* and a regular expression *scope* which represents the set of scopes over which this resource is visible.

10.5.3 Resource database interface

10.5.3.1 get_type_handle

```
virtual uvm_resource_base* get_type_handle() const = 0;
```

The member function **get_type_handle** shall return the type handle of the resource container.

10.5.4 Read-only interface

10.5.4.1 set_read_only

```
void set_read_only();
```

The member function **set_read_only** shall define the resource as a read-only resource. An attempt to call **uvm_resource**<T>::**write** on the resource shall cause an error.

10.5.4.2 is_read_only

```
bool is_read_only() const;
```

The member function **is_read_only** shall return true if this resource has been set to read-only; otherwise it shall return false.

10.5.5 Notification

10.5.5.1 wait_modified

```
void wait_modified();
```

The member function **wait_modified** shall block execution until the resource has been modified, that is, it waits till a **uvm_resource**<T>::**write** operation has been performed.

10.5.6 Scope interface

10.5.6.1 set_scope

```
void set_scope( const std::string& scope );
```

The member function **set_scope** shall set the value of the regular expression that identifies the set of scopes over which this resource is visible. If the supplied argument is a glob it shall be converted to a regular expression before it is stored.

10.5.6.2 get_scope

```
std::string get_scope() const;
```

The member function **get_scope** shall retrieve the regular expression string that identifies the set of scopes over which this resource is visible.

10.5.6.3 match_scope

```
bool match_scope( const std::string& scope );
```

The member function **match_scope** shall return true if this resource is visible in a scope. The scope is specified as argument and may use regular expressions.

10.5.7 Priority

10.5.7.1 set_priority

```
virtual void set_priority( uvm_resource_types::priority_e pri ) = 0;
```

The member function **set_priority** shall change the search priority of the resource based on the value of the priority enumeration given as argument.

10.5.8 Utility functions

10.5.8.1 do_print

```
void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall be called by member function **print**. It allows an application to implement application-specific printing routines.

10.5.9 Audit trail

10.5.9.1 record_read_access

```
void record_read_access( uvm_object* accessor = NULL );
```

The member function **record_read_access** shall record the read access for this resource.

10.5.9.2 record_write_access

```
void record_write_access( uvm_object* accessor = NULL );
```

The member function **record_write_access** shall record the write access for this resource.

10.5.9.3 print_accessors

```
virtual void print_accessors() const;
```

The member function **print_accessors** shall print the access records for this resource.

10.5.9.4 init_access_record

```
void init_access_record( uvm_resource_types::access_t access_record );
```

The member function **init_access_record** shall initialize a new access record.

10.5.10 Data members

10.5.10.1 precedence

```
unsigned int precedence;
```

The data member **precedence** shall be used to associate a precedence that a resource has with respect to other resources which match the same scope and name. Resources are set to the **default_precedence** initially, and may be set to a higher or lower precedence as desired.

10.5.10.2 default_precedence

```
static int unsigned default_precedence;
```

The data member **default_precedence** is the default precedence for a resource that has been created. When two resources have the same precedence, the first resource found has precedence.

10.6 uvm_resource_pool

The class **uvm_resource_pool** shall provide the centralized resource pool to store each resource both by primary name and by type handle.

10.6.1 Class definition

```
namespace uvm {

  class uvm_resource_pool
  {
   public:
    static uvm_resource_pool* get();
    bool spell_check( const std::string& s ) const;

    // Group: Set interface
    void set( uvm_resource_base* rsrc, int override = 0 );
    void set_override( uvm_resource_base* rsrc );
    void set_name_override( uvm_resource_base* rsrc );
    void set_type_override( uvm_resource_base* rsrc );

    // Group: Lookup
    uvm_resource_types::rsrc_q_t* lookup_name( const std::string& scope,
                                               const std::string& name,
                                               uvm_resource_base* type_handle,
                                               bool rpterr = true ) const;

    uvm_resource_base* get_highest_precedence( uvm_resource_types::rsrc_q_t* q ) const;

    static void sort_by_precedence( uvm_resource_types::rsrc_q_t* q );

    uvm_resource_base* get_by_name( const std::string& scope,
                                    const std::string& name,
                                    uvm_resource_base* type_handle,
                                    bool rpterr = true );

    uvm_resource_types::rsrc_q_t* lookup_type( const std::string& scope,
                                               uvm_resource_base* type_handle ) const;

    uvm_resource_base* get_by_type( const std::string& scope,
                                    uvm_resource_base* type_handle );

    uvm_resource_types::rsrc_q_t* lookup_regex_names( const std::string& scope,
                                                      const std::string& name,
                                                      uvm_resource_base* type_handle = NULL );

    uvm_resource_types::rsrc_q_t* lookup_regex( const std::string& re,
                                                const std::string& scope );

    uvm_resource_types::rsrc_q_t* lookup_scope( const std::string& scope );

    // Group: Priority interface
    void set_priority_type( uvm_resource_base* rsrc,
                            uvm_resource_types::priority_e pri );

    void set_priority_name( uvm_resource_base* rsrc,
                            uvm_resource_types::priority_e pri );

    void set_priority( uvm_resource_base* rsrc,
                       uvm_resource_types::priority_e pri );

    // Group: Debug
    uvm_resource_types::rsrc_q_t* find_unused_resources() const;
    void print_resources( uvm_resource_types::rsrc_q_t rq, bool audit = false ) const;
    void dump( bool audit = false ) const;

  }; // class uvm_resource_pool

} // namespace uvm
```

10.6.2 get

```
static uvm_resource_pool* get();
```

The member function **get** shall return the singleton handle to the resource pool.

10.6.3 spell_check

```
bool spell_check( const std::string& s ) const;
```

The member function **spell_check** shall invoke the spell checker for the string *s* passed as argument. The universe of correctly spelled strings—i.e. the dictionary—is the name map.

10.6.4 Set interface

10.6.4.1 set

```
void set( uvm_resource_base* rsrc, int override = 0 );
```

The member function **set** shall add a new resource to the resource pool. The resource is inserted into both the name map and type map so it can be located by either.

An object creates a resource and sets it into the resource pool. Later, other objects that want to access the resource shall get it from the pool.

Overrides can be specified using this interface. Either a name override, a type override or both can be specified. If an override is specified, then the resource is entered at the front of the queue instead of at the back.

It is not recommended that an application specify the override parameter directly. Instead, an application should use the member functions **set_override**, **set_name_override**, or **set_type_override**.

10.6.4.2 set_override

```
void set_override( uvm_resource_base* rsrc );
```

The member function **set_override** shall override the resource, provided as an argument, in the resource pool both by name and type.

10.6.4.3 set_name_override

```
void set_name_override( uvm_resource_base* rsrc );
```

The member function **set_name_override** shall override the resource, provided as argument *rsrc*, in the resource pool using normal precedence in the type map and shall override the name.

10.6.4.4 set_type_override

```
void set_type_override( uvm_resource_base* rsrc );
```

The member function **set_type_override** shall override the resource, provided as argument *rsrc*, in the resource pool using normal precedence in the name map and shall override the type.

10.6.5 Lookup

10.6.5.1 lookup_name

```
uvm_resource_types::rsrc_q_t* lookup_name( const std::string& scope,
                                           const std::string& name,
                                           uvm_resource_base* type_handle,
                                           bool rpterr = true ) const;
```

The member function **lookup_name** shall return a queue of resources that match the *name*, *scope*, and *type_handle*, which are passed as arguments. If no resources match the queue is returned empty. If *rpterr* is set, then a warning is issued if no matches are found, and the spell checker is invoked on *name*. If *type_handle* is NULL, then a type check is not made and resources are returned that match only *name* and *scope*.

10.6.5.2 get_highest_precedence

```
uvm_resource_base* get_highest_precedence( uvm_resource_types::rsrc_q_t* q ) const;
```

The member function **get_highest_precedence** shall traverse the queue passes as argument, *q*, of resources and return the one with the highest precedence. In the case where there exists more than one resource with the highest precedence value, the first one that has that precedence shall be the one that is returned.

10.6.5.3 sort_by_precedence

```
static void sort_by_precedence( uvm_resource_types::rsrc_q_t* q );
```

The member function **sort_by_precedence** shall sort the resources, passed as argument as a list of resources, in precedence order. The highest precedence resource shall be first in the list and the lowest precedence shall be last. Resources that have the same precedence and the same name shall be ordered by most recently set first.

10.6.5.4 get_by_name

```
uvm_resource_base* get_by_name( const std::string& scope,
                                const std::string& name,
                                uvm_resource_base* type_handle,
                                bool rpterr = true );
```

The member function **get_by_name** shall return the resource by using the arguments *name*, *scope*, and *type_handle*. Whether the get succeeds or fails, save a record of the get attempt. If the argument *rpterr* is set to true, the member function shall report potential errors.

10.6.5.5 lookup_type

```
uvm_resource_types::rsrc_q_t* lookup_type( const std::string& scope,
                                           uvm_resource_base* type_handle ) const;
```

The member function **lookup_type** shall return a queue of resources that match the argument *type_handle* and argument *scope*. If no resources match, then the returned queue is empty.

10.6.5.6 get_by_type

```
uvm_resource_base* get_by_type( const std::string& scope,
                                uvm_resource_base* type_handle );
```

The member function **get_by_type** shall return the resources that match the argument *type_handle* and argument *scope*. It shall insert a record into the get history list whether or not the get succeeded.

10.6.5.7 lookup_regex_names

```
uvm_resource_types::rsrc_q_t* lookup_regex_names( const std::string& scope,
                                                  const std::string& name,
                                                  uvm_resource_base* type_handle = NULL );
```

The member function **lookup_regex_names** shall return a queue of resources that match the arguments *name*, *scope*, and *type_handle*, where *name* and *scope* may be expressed as a regular expression.

10.6.5.8 lookup_regex

```
uvm_resource_types::rsrc_q_t* lookup_regex( const std::string& re,
                                            const std::string& scope );
```

The member function **lookup_regex** shall return a queue of resources that whose name matches the regular expression argument *re* and whose scope matches the specified argument *scope*.

10.6.5.9 lookup_scope

```
uvm_resource_types::rsrc_q_t* lookup_scope( const std::string& scope );
```

The member function **lookup_scope** shall return a queue of resources that are visible to a particular *scope*.

NOTE—This member function could be quite computation expensive, as it has to traverse all of the resources in the resource database.

10.6.6 Priority interface

10.6.6.1 set_priority_type

```
void set_priority_type( uvm_resource_base* rsrc,
                        uvm_resource_types::priority_e pri );
```

The member function **set_priority_type** shall change the priority of the resource *rsrc* in the resource type map only, based on the value of priority enumeration argument *pri*. The priority in the resource name map remains unchanged.

10.6.6.2 set_priority_name

```
void set_priority_name( uvm_resource_base* rsrc,
                        uvm_resource_types::priority_e pri );
```

The member function **set_priority_name** shall change the priority of the resource *rsrc* in the resource name map only, based on the value of priority enumeration argument *pri*. The priority in the resource type map remains unchanged.

10.6.6.3 set_priority

```
void set_priority( uvm_resource_base* rsrc,
                   uvm_resource_types::priority_e pri );
```

The member function **set_priority** shall change the priority of the resource *rsrc* in the resource name map and type map, based on the value of priority enumeration argument *pri*.

10.6.7 Debug

10.6.7.1 find_unused_resources

```
uvm_resource_types::rsrc_q_t* find_unused_resources() const;
```

The member function **find_unused_resources** shall return a queue of resources that have at least one write and no reads.

10.6.7.2 print_resources

```
void print_resources( uvm_resource_types::rsrc_q_t rq, bool audit = false ) const;
```

The member function **print_resources** shall print the queue of resources passed as argument *rq*. If the argument *audit* is set to true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

10.6.7.3 dump

```
void dump( bool audit = false ) const;
```

The member function **dump** shall print the entire resource pool. The member function **print_resources** shall be used to initiate the printing. If the argument *audit* is set to true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

10.7 uvm_resource

The class **uvm_resource** shall provide the interface to read and write to the resource database.

10.7.1 Class definition

```
namespace uvm {

  template <typename T = int>
  class uvm_resource : public uvm_resource_base
  {
   public:

    // Group: Type Interface
    static uvm_resource<T>* get_type();
    uvm_resource_base* get_type_handle() const;

    // Group: Set/Get Interface
    void set();
    void set_override( uvm_resource_types::override_t override =
                       uvm_resource_types::BOTH_OVERRIDE );

    static uvm_resource<T>* get_by_name( const std::string& scope,
                                         const std::string& name,
                                         bool rpterr = true );

    static uvm_resource<T>* get_by_type( const std::string& scope,
                                         uvm_resource_base* type_handle );

    // Group: Read/Write Interface
    T read( uvm_object*& accessor );
```

```
    void write( const T& t, uvm_object*& accessor );

    // Group: Priority
    void set_priority( uvm_resource_types::priority_e pri );
    static uvm_resource<T>* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );

  }; // class uvm_resource

} // namespace uvm
```

10.7.2 Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

10.7.3 Type interface

10.7.3.1 get_type

```
static uvm_resource<T>* get_type();
```

The member function **get_type** shall return the static type handle. The return type is the type of the parameterized class.

10.7.3.2 get_type_handle

```
uvm_resource_base* get_type_handle() const;
```

The member function **get_type_handle** shall return the static type handle of this resource in a polymorphic fashion. The return type of **get_type_handle** is **uvm_resource_base**.

NOTE—As the member function is not static, it can only be used by instances of a parameterized resource.

10.7.4 Set/Get interface

10.7.4.1 set

```
void set();
```

The member function **set** shall put the resource into the global resource pool.

10.7.4.2 set_override

```
void set_override( uvm_resource_types::override_t override =
                      uvm_resource_types::BOTH_OVERRIDE );
```

The member function **set_override** shall put the resource into the global resource pool as an override. This means it gets put at the head of the list and is searched before other existing resources that occupy the same position in the name map or the type map. The default is to override both the name and type maps. However, using the override argument you can specify that either the name map or type map is overridden.

10.7.4.3 get_by_name

```
static uvm_resource<T>* get_by_name( const std::string& scope,
                                     const std::string& name,
```

```
                                     bool rpterr = true );
```

The member function **get_by_name** shall look up a resource by name in the name map. The first resource with the specified name, whose type is the current type, and is visible in the specified scope is returned, if one exists. The rpterr flag indicates whether or not an error should be reported if the search fails. If the argument *rpterr* is set to one then a failure message is issued, including suggested spelling alternatives, based on resource names that exist in the database, gathered by the spell checker.

10.7.4.4 get_by_type

```
static uvm_resource<T>* get_by_type( const std::string& scope,
                                     uvm_resource_base* type_handle );
```

The member function **get_by_type** shall look up a resource by *type_handle* in the type map. The first resource with the specified *type_handle* that is visible in the specified scope is returned, if one exists. The member function shall return NULL if there is no resource matching the specifications.

10.7.5 Read/Write interface

10.7.5.1 read

```
T read( uvm_object*& accessor );
```

The member function **read** shall return the object stored in the resource container. If an accessor object is supplied then also update the accessor record for this resource.

10.7.5.2 write

```
void write( const T& t, uvm_object*& accessor );
```

The member function **write** shall modify the object stored in this resource container. If the resource is read-only then issue an error message and return without modifying the object in the container. If the resource is not read-only and an accessor object has been supplied then also update the accessor record. Lastly, replace the object value in the container with the value supplied as the argument, t, and release any processes blocked on **uvm_resource_base**::**wait_modified**.

10.7.6 Priority interface

10.7.6.1 set_priority

```
void set_priority( uvm_resource_types::priority_e pri );
```

The member function **set_priority** shall change the search priority of the resource based on the value of the priority enum argument, *pri*.

10.7.6.2 get_highest_precedence

```
static uvm_resource<T>* get_highest_precedence( uvm_resource_types::rsrc_q_t* q );
```

The member function **get_highest_precedence** shall locate the first resource, in a queue of resources, with the highest precedence whose type is T.

10.8 uvm_resource_types

The class **uvm_resource_types** shall provide typedefs and enums used throughout the resources facility. This class shall not contain any member function or data members, only typedefs. It's used in lieu of package-scope types.

10.8.1 Class definition

```
namespace uvm {

  class uvm_resource_types
  {
   public:

    typedef std::queue<uvm_resource_base* > rsrc_q_t;
    typedef enum { TYPE_OVERRIDE, NAME_OVERRIDE, BOTH_OVERRIDE } override_t;
    typedef enum { PRI_HIGH, PRI_LOW } priority_e;

  }; // class uvm_resource_types

} // namespace uvm
```

10.8.2 Type definitions (typedefs)

10.8.2.1 rsrc_q_t

The typedef **rsrc_q_t** shall define a queue of handles of type **uvm_resource_base**.

10.8.2.2 override_t

The typedef **override_t** shall define an enumeration to override a resource. Valid values are:

— **TYPE_OVERRIDE**: Override a resource in the resource pool both by type.

— **NAME_OVERRIDE**: Override a resource in the resource pool both by name.

— **BOTH_OVERRIDE**: Override a resource in the resource pool both by name and type.

10.8.2.3 priority_e

The typedef **priority_e** shall define an enumeration for the priority of a resource. Valid values are:

— **PRI_HIGH**: High priority, which places the resource at the front of the queue.

— **PRI_LOW**: Low priority, which places the resource at the back of the queue.

11. Phasing and synchronization classes

The phasing and synchronization concept in UVM defines standardized stages called *phases* which are executed in a well defined order. Each UVM component offers dedicated callbacks for each of these phases to implement application-specific behavior. Phases are executed sequentially, but each phase may consist of multiple function calls (of components contributing to that phase) in parallel. Besides standardized common and UVM run-time phases, user-defined phases can be added.

In order to support synchronization during the execution of the run-time phases, which run as concurrent processes, additional methods are available to coordinate the execution of or status of these processes between all UVM components or objects.

The following phasing and synchronization classes are defined:

- **uvm_phase**: The base class for defining a phase's behavior, state, context.
- **uvm_domain**: Phasing schedule node representing an independent branch of the schedule.
- **uvm_bottomup_phase**: A phase implementation for bottom up function phases.
- **uvm_topdown_phase**: A phase implementation for top-down function phases.
- **uvm_process_phase**° (**uvm_task_phase**†): A phase implementation for phases which are launched as spawned processes.
- **uvm_objection**: Mechanism to synchronize phases based on passing execution status information between running processes.

11.1 uvm_phase

The class **uvm_phase** shall provide the base class for the UVM phasing mechanism.

11.1.1 Class definition

```
namespace uvm {

  class uvm_phase : public uvm_object
  {
   public:
    // Constructor
    explicit uvm_phase( const std::string& name,
                        uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
                        uvm_phase* parent = NULL );

    uvm_phase_type get_phase_type() const;

    // Group: State
    uvm_phase_state get_state() const;

    int get_run_count() const;
    uvm_phase* find_by_name( const std::string& name, bool stay_in_scope = true ) const;
    uvm_phase* find( const uvm_phase* phase, bool stay_in_scope = true ) const;
    bool is( const uvm_phase* phase ) const;
    bool is_before( const uvm_phase* phase ) const;
    bool is_after( const uvm_phase* phase ) const;

    // Group: Callbacks
    virtual void exec_func( uvm_component* comp, uvm_phase* phase );
    virtual void exec_process( uvm_component* comp, uvm_phase* phase );

    // Group: Schedule
    void add( uvm_phase* phase,
              uvm_phase* with_phase = NULL,
              uvm_phase* after_phase = NULL,
              uvm_phase* before_phase = NULL);
```

```
    uvm_phase* get_parent() const;
    virtual const std::string get_full_name() const;
    uvm_phase* get_schedule( bool hier = false ) const;
    std::string get_schedule_name( bool hier = false ) const;
    uvm_domain* get_domain() const;
    std::string get_domain_name() const;
    uvm_phase* get_imp() const;

    // Group: Objection
    uvm_objection* get_objection() const;
    virtual void raise_objection( uvm_object* obj,
                                  const std::string& description = "",
                                  int count = 1 );

    virtual void drop_objection( uvm_object* obj,
                                 const std::string& description = "",
                                 int count = 1 );

    // Group: Synchronization
    void sync( uvm_domain* target,
               uvm_phase* phase = NULL,
               uvm_phase* with_phase = NULL );

    void unsync( uvm_domain* target,
                 uvm_phase* phase = NULL,
                 uvm_phase* with_phase = NULL );

    void wait_for_state( uvm_phase_state state, uvm_wait_op op = UVM_EQ );

    // Group: Jumping
    void jump( const uvm_phase* phase );
    uvm_phase* get_jump_target() const;
  }; // class uvm_phase

} // namespace uvm
```

11.1.2 Construction

11.1.2.1 Constructor

```
explicit uvm_phase( const std::string& name,
                    uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
                    uvm_phase* parent = NULL );
```

The constructor shall create a new phase node, using the arguments *name*, the type name of type *type_name* and optionally the pointer to the parent phase *parent*, as argument.

11.1.2.2 get_phase_type

```
uvm_phase_type get_phase_type() const;
```

The member function **get_phase_type** shall return the phase type as defined by **uvm_phase_type** (see Section 17.4.6).

11.1.3 State

11.1.3.1 get_state

```
uvm_phase_state get_state() const;
```

The member function **get_state** shall return the current state of this phase.

11.1.3.2 uvm_phase_get_run_count

```
int get_run_count() const;
```

The member function **get_run_count** shall return the integer number of times this phase has executed.

11.1.3.3 find_by_name

```
uvm_phase* find_by_name( const std::string& name,
                         bool stay_in_scope = true ) const;
```

The member function **find_by_name** shall locate a phase node with the specified *name* and return its handle. If argument *stay_in_scope* is set to true, it searches only within this phase's schedule or domain.

11.1.3.4 find

```
uvm_phase* find( const uvm_phase* phase,
                 bool stay_in_scope = true ) const;
```

The member function **find** shall locate the phase node with the specified phase implementation and return its handle. If argument *stay_in_scope* is set to true, it searches only within this phase's schedule or domain.

11.1.3.5 is

```
bool is( const uvm_phase* phase ) const;
```

The member function **is** shall return true if the containing **uvm_phase** refers to the same phase as the phase argument; otherwise it shall return false.

11.1.3.6 is_before

```
bool is_before( const uvm_phase* phase ) const;
```

The member function **is_before** shall return true if the containing **uvm_phase** refers to a phase that is earlier than the phase argument; otherwise it shall return false.

11.1.3.7 is_after

```
bool is_after( const uvm_phase* phase ) const;
```

The member function **is_after** shall return true if the containing **uvm_phase** refers to a phase that is later than the phase argument; otherwise it shall return false.

11.1.4 Callbacks

11.1.4.1 exec_func

```
virtual void exec_func( uvm_component* comp, uvm_phase* phase );
```

The member function **exec_func** shall implement the functor/delegate functionality for a function phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call.

11.1.4.2 exec_process° (exec_task†)

```
virtual void exec_process( uvm_component* comp, uvm_phase* phase );
```

The member function **exec_process**° shall implement the functor/delegate functionality for a task phase type comp—the component to execute the functionality upon phase—the phase schedule that originated this phase call.

NOTE—The member function was called exec_task in UVM in SystemVerilog, but has been renamed in line with SystemC processes.

11.1.5 Schedule

11.1.5.1 add

```
void add( uvm_phase* phase,
          uvm_phase* with_phase = NULL,
          uvm_phase* after_phase = NULL,
          uvm_phase* before_phase = NULL);
```

The member function **add** shall build a schedule structure, inserting phase by phase, specifying linkage. Phases can be added anywhere, in series or parallel with existing nodes. The argument *phase* is the handle of a singleton derived phase implementation containing actual functor. By default the new phase shall be appended to the schedule. When argument *with_phase* is passed, the new phase shall be added in parallel to the actual phase. When argument *after_phase* is passed, the new phase shall be added as successor to the actual phase. When the argument *before_phase* is passed, the new phase shall be added as predecessor to the actual phase.

11.1.5.2 get_parent

```
uvm_phase* get_parent() const;
```

The member function **get_parent** shall return the parent schedule node, if any, for hierarchical graph traversal.

11.1.5.3 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the full path from the enclosing domain down to this node. The singleton phase implementations have no hierarchy.

11.1.5.4 get_schedule

```
uvm_phase* get_schedule( bool hier = false ) const;
```

The member function **get_schedule** shall return the topmost parent schedule node, if any, for hierarchical graph traversal.

11.1.5.5 get_schedule_name

```
std::string get_schedule_name( bool hier = false ) const;
```

The member function **get_schedule_name** shall return the schedule name associated with this phase node.

11.1.5.6 get_domain

```
uvm_domain* get_domain() const;
```

The member function **get_domain** shall return the enclosing domain.

11.1.5.7 get_domain_name

```
std::string get_domain_name() const;
```

The member function **get_domain_name** shall returns the domain name associated with this phase node.

11.1.5.8 get_imp

```
uvm_phase* get_imp() const;
```

The member function **get_imp** shall return the phase implementation for this node. It shall return NULL if this phase type is not a **UVM_PHASE_LEAF_NODE**.

11.1.6 Synchronization

11.1.6.1 get_objection

```
uvm_objection* get_objection() const;
```

The member function **get_objection** shall return the object of class **uvm_objection** that gates the termination of the phase.

11.1.6.2 raise_objection

```
virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
                              int count = 1 );
```

The member function **raise_objection** shall return the object of class **uvm_objection** that gates the termination of the phase.

11.1.6.3 drop_objection

```
virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );
```

The member function **drop_objection** shall drop an objection to ending a phase. The drop is expected to be matched with an earlier raise.

11.1.6.4 sync

```
void sync( uvm_domain* target,
           uvm_phase* phase = NULL,
           uvm_phase* with_phase = NULL );
```

The member function **sync** shall synchronize two domains, fully or partially. The argument *target* is a handle of the target domain to synchronize this one to. The optional argument *phase* is the phase in this domain to synchronize with; otherwise synchronize to all. The optional argument *with_phase* is the target-domain phase to synchronize with; otherwise use *phase* in the target domain.

11.1.6.5 unsync

```
void unsync( uvm_domain* target,
             uvm_phase* phase = NULL,
             uvm_phase* with_phase = NULL );
```

The member function **unsync** shall remove the synchronization between two domains, fully or partially. The argument *target* is a handle of the target domain to remove synchronize from. The optional argument *phase* is the phase in this domain to un-synchronize with; otherwise un-synchronize to all. The optional argument *with_phase* is the target-domain phase to un-synchronize with; otherwise use *phase* in the target domain.

11.1.6.6 wait_for_state

```
void wait_for_state( uvm_phase_state state, uvm_wait_op op = UVM_EQ );
```

The member function **wait_for_state** shall wait until this phase compares with the given state and op operand. For **UVM_EQ** and **UVM_NE** operands, several **uvm_phase_states** can be supplied by their enum constants, in which case the caller shall wait until the phase state is any of **UVM_EQ** or none of **UVM_NE** the provided states.

11.1.7 Jumping

11.1.7.1 jump

```
void jump( const uvm_phase* phase );
```

The member function **jump** shall jump to a specified phase. If the destination phase is within the current phase schedule, a simple local jump takes place. If the jump-to phase is outside of the current schedule then the jump affects other schedules which share the phase.

11.1.7.2 get_jump_target

```
uvm_phase* get_jump_target() const;
```

The member function **get_jump_target** shall return the handle to the target phase of the current jump, or NULL if no jump is in progress. This member function shall only be used during the **phase_ended** callback.

11.2 uvm_domain

The class **uvm_domain** shall provide a phasing schedule node representing an independent branch of the schedule.

11.2.1 Class definition

```
namespace uvm {

  class uvm_domain : public uvm_phase
  {
```

```
   public:
    // Constructor
    explicit uvm_domain( const std::string& name );

    // Member functions
    static std::map< std::string, uvm_domain* > get_domains();
    static uvm_phase* get_uvm_schedule();
    static uvm_domain* get_common_domain();
    static void add_uvm_phases( uvm_phase* schedule );
    static uvm_domain* get_uvm_domain();

  }; // class uvm_domain

} // namespace uvm
```

11.2.2 Constructor

```
explicit uvm_domain( const std::string& name );
```

The constructor shall create a new instance of a phase domain with the *name* passed as argument.

11.2.3 Member functions

11.2.3.1 get_domains

```
static std::map< std::string, uvm_domain* > get_domains();
```

The member function **get_domains** shall provide a list of all domains in the provided domains argument.

11.2.3.2 get_uvm_schedule

```
static uvm_phase* get_uvm_schedule();
```

The member function **get_uvm_schedule** shall return the “UVM” schedule, which consists of the run-time phases that all components execute when participating in the “UVM” domain.

11.2.3.3 get_common_domain

```
static uvm_domain* get_common_domain();
```

The member function **get_common_domain** shall return the “common” domain, which consists of the common phases that all components execute in sync with each other. Phases in the “common” domain are build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report, and final.

11.2.3.4 add_uvm_phases

```
static void add_uvm_phases( uvm_phase* schedule );
```

The member function **add_uvm_phases** shall append to the given schedule the built-in UVM phases.

11.2.3.5 get_uvm_domain

```
static uvm_domain* get_uvm_domain();
```

The member function **get_uvm_domain** shall return the handle to the singleton *uvm* domain.

11.3 uvm_bottomup_phase

The class **uvm_bottomup_phase** shall provide the base class for function phases that operate bottom-up. The member function **execute** is called for each component. This is the default traversal so is included only for naming. The bottom-up phase completes when the member function **execute** has been called and returned on all applicable components in the hierarchy.

11.3.1 Class definition

```
namespace uvm {

  class uvm_bottomup_phase : public uvm_phase
  {
   public:
    // Constructor
    explicit uvm_bottomup_phase( const std::string& name );

    // Member functions
    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );

    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );

  }; // class uvm_bottomup_phase

} // namespace uvm
```

11.3.2 Constructor

```
explicit
                uvm_bottomup_phase( const std::string& name );
```

The constructor shall create a new instance of a bottom-up phase using the *name* passed as argument.

11.3.3 Member functions

11.3.3.1 traverse

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

The member function **traverse** shall traverse the component tree in bottom-up order, calling member function **execute** for each component.

11.3.3.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall execute the bottom-up phase *phase* for the component *comp*.

11.4 uvm_topdown_phase

The class **uvm_topdown_phase** shall provide the base class for function phases that operate top-down. The member function **execute** is called for each component. This is the default traversal so is included only for naming. The top-down phase completes when the member function **execute** has been called and returned on all applicable components in the hierarchy.

11.4.1 Class definition

```
namespace uvm {

  class uvm_topdown_phase : public uvm_phase
  {
   public:
    // Constructor
    explicit uvm_topdown_phase( const std::string& name );

    // Member functiond
    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );

    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );

  }; // class uvm_topdown_phase

} // namespace uvm
```

11.4.2 Constructor

```
explicit uvm_topdown_phase( const std::string& name );
```

The constructor shall create a new instance of a top-down phase using the name *name* passed as argument.

11.4.3 Member functions

11.4.3.1 traverse

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

The member function **traverse** shall traverse the component tree in top-down order, calling member function **execute** for each component.

11.4.3.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall execute the top-down phase *phase* for the component *comp*.

11.5 uvm_process_phase° (uvm_task_phase†)

The class **uvm_process_phase**° shall provide the base class for all process-oriented phases. It is responsible to create spawned processes as part of the execution of the callback **uvm_phase::exec_process** for each component in the hierarchy. The completion of the execution of this callback does not imply, nor is it required for, the end of phase. Once the phase completes, any remaining spawned processes caused by executing **uvm_phase::exec_process** are forcibly and immediately killed. By default, the way for a process phase to extend over time is if there is at least one component that raises an objection.

11.5.1 Class definition

```
namespace uvm {

  class uvm_process_phase° : public uvm_phase
  {
   public:
    // Constructor
    explicit uvm_process_phase°( const std::string& name );

    // Member functions
    virtual void traverse( uvm_component* comp,
                           uvm_phase* phase,
                           uvm_phase_state state );

    virtual void execute( uvm_component* comp,
                          uvm_phase* phase );

  }; // class uvm_process_phase

} // namespace uvm
```

11.5.2 Member functions

11.5.2.1 traverse

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

The member function **traverse** shall traverse the component tree in bottom-up order, calling member function **execute** for each component.

NOTE—The actual order for process-based phases does not really matter, as each component process is executed in a separate process whose starting order is not deterministic.

11.5.2.2 execute

```
virtual void execute( uvm_component* comp,
                      uvm_phase* phase );
```

The member function **execute** shall spawn a process of phase *phase* for the component *comp*.

11.6 uvm_objection

The class **uvm_objection** shall provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP.

11.6.1 Class definition

```
namespace uvm {

  class uvm_objection : public uvm_object
  {
   public:
    // Constructors
    uvm_objection();
    uvm_objection( const std::string& name );

    // Group: Objection control
    virtual void clear( uvm_object* obj = NULL );
    bool trace_mode( int mode = -1 );
```

```
    virtual void raise_objection( uvm_object* obj,
                                  const std::string& description = "",
                                  int count = 1 );

    virtual void drop_objection( uvm_object* obj,
                                 const std::string& description = "",
                                 int count = 1 );

    void set_drain_time( uvm_object* obj = NULL,
                         const sc_core::sc_time& drain = sc_core::SC_ZERO_TIME );

    // Group: Callback hooks
    virtual void raised( uvm_object* obj,
                         uvm_object* source_obj,
                         const std::string& description,
                         int count );

    virtual void dropped( uvm_object* obj,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );

    virtual void all_dropped( uvm_object* obj,
                              uvm_object* source_obj,
                              const std::string& description,
                              int count );

    // Group: Objection status
    void get_objectors( std::vector<uvm_object*>& objlist ) const;

    void wait_for( uvm_objection_event objt_event,
                   uvm_object* obj = NULL );

    int get_objection_count( uvm_object* obj = NULL ) const;
    int get_objection_total( uvm_object* obj = NULL ) const;

    const sc_core::sc_time get_drain_time( uvm_object* obj = NULL ) const;

    void display_objections( uvm_object* obj = NULL,
                             bool show_header = true ) const;

  }; // class uvm_objection

} // namespace uvm
```

11.6.2 Constructors

```
uvm_objection();
uvm_objection( const std::string& name );
```

The constructor shall create a new objection instance with name *name*, if specified.

11.6.3 Objection control

11.6.3.1 clear

```
virtual void clear( uvm_object* obj = NULL );
```

The member function **clear** shall clear the objection state immediately. All counts are cleared and any processes that called **wait_for**(**UVM_ALL_DROPPED**, **uvm_top**) are released An application should pass `this` to the *obj* argument for record keeping. Any configured drain times are not affected.

11.6.3.2 trace_mode

```
bool trace_mode( int mode = -1 );
```

The member function **trace_mode** shall set or get the trace mode for the objection object. If no argument is specified (or an argument other than 0 or 1) the current trace mode is unaffected. A trace_mode of 0 turns tracing off. A trace mode of 1 turns tracing on. The return value is the mode prior to being reset.

11.6.3.3 raise_objection

```
virtual void raise_objection( uvm_object* obj,
                              const std::string& description = "",
                              int count = 1 );
```

The member function **raise_objection** shall increase the number of objections for the source object by *count*, which defaults to 1. The object is usually the current (`this`) handle of the caller. If an object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen.

Raising an objection shall cause the following.

- The source and total objection counts for object are increased by *count*.
- The member function **raised** is called, which calls the member function **uvm_component**::**raised** for all of the components up the hierarchy.

The description is a string that marks a specific objection and is used in tracing or debug.

11.6.3.4 drop_objection

```
virtual void drop_objection( uvm_object* obj,
                             const std::string& description = "",
                             int count = 1 );
```

The member function **drop_objection** shall decrease the number of objections for the source object by *count*, which defaults to 1. The object is usually the current handle (`this`) of the caller. If object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen.

Dropping an objection shall cause the following:

- The source and total objection counts for object are decreased by *count*. It shall be an error to drop the objection count for object below zero.
- The member function **dropped** is called, which calls the member function **uvm_component**::**dropped** for all of the components up the hierarchy.

If the total objection count has not reached zero for the object, then the drop is propagated up the object hierarchy as with **raise_objection**. Then, each object in the hierarchy shall update its 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, propagation up the hierarchy is deferred until a configurable drain-time has passed and the **uvm_component**::**all_dropped** callback for the current hierarchy level has returned.

For each instance up the hierarchy from the source caller, a process is forked in a non-blocking fashion, allowing the **drop** call to return. The forked process then does the following:

- If a drain time was set for the given object, the process waits for that amount of time.
- The objection's virtual member function **all_dropped** is called, which calls the member function **uvm_component**::**all_dropped** (if object is a component).
- The process then waits for the **all_dropped** callback to complete.
- 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**, except as described below.

If a new objection for this object or any of its descendents is raised during the drain time or during execution of the **all_dropped** callback at any point, the hierarchical chain described above is terminated and the **dropped** callback does not go up the hierarchy. The raised objection shall propagate 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, 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 drain-time set and no registered callbacks, the forked process can be skipped and propagation proceeds immediately to the parent as described.

11.6.3.5 set_drain_time

```
void set_drain_time( uvm_object* obj = NULL,
                     const sc_core::sc_time& drain = sc_core::SC_ZERO_TIME );
```

The member function **set_drain_time** shall set the drain time on the given object to drain. The drain time is the amount of time to wait once all objections have been dropped before calling the **all_dropped** callback and propagating the objection to the parent. If a new objection for this object or any of its descendents is raised during the drain time or during execution of the **all_dropped** callbacks, the drain_time/all_dropped execution is terminated.

11.6.4 Callback hooks

11.6.4.1 raised

```
virtual void raised( uvm_object* obj,
                     uvm_object* source_obj,
                     const std::string& description,
                     int count );
```

The member function **raised** shall be called when a **raise_objection** has reached *obj*. The default implementation shall call **uvm_component**::**raised** (see Section 7.1.7.1).

11.6.4.2 dropped

```
virtual void dropped( uvm_object* obj,
                      uvm_object* source_obj,
                      const std::string& description,
                      int count );
```

The member function **dropped** shall be called when a **drop_objection** has reached *obj*. The default implementation shall call **uvm_component**::**dropped** (see Section 7.1.7.2).

11.6.4.3 all_dropped

```
virtual void all_dropped( uvm_object* obj,
                          uvm_object* source_obj,
                          const std::string& description,
                          int count );
```

The member function **all_dropped** shall be called when a **drop_objection** has reached *obj*, and the total count for *obj* goes to zero. This callback is executed after the drain time associated with *obj*. The default implementation shall call **uvm_component**::**all_dropped** (see Section 7.1.7.3).

11.6.5 Objections status

11.6.5.1 get_objectors

```
void get_objectors( std::vector<uvm_object*>& objlist ) const;
```

The member function **get_objectors** shall return the current list of objecting objects (objects that raised an objection but have not dropped it).

11.6.5.2 wait_for

```
void wait_for( uvm_objection_event objt_event,
               uvm_object* obj = NULL );
```

The member function **wait_for** shall wait for the raised, dropped, or all_dropped event to occur in the given object *obj*. The member function returns after all corresponding callbacks for that event have been executed.

11.6.5.3 get_objection_count

```
int get_objection_count( uvm_object* obj = NULL ) const;
```

The member function **get_objection_count** shall return the current number of objections raised by the given object *obj*.

11.6.5.4 get_objection_total

```
int get_objection_total( uvm_object* obj = NULL ) const;
```

The member function **get_objection_total** shall return the current number of objections raised by the given object *obj* and all descendants.

11.6.5.5 get_drain_time

```
const sc_core::sc_time get_drain_time( uvm_object* obj = NULL ) const;
```

The member function **get_drain_time** shall return the current drain time set for the given object *obj*. The default drain time shall be set to **sc_core**::**SC_ZERO_TIME**.

11.6.5.6 display_objections

```
void display_objections( uvm_object* obj = NULL,
                         bool show_header = true ) const;
```

The member function **display_objections** shall display objection information about the given object *obj*. If object is not specified or NULL, the implicit top-level component, **uvm_root**, is chosen. The argument *show_header* allows control of whether a header is output.

11.7 uvm_callback

The class **uvm_callback** shall provide 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, he defines one or more virtual member functions, called a callback interface, that represent the hooks available for user override.

The member functions intended for optional override should not be declared pure virtual. Usually, all the callback member functions are defined with empty implementations so users have the option of overriding any or all of them. The prototypes for each hook member function are completely application specific with no restrictions.

11.7.1 Class definition

```
namespace uvm {

  class uvm_callback : public uvm_object
  {
   public:
    // Constructor
    uvm_callback( const std::string& name = "uvm_callback" );

    // Member functions
    bool callback_mode( int on = -1 );
    bool is_enabled();
    virtual const std::string get_type_name() const;

  }; // class uvm_callback

} // namespace uvm
```

11.7.2 Constructor

```
uvm_callback( const std::string& name = "uvm_callback" );
```

The constructor shall create a new object of type **uvm_callback**, giving it an optional name.

11.7.3 Member functions

11.7.3.1 callback_mode

```
bool callback_mode( int on = -1 );
```

The member function **callback_mode** shall enable or disable callbacks. If argument *on* is set 1, callbacks are enabled. If argument *on* is set 0, callbacks are disabled.

11.7.3.2 is_enabled

```
bool is_enabled();
```

The member function **is_enabled** shall return 1 if the callback is enabled, otherwise it shall return 0.

11.7.3.3 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the type name of this callback object.

11.8 uvm_callback_iter

The class **uvm_callback_iter** is an iterator class for iterating over callback queues of a specific callback type.

11.8.1 Class definition

```
namespace uvm {

  template < typename T = uvm_object, typename CB = uvm_callback>
  class uvm_callback_iter
  {
   public:
    // Constructor
    uvm_callback_iter( T* obj );

    // Member functions
    CB* first();
    CB* last();
    CB* next();
    CB* prev();
    CB* get_cb();

  }; // class uvm_callback

} // namespace uvm
```

11.8.2 Template parameter T

The template parameter T specifies the base object type with which the callback objects CB are registered. This object shall be a derivative of class **uvm_object**.

11.8.3 Template parameter CB

The template parameter T specifies the base callback type that is managed by this callback class. The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm_callback**.

11.8.4 Constructor

```
uvm_callback_iter( T* obj );
```

The constructor shall create a new callback iterator object. It is required that the object context be provided.

11.8.5 Member functions

11.8.5.1 first

```
CB* first();
```

The member function **first** shall return 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, then NULL is returned.

11.8.5.2 last

```
CB* last();
```

The member function **last** shall return 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, then NULL is returned.

11.8.5.3 next

```
CB* next();
```

The member function **next** shall return 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, then NULL is returned.

11.8.5.4 prev

```
CB* prev();
```

The member function **prev** shall return 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, then NULL is returned.

11.8.5.5 get_cb

```
CB* get_cb();
```

The member function **get_cb** shall return the last callback accessed via the call **first** or **next**.

11.9 uvm_callbacks

The class **uvm_callbacks** shall provide 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 are associated together using the macro **UVM_REGISTER_CB** to define a valid pairing; valid pairings are checked when a user attempts to add a callback to an object (see Section 13.4.2).

To provide the most flexibility for end-user customization and reuse, it is recommended that the component developer also define a corresponding set of virtual method hooks in the component itself. This affords users the ability to customize via inheritance/factory overrides as well as callback object registration. The implementation of each virtual method would provide the default traversal algorithm for the particular callback being called. Being virtual, an application can define subtypes that override the default algorithm, perform tasks before and/or after calling the base class to execute any registered callbacks, or to not call the base implementation, effectively disabling that particular hook.

11.9.1 Class definition

```
namespace uvm {

  template <typename T = uvm_object, typename CB = uvm_callback>
  class uvm_callbacks : public uvm_typed_callbacks<T>
  {
   public:
    // Constructor
    uvm_callbacks();

    // Group: Add/delete inteface
```

```
    static void add( T* obj, uvm_callback* cb, uvm_apprepend ordering = UVM_APPEND );

    static void add_by_name( const std::string& name,
                             uvm_callback* cb,
                             uvm_component* root,
                             uvm_apprepend ordering = UVM_APPEND );

    static void do_delete°( T* obj, uvm_callback* cb );

    static void delete_by_name( const std::string& name,
                                uvm_callback* cb,
                                uvm_component* root );

    // Group: Iterator Interface
    static CB* get_first( int& itr, T* obj );
    static CB* get_last( int& itr, T* obj );
    static CB* get_next( int& itr, T* obj );
    static CB* get_prev( int& itr, T* obj );

    // Group: Debug
    static void display( T* obj = NULL );

  }; // class uvm_callbacks

} // namespace uvm
```

11.9.2 Template parameter T

The template parameter T specifies the base object type with which the callback objects CB are registered. This object shall be a derivative of class **uvm_object**.

11.9.3 Template parameter CB

The template parameter CB specifies the base callback type that is managed by this callback class.The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm_callback**.

11.9.4 Constructor

```
uvm_callbacks();
```

The constructor shall create a new object of type **uvm_callbacks** <T, CB>.

11.9.5 Add/delete interface

11.9.5.1 add

```
static void add( T* obj, uvm_callback* cb, uvm_apprepend ordering = UVM_APPEND );
```

The member function **add** shall register the given callback object, *cb*, with the given handle *obj*. The object handle can be NULL, which allows registration of callbacks without an object context. If ordering is **UVM_APPEND** (default), the callback shall be executed after previously added callbacks, else the callback shall be executed ahead of previously added callbacks. The argument *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.

11.9.5.2 add_by_name

```
static void add_by_name( const std::string& name,
                         uvm_callback* cb,
                         uvm_component* root,
                         uvm_apprepend ordering = UVM_APPEND );
```

The member function **add_by_name** shall register the given callback object, *cb*, with one or more components of type **uvm_component**. The components shall already exist and shall be type T or a derivative. As with add the CB parameter is optional. Argument *root* specifies the location in the component hierarchy to start the search for *name*. See **uvm_root**::**find_all** (Section 4.3.3.2) for more details on searching by name.

11.9.5.3 do_delete°(delete†)

```
static void do_delete° ( T* obj, uvm_callback* cb );
```

The member function **do_delete**° shall delete the given callback object, *cb*, from the queue associated with the given object handle *obj*. The object handle can be NULL, which allows de-registration of callbacks without an object context. The argument *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 is issued.

11.9.5.4 delete_by_name

```
static void delete_by_name( const std::string& name,
                            uvm_callback* cb,
                            uvm_component* root );
```

The member function **delete_by_name** shall remove the given callback object, *cb*, associated with one or more **uvm_component** callback queues. Argument *root* specifies the location in the component hierarchy to start the search for name. See **uvm_root**::**find_all** (Section 4.3.3.2) for more details on searching by name.

11.9.6 Iterator interfaces

This set of member functions shall provide an iterator interface for callback queues. A facade class, **uvm_callback_iter** is also available, and is the generally preferred way to iterate over callback queues. (See Section 11.8).

11.9.6.1 get_first

```
static CB* get_first( int& itr, T* obj );
```

The member function **get_first** shall return the first enabled callback of type CB which resides in the queue for object *obj*. If object *obj* is NULL, then the typewide queue for T is searched. Argument *itr* is the iterator; it is being updated with a value that can be supplied to **get_next** to get the next callback object. If the queue is empty, then NULL is returned. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.2 get_last

```
static CB* get_last( int& itr, T* obj );
```

The member function **get_last** shall return the last enabled callback of type CB which resides in the queue for object *obj*. If object *obj* is NULL, then the typewide queue for T is searched. Argument *itr* is the iterator; it is being updated with a value that can be supplied to **get_prev** to get the previous callback object. If the queue is empty then NULL is returned. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.3 get_next

```
static CB* get_next( int& itr, T* obj );
```

The member function **get_next** shall return the next enabled callback of type CB which resides in the queue for object *obj*, using iterator *itr* as the starting point. If object *obj* is NULL, then the typewide queue for T is searched.

The iterator is being updated with a value that can be supplied to **get_next** to get the next callback object. If no more callbacks exist in the queue, then NULL is returned. The member function **get_next** shall continue to return NULL in this case until member function **get_first** or **get_last** has been used to reset the iterator. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.6.4 get_prev

```
static CB* get_prev( int& itr, T* obj );
```

The member function **get_prev** shall return the previous enabled callback of type CB which resides in the queue for object *obj*, using iterator *itr* as the starting point. If object *obj* is NULL, then the typewide queue for T is searched. The iterator is being updated with a value that can be supplied to member function **get_prev** to get the previous callback object. If no more callbacks exist in the queue, then NULL is returned. The member function **get_prev** shall continue to return NULL in this case until member function **get_first** or **get_last** has been used to reset the iterator. The iterator class **uvm_callback_iter** may be used as an alternative, simplified, iterator interface.

11.9.7 Debug

11.9.7.1 display

```
static void display( T* obj = NULL );
```

The member function **display** shall display callback information for object *obj*. If object *obj* is NULL, then it displays callback information for all objects of type T, including typewide callbacks.

12. Reporting classes

The UVM-SystemC reporting classes provide an additional 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. It supports a component-level reporting mechanism by setting the severity level on a per-instance basis. In addition, some convenience macros are available for the reporting of information, warnings, errors, or fatal errors.

SystemC has already an extensive and highly configurable message-reporting mechanism using the **sc_core**::**sc_report_handler** class and **sc_core**::**sc_report** objects. An application may also use this native SystemC global-level reporting mechanism where appropriate.

The following reporting classes are defined:

- **uvm_report_message**: The class which provides the fields that are common to all messages.
- **uvm_report_object**: The base class which provides the interface to the UVM reporting mechanism.
- **uvm_report_handler**: The class which acting as implementation for the member functions defined in the class **uvm_report_object**.
- **uvm_report_server**: The class acting as global server that processes all of the reports generated by the class **uvm_report_handler**.
- **uvm_report_catcher**: The class which captures and counts all reports issued by the class **uvm_report_server**.

The primary interface to the UVM reporting facility is the class **uvm_report_object** from which class **uvm_component** is derived. The class **uvm_report_object** delegates most tasks to its internal **uvm_report_handler**. If the report handler determines the report is not filtered based the configured verbosity setting, it sends the report to the central **uvm_report_server** for formatting and processing.

12.1 uvm_report_message

The class **uvm_report_message** shall be used to compose a UVM object message. It provides the fields that are common to all messages. It also has a message element container and provides the APIs necessary to add integral types, strings and uvm_objects to the container. The report message object can be initialized with the common fields, and passes through the whole reporting system (i.e. report object, report handler, report server, report catcher, etc) 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.

12.1.1 Class definition

```
namespace uvm {

  class uvm_report_message : public uvm_object
  {
   public:

    uvm_report_message( const std::string& name = "uvm_report_message" );

    // Group: Infrastructure References
    virtual void do_print( const uvm_printer& printer ) const;
    virtual uvm_report_object* get_report_object() const;
    virtual void set_report_object( uvm_report_object* ro );
    virtual uvm_report_handler* get_report_handler() const;
    virtual void set_report_handler( uvm_report_handler* rh );
    virtual uvm_report_server* get_report_server() const;
    virtual void set_report_server( uvm_report_server* rs );

    // Group: Message Fields
    virtual uvm_severity get_severity() const;
```

```
    virtual void set_severity( uvm_severity sev );
    virtual const std::string get_id() const;
    virtual void set_id( const std::string& id );
    virtual const std::string get_message() const;
    virtual void set_message( const std::string& msg );
    virtual int get_verbosity() const;
    virtual void set_verbosity( int ver );
    virtual const std::string get_filename() const;
    virtual void set_filename( const std::string& fname );
    virtual int get_line() const;
    virtual void set_line( int ln );
    virtual const std::string get_context() const;
    virtual void set_context( const std::string& cn );
    virtual uvm_action get_action() const;
    virtual void set_action( uvm_action act );
    virtual UVM_FILE get_file() const;
    virtual void set_file( UVM_FILE fl );
    virtual uvm_report_message_element_container* get_element_container() const;

    virtual void set_report_message( uvm_severity severity,
                                     const std::string& id,
                                     const std::string& message,
                                     int verbosity,
                                     const std::string& filename,
                                     int line,
                                     const std::string& context_name );

    // Group: Message Element APIs

    virtual void add_int( const std::string& name,
                          uvm_bitstream_t value,
                          int size,
                          uvm_radix_enum radix,
                          uvm_action action = (UVM_LOG | UVM_RM_RECORD) );

    virtual void add_string( const std::string& name,
                             const std::string& value,
                             uvm_action action = (UVM_LOG | UVM_RM_RECORD) );

    virtual void add_object( const std::string& name,
                             uvm_object* obj,
                             uvm_action action = (UVM_LOG | UVM_RM_RECORD) );

  }; // class uvm_report_message

} // namespace uvm
```

12.1.2 Constructor

```
uvm_report_message( const std::string& name = "uvm_report_message" );
```

The constructor shall create a new report message with the given *name*.

12.1.3 Infrastructure references

12.1.3.1 do_print

```
virtual void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall provide UVM printer formatted output of the message.

12.1.3.2 get_report_object

```
virtual uvm_report_object* get_report_object() const;
```

The member function **get_report_object** shall return the **uvm_report_object** that originated the message.

12.1.3.3 set_report_object

```
virtual void set_report_object( uvm_report_object* ro );
```

The member function **set_report_object** shall define the **uvm_report_object** for the message.

12.1.3.4 get_report_handler

```
virtual uvm_report_handler* get_report_handler() const;
```

The member function **get_report_handler** shall return the **uvm_report_handler**.

12.1.3.5 set_report_handler

```
virtual void set_report_handler( uvm_report_handler* rh );
```

The member function **set_report_handler** shall define the **uvm_report_handler**.

12.1.3.6 get_report_server

```
virtual uvm_report_server* get_report_server() const;
```

The member function **get_report_server** shall return the **uvm_report_server** that is responsible for servicing the message's actions.

12.1.3.7 set_report_server

```
virtual void set_report_server( uvm_report_server* rs );
```

The member function **set_report_server** shall define the **uvm_report_server** that is responsible for servicing the message's actions.

12.1.4 Message fields

12.1.4.1 get_severity

```
virtual uvm_severity get_severity() const;
```

The member function **get_severity** shall return the severity of the message (**UVM_INFO**, **UVM_WARNING**, **UVM_ERROR** or **UVM_FATAL**). The value of this field is determined via the API used (e.g. use of macro's **UVM_INFO**, **UVM_WARING**, etc.) and is populated for the application.

12.1.4.2 set_severity

```
virtual void set_severity( uvm_severity sev );
```

The member function **set_severity** shall define the severity of the message (**UVM_INFO**, **UVM_WARNING**, **UVM_ERROR** or **UVM_FATAL**).

12.1.4.3 get_id

```
virtual const std::string get_id() const;
```

The member function **get_id** shall define the id of the message.

12.1.4.4 set_id

```
virtual void set_id( const std::string& id );
```

The member function **set_id** shall return the id of the message.

NOTE—It is recommended that an application follows a consistent convention. Settings in the **uvm_report_handler** allow various messaging controls based on this field. (See Section 12.3).

12.1.4.5 get_message

```
virtual const std::string get_message() const;
```

The member function **get_message** shall return the message content as string.

12.1.4.6 set_message

```
virtual void set_message( const std::string& msg );
```

The member function **set_message** shall set the message content given as string argument.

12.1.4.7 get_verbosity

```
virtual int get_verbosity() const;
```

The member function **get_verbosity** shall return the message threshold value. This value is compared against settings in the **uvm_report_handler** to determine whether this message should be executed.

12.1.4.8 set_verbosity

```
virtual void set_verbosity( int ver );
```

The member function **set_verbosity** shall define the message threshold value.

12.1.4.9 get_filename

```
virtual const std::string get_filename() const;
```

The member function **get_filename** shall return the filename from which the message originates. This value is automatically populated by the messaging macros.

12.1.4.10 set_filename

```
virtual void set_filename( const std::string& fname );
```

The member function **set_filename** shall define the filename in which the message is created.

12.1.4.11 get_line

```
virtual int get_line() const;
```

The member function **get_line** shall return the line number in the file from which the message originates. This value is automatically populate by the messaging macros.

12.1.4.12 set_line

```
virtual void set_line(int ln);
```

The member function **set_line** shall define the line number at which the message is created.

12.1.4.13 get_context

```
virtual const std::string get_context() const;
```

The member function **get_context** shall return the context of the message.

12.1.4.14 set_context

```
virtual void set_context( const std::string& cn );
```

The member function **set_context** shall specify the optional user-supplied string that is meant to convey the context of the message.

12.1.4.15 get_action

```
virtual uvm_action get_action() const;
```

The member function **get_action** shall return the action(s) that the **uvm_report_server** should perform for this message.

12.1.4.16 set_action

```
virtual void set_action( uvm_action act );
```

The member function **set_action** shall define the action(s) that the **uvm_report_server** should perform for this message.

12.1.4.17 get_file

```
virtual UVM_FILE get_file() const;
```

The member function **get_file** shall return the file handle to the file where the message has been written to, when the message's action is **UVM_LOG**.

12.1.4.18 set_file

```
virtual void set_file( UVM_FILE fl );
```

The member function **set_file** shall define the file handle to the file where the message is to be written to, when the message's action is **UVM_LOG**.

12.1.4.19 get_element_container

```
virtual uvm_report_message_element_container* get_element_container() const;
```

The member function **get_element_container** shall return the element container of the message.

12.1.4.20 set_report_message

```
virtual void set_report_message( uvm_severity severity,
                                 const std::string& id,
                                 const std::string& message,
                                 int verbosity,
                                 const std::string& filename,
                                 int line,
                                 const std::string& context_name );
```

The member function **set_report_message** shall set all the common fields of the report message.

12.1.5 Message element APIs

12.1.5.1 add_int

```
virtual void add_int( const std::string& name,
                      uvm_bitstream_t value,
                      int size,
                      uvm_radix_enum radix,
                      uvm_action action = (UVM_LOG | UVM_RM_RECORD) );
```

The member function **add_int** shall add an integral type of the name *name* and value *value* to the message. The required size field indicates the size of *value*. The required *radix* field determines how to display and record the field. The optional print/record bit is to specify whether the element is printed/recorded.

12.1.5.2 add_string

```
virtual void add_string( const std::string& name,
                         const std::string& value,
                         uvm_action action = (UVM_LOG | UVM_RM_RECORD) );
```

The member function **add_string** shall add a string of the name *name* and value *value* to the message. The optional print/record bit is to specify whether the element is printed/recorded.

12.1.5.3 add_object

```
virtual void add_object( const std::string& name,
                         uvm_object* obj,
                         uvm_action action = (UVM_LOG | UVM_RM_RECORD) );
```

The member function **add_object** shall add a **uvm_object** of the name *name* and reference *obj* to the message. The optional print/record bit is to specify whether the element is printed/recorded.

12.2 uvm_report_object

The class **uvm_report_object** shall provide the primary interface to the UVM reporting facility. Through this interface, components issue the various messages that occur during simulation. An application can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment. Defaults are applied where there is no explicit configuration.

A report consists of an id string, severity, verbosity level, and the textual message itself. They may optionally include the filename and line number from which the message came. If the verbosity level of a report is greater than the configured maximum verbosity level 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.

— *Actions* can be set for (in increasing priority) severity, id, and (severity, id) pair. They include output to the screen or log file (**UVM_DISPLAY** or **UVM_LOG** respectively), whether the message counters should be incremented (**UVM_COUNT**), whether a simulation should be finished (**UVM_EXIT**) or stopped (**UVM_STOP**). The action can also specify if a specific callback should be called as soon as the reporting occurs (**UVM_CALL_HOOK**). Actions are of type **uvm_action** and can take the value **UVM_NO_ACTION**, or it can be a bitwise OR of any combination of **UVM_DISPLAY**, **UVM_LOG**, **UVM_COUNT**, **UVM_STOP**, **UVM_EXIT**, and **UVM_CALL_HOOK.** (See Section 17.4.1).

— *Default actions*: The following provides the default actions assigned to each severity. These can be overridden by any of the member function **set_report_id_action**.

Severity	**Default action(s)**
UVM_INFO	**UVM_DISPLAY**
UVM_WARNING	**UVM_DISPLAY**, **UVM_COUNT**
UVM_ERROR	**UVM_DISPLAY**, **UVM_COUNT**
UVM_FATAL	**UVM_DISPLAY**, **UVM_COUNT**, **UVM_EXIT**

— *File descriptors*: These can be set by (in increasing priority) default, severity level, an id, or (severity, id) pair. File descriptors are of type **UVM_FILE**. They may refer to more than one file. It is the application's responsibility to open and close the files.

— *Default file handle*: The default file handle is 0, which means that reports are not sent to a file even if a **UVM_LOG** attribute is set in the action associated with the report. This can be overridden by the member function **set_report_default_file**, **set_report_severity_file**, **set_report_id_file** or **set_report_severity_id_file**. As soon as the file descriptor is set and the action **UVM_LOG** is set, the report is sent to its associated file descriptor.

12.2.1 Class definition

```
namespace uvm {

  class uvm_report_object : public uvm_object
  {
   public:
    // Constructors
    uvm_report_object();
    explicit uvm_report_object( const std::string& name );

    // Group: Reporting
    bool uvm_report_enabled( int verbosity,
                             uvm_severity_type severity = UVM_INFO,
                             const std::string& id = "" );

    virtual void uvm_report_info( const std::string& id,
                                  const std::string& message,
                                  int verbosity = UVM_MEDIUM,
                                  const std::string& filename = "",
                                  int line = 0 ) const;
```

```
    virtual void uvm_report_warning( const std::string& id,
                                     const std::string& message,
                                     int verbosity = UVM_MEDIUM,
                                     const std::string& filename = "",
                                     int line = 0 ) const;

    virtual void uvm_report_error( const std::string& id,
                                   const std::string& message,
                                   int verbosity = UVM_LOW,
                                   const std::string& filename = "",
                                   int line = 0 ) const;

    virtual void uvm_report_fatal( const std::string& id,
                                   const std::string& message,
                                   int verbosity = UVM_NONE,
                                   const std::string& filename = "",
                                   int line = 0 ) const;

    // Group: Verbosilty Configuration
    int get_report_verbosity_level( uvm_severity_type severity = UVM_INFO,
                                    const std::string& id = "" ) const;
    void set_report_verbosity_level( int verbosity_level );
    void set_report_id_verbosity( const std::string& id, int verbosity );
    void set_report_severity_id_verbosity( uvm_severity severity,
                                           const std::string& id,
                                           int verbosity );

    // Action configuration
    int get_report_action( uvm_severity severity,
                           const std::string& id ) const;
    void set_report_severity_action( uvm_severity severity,
                                     uvm_action action );
    void set_report_id_action( const std::string& id,
                               uvm_action action );
    void set_report_severity_id_action( uvm_severity severity,
                                        const std::string& id,
                                        uvm_action action );

    // File configuration
    UVM_FILE get_report_file_handle( uvm_severity severity,
                                     const std::string& id ) const;
    void set_report_default_file( UVM_FILE file );
    void set_report_id_file( const std::string& id, UVM_FILE file );
    void set_report_severity_file( uvm_severity severity, UVM_FILE file );
    void set_report_severity_id_file( uvm_severity severity,
                                      const std::string& id,
                                      UVM_FILE file);

    // Override Configuration
    void set_report_severity_override( uvm_severity cur_severity,
                                       uvm_severity new_severity );

    void set_report_severity_id_override( uvm_severity cur_severity,
                                          const std::string& id,
                                          uvm_severity new_severity );

    // Group: Report Handler Configuration
    void set_report_handler( uvm_report_handler* handler );
    uvm_report_handler* get_report_handler() const;
    void reset_report_handler();

  }; // class uvm_report_object

} // namespace uvm
```

12.2.2 Constructors

```
uvm_report_object();
explicit uvm_report_object( const std::string& name );
```

The constructors shall create a new report object with the given name. This member function shall also create a new **uvm_report_handler** object to which most tasks are delegated.

12.2.3 Reporting

The member functions **uvm_report_info**, **uvm_report_warning** and **uvm_report_fatal** are the primary reporting methods in UVM. They ensure a consistent output and central control over where output is directed and any actions that result. All reporting member functions have the same arguments, although each has a different default verbosity:

- *id*: a unique id of type std::string for the report or report group that can be used for identification and therefore targeted filtering. An application can configure an individual report's actions and output file(s) using this id.
- *message*: the message body, preformatted to a single string of type std::string.
- *verbosity*: the verbosity of the message, indicating its relative importance. The verbosity shall be specified as an enumeration of type **uvm_verbosity**. If the equivalent verbosity value is less than or equal to the effective verbosity level (see Section 12.2.4.2), then the report is issued, subject to the configured action and file descriptor settings. 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**, then the verbosity is taken into account. The predefined **uvm_verbosity** values are **UVM_NONE**, **UVM_LOW**, **UVM_MEDIUM**, **UVM_HIGH**, and **UVM_FULL**.
- *filename* (optional): The file from which the report was issued. An application can use the predefined macros __FILE__ and __LINE__. If specified, it is displayed in the output.
- *line* (optional): The location from which the report was issued. An application can use the predefined macro __LINE__. If specified, it is displayed in the output.

12.2.3.1 uvm_report_enabled

```
bool uvm_report_enabled( int verbosity,
                         uvm_severity_type severity = UVM_INFO,
                         const std::string& id = "" );
```

The member function **uvm_report_enabled** shall return true if the configured verbosity for this severity/id is greater than or equal to the given argument *verbosity*; otherwise it shall return false.

12.2.3.2 uvm_report_info

```
virtual void uvm_report_info( const std::string& id,
                              const std::string& message,
                              int verbosity = UVM_MEDIUM,
                              const std::string& filename = "",
                              int line = 0 ) const;
```

The member function **uvm_report_info** shall issue an info message using the current messages report object.

12.2.3.3 uvm_report_warning

```
virtual void uvm_report_warning( const std::string& id,
                                 const std::string& message,
                                 int verbosity = UVM_MEDIUM,
                                 const std::string& filename = "",
                                 int line = 0 ) const;
```

The member function **uvm_report_warning** shall issue a warning message using the current messages report object.

12.2.3.4 uvm_report_error

```
virtual void uvm_report_error( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_LOW,
                               const std::string& filename = "",
                               int line = 0 ) const;
```

The member function **uvm_report_error** shall issue an error message using the current messages report object.

12.2.3.5 uvm_report_fatal

```
virtual void uvm_report_fatal( const std::string& id,
                               const std::string& message,
                               int verbosity = UVM_NONE,
                               const std::string& filename = "",
                               int line = 0 ) const;
```

The member function **uvm_report_fatal** shall issue a fatal message using the current messages report object.

12.2.4 Verbosity configuration

12.2.4.1 get_report_verbosity_level

```
int get_report_verbosity_level( uvm_severity_type severity = UVM_INFO,
                                const std::string& id = "" ) const;
```

The member function **get_report_verbosity_level** shall get the verbosity level in effect for this object. Reports issued with verbosity greater than this shall be filtered out. The severity and tag arguments check if the verbosity level has been modified for specific severity/tag combinations.

12.2.4.2 set_report_verbosity_level

```
void set_report_verbosity_level( int verbosity_level );
```

The member function **set_report_verbosity_level** shall set the maximum verbosity level for reports for this component. Any report from this component whose verbosity exceeds this maximum is ignored.

12.2.4.3 set_report_id_verbosity

```
void set_report_id_verbosity( const std::string& id, int verbosity );
```

The member function **set_report_id_verbosity** shall associate the specified verbosity with reports of the given id. A verbosity associated with a particular id takes precedence over a verbosity associated with a severity.

12.2.4.4 set_report_severity_id_verbosity

```
void set_report_severity_id_verbosity( uvm_severity severity,
                                       const std::string& id,
                                       int verbosity );
```

The member function **set_report_severity_id_verbosity** shall associate the specified verbosity with reports of the given severity-id pair. A verbosity associated with a particular severity-id pair takes precedence over a verbosity associated with id, which take precedence over a verbosity associated with a severity.

12.2.5 Action configuration

12.2.5.1 get_report_action

```
int get_report_action( uvm_severity severity,
                       const std::string& id ) const;
```

The member function **get_report_action** shall get the action associated with reports having the given *severity* and *id*.

12.2.5.2 set_report_severity_action

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

The member function **set_report_severity_action** shall associate the specified action or actions with the given severity. An action associated with a particular severity-id pair or id, using the member functions **set_report_severity_id_action** or **set_report_id_action** respectively, shall take precedence over the association set by this member function.

12.2.5.3 set_report_id_action

```
void set_report_id_action( const std::string& id,
                           uvm_action action );
```

The member function **set_report_id_action** shall associate the specified action or actions with the given id. An action associated with a particular severity-id pair, using the member functions **set_report_severity_id_action**, shall take precedence over the association set by this member function.

12.2.5.4 set_report_severity_id_action

```
void set_report_severity_id_action( uvm_severity severity,
                                    const std::string& id,
                                    uvm_action action );
```

The member function **set_report_severity_id_action** shall associate the specified action or actions with the given id. An action associated with a particular severity-id pair shall take precedence over an action associated with id, which takes precedence over an action associated with a severity.

12.2.6 File configuration

12.2.6.1 get_report_file_handle

```
UVM_FILE get_report_file_handle( uvm_severity severity,
                                 const std::string& id ) const;
```

The member function **get_report_file_handle** shall get the file descriptor associated with reports having the given *severity* and *id*.

12.2.6.2 set_report_default_file

```
void set_report_default_file( UVM_FILE file );
```

The member function **set_report_default_file** shall configure the report handler to direct some or all of its output to the default *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id pair shall take precedence over a FILE associated with id, which shall take precedence over a file associated with a severity, which shall takes precedence over the association set by this member function.

12.2.6.3 set_report_id_file

```
void set_report_id_file( const std::string& id, UVM_FILE file );
```

The member function **set_report_id_file** shall configure the report handler to direct reports of the given *id* to the *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id shall take precedence over the association set by this member function.

12.2.6.4 set_report_severity_file

```
void set_report_severity_file( uvm_severity severity, UVM_FILE file );
```

The member function **set_report_severity_file** shall configure the report handler to direct reports of the given *severity* to the *file* descriptor of type **UVM_FILE**. A file associated with a particular severity-id or associated with a specific id, shall take precedence over the association set by this member function.

12.2.6.5 set_report_severity_id_file

```
void set_report_severity_id_file( uvm_severity severity,
                                  const std::string& id,
                                  UVM_FILE file);
```

The member function **set_report_severity_id_file** shall configure the report handler to direct reports of the given *severity-id* pair to the given *file* descriptor of type **UVM_FILE**. A file associated with a particular *severity-id* pair shall take precedence over a file associated with *id*, which shall take precedence over a file associated with a *severity*, which takes precedence over the default file descriptor.

12.2.7 Override configuration

12.2.7.1 set_report_severity_override

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

The member function **set_report_severity_override** shall provide the ability to upgrade or downgrade a message in terms of severity given *severity*. An upgrade or downgrade for a specific id, using member function **set_report_severity_id_override**, shall take precedence over an upgrade or downgrade set by this member function.

12.2.7.2 set_report_severity_id_override

```
void set_report_severity_id_override( uvm_severity cur_severity,
                                      const std::string& id,
                                      uvm_severity new_severity );
```

The member function **set_report_severity_id_override** shall provide the ability to upgrade or downgrade a message in terms of severity given *severity*. An upgrade or downgrade for a specific *id* takes precedence over an upgrade or downgrade associated with a *severity*.

12.2.8 Report handler configuration

12.2.8.1 set_report_handler

```
void set_report_handler( uvm_report_handler* handler );
```

The member function **set_report_handler** shall set the report handler, overwriting the default instance. This allows more than one component to share the same report handler.

12.2.8.2 get_report_handler

```
uvm_report_handler* get_report_handler() const;
```

The member function **get_report_handler** shall return the underlying report handler to which most reporting tasks are delegated.

12.2.8.3 reset_report_handler

```
void reset_report_handler();
```

The member function **reset_report_handler** shall reset the underlying report handler to its default settings. This clears any settings made with the member functions **set_report_id_verbosity_hier**, **set_report_severity_id_verbosity_hier**, **set_report_severity_action_hier**, **set_report_id_action_hier**, **set_report_severity_id_action_hier**, **set_report_default_file_hier**, **set_report_severity_file_hier**, **set_report_id_file_hier**, **set_report_severity_id_file_hier** and **set_report_verbosity_level_hier**. (See Section 7.1.9).

12.3 uvm_report_handler

The class **uvm_report_handler** is the class to which most member functions in **uvm_report_object** delegate. It stores the maximum verbosity, actions, and files that affect the way reports are handled.

The report handler is not intended for direct use. See Section 12.2 for information on the UVM reporting mechanism.

The relationship between class **uvm_report_object**, which is a base class for **uvm_component**, and class **uvm_report_handler** is typically one to one, but it can be many to one if several objects of type **uvm_report_object** are configured to use the same **uvm_report_handler**. (See Section 12.2.8.1).

The relationship between an object of type **uvm_report_handler** and an object of type **uvm_report_server** is many to one.

12.3.1 Class definition

```
namespace uvm {

  class uvm_report_handler
  {
   public:
    uvm_report_handler();

    int get_verbosity_level( uvm_severity severity = UVM_INFO,
                             const std::string& id = "" );

    uvm_action get_action( uvm_severity severity,
```

```
                            const std::string& id );

    UVM_FILE get_file_handle( uvm_severity severity,
                               const std::string& id );

    virtual void report( uvm_severity severity,
                         const std::string& name,
                         const std::string& id,
                         const std::string& message,
                         int verbosity_level = UVM_MEDIUM,
                         const std::string& filename = "",
                         int line = 0,
                         uvm_report_object* client = NULL );

    std::string format_action( uvm_action action );

  }; // class uvm_report_handler

} // namespace uvm
```

12.3.2 Constructor

```
uvm_report_handler();
```

The constructor shall create and initialize a new handler object.

12.3.3 Member functions

12.3.4 get_verbosity_level

```
int get_verbosity_level( uvm_severity severity = UVM_INFO,
                         const std::string& id = "" );
```

The member function **get_verbosity_level** shall return the verbosity associated with the given *severity* and *id*.

First, if there is a verbosity associated with the pair (*severity*, *id*), return that. Else, if there is a verbosity associated with the *id*, return that. Else, return the maximum verbosity setting.

12.3.5 get_action

```
uvm_action get_action( uvm_severity severity,
                       const std::string& id );
```

The member function **get_action** shall return the action associated with the given *severity* and *id*. First, if there is an action associated with the pair(*severity*, *id*), return that. Else, if there is an action associated with the *id*, return that. Else, if there is an action associated with the *severity*, return that. Else, return the default action associated with the severity.

12.3.6 get_file_handle

```
UVM_FILE get_file_handle( uvm_severity severity,
                          const std::string& id );
```

The member function **get_file_handle** shall return the file descriptor **UVM_FILE** associated with the given *severity* and *id*. First, if there is a file handle associated with the pair(*severity*, *id*), return that. Else, if there is a file handle associated with the *id*, return that. Else, if there is a file handle associated with the *severity*, return that. Else, return the default file handle.

12.3.7 report

```
virtual void report( uvm_severity severity,
                     const std::string& name,
                     const std::string& id,
                     const std::string& message,
                     int verbosity_level = UVM_MEDIUM,
                     const std::string& filename = "",
                     int line = 0,
                      uvm_report_object* client = NULL );
```

The member function **report** shall be used by the four core reporting member functions, **uvm_report_error**, **uvm_report_info**, **uvm_report_warning**, **uvm_report_fatal**, of class **uvm_report_object**.

12.3.8 format_action

```
std::string format_action( uvm_action action );
```

The member function **format_action** shall return a string representation of the action, e.g., "DISPLAY".

12.4 uvm_report_server

The class **uvm_report_server** shall act as a global server that processes all of the reports generated by a **uvm_report_handler**.

The **uvm_report_server** is an abstract class which declares many of its member functions as pure virtual. UVM defines the class **uvm_default_report_server** as its default report server.

12.4.1 Class definition

```
namespace uvm {

  class uvm_report_server : public uvm_object
  {
   public:

    virtual void set_max_quit_count( int count, bool overridable = true ) = 0;
    virtual int get_max_quit_count() const = 0;

    virtual void set_quit_count( int quit_count ) = 0;
    virtual int get_quit_count() const = 0;

    virtual void set_severity_count( uvm_severity severity, int count ) = 0;
    virtual int get_severity_count( uvm_severity severity ) const = 0;

    virtual void set_id_count( const std::string& id, int count ) = 0;
    virtual int get_id_count( const std::string& id ) const = 0;

    virtual void get_id_set( std::vector<std::string>& q ) const = 0;
    virtual void get_severity_set( std::vector<uvm_severity>& q ) const = 0;

    void do_copy( const uvm_object& rhs );

    virtual void execute_report_message( uvm_report_message* report_message,
                                         const std::string& composed_message ) = 0;

    virtual std::string compose_report_message( uvm_report_message* report_message,
                                                const std::string& report_object_name = "") const = 0;

    virtual void report_summarize( UVM_FILE file = 0 ) const = 0;

    static void set_server( uvm_report_server* server );
    static uvm_report_server* get_server();

  }; // class uvm_report_server
```

```
} // namespace uvm
```

12.4.2 Member functions

12.4.2.1 set_max_quit_count

```
virtual void set_max_quit_count( int count, bool overridable = true ) = 0;
```

The member function **set_max_quit_count** shall set the maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The default is 0, which specifies no maximum. When argument *overridable* is set to false, the set quit count cannot be changed again.

12.4.2.2 get_max_quit_count

```
virtual int get_max_quit_count() const = 0;
```

The member function **get_max_quit_count** shall return the currently configured maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The member function shall return 0 if no maximum is set.

12.4.2.3 set_quit_count

```
virtual void set_quit_count( int quit_count ) = 0;
```

The member function **set_quit_count** shall set the current number of **UVM_QUIT** actions already passed through this **uvm_report_server**.

12.4.2.4 get_quit_count

```
virtual int get_quit_count() const = 0;
```

The member function **get_quit_count** shall return the current number of **UVM_QUIT** actions already passed through this server.

12.4.2.5 set_severity_count

```
virtual void set_severity_count( uvm_severity severity, int count ) = 0;
```

The member function **set_severity_count** shall set the counter for the given *severity* to counter value *count*.

12.4.2.6 get_severity_count

```
virtual int get_severity_count( uvm_severity severity ) const = 0;
```

The member function **get_severity_count** shall return the counter value for the given *severity*.

12.4.2.7 set_id_count

```
virtual void set_id_count( const std::string& id, int count ) = 0;
```

The member function **set_id_count** shall set the counter for reports with the given *id*.

12.4.2.8 get_id_count

```
virtual int get_id_count( const std::string& id ) const = 0;
```

The member function **get_id_count** shall return the counter for reports with the given *id*.

12.4.2.9 get_id_set

```
virtual void get_id_set( std::vector<std::string>& q ) const = 0;
```

The member function **get_id_set** shall return the set of id's already used by this **uvm_report_server**.

12.4.2.10 get_severity_set

```
virtual void get_severity_set( std::vector<uvm_severity>& q ) const = 0;
```

The member function **get_severity_set** shall return the set of severities already used by this **uvm_report_server**.

12.4.2.11 do_copy

```
void do_copy( const uvm_object& rhs );
```

The member function **do_copy** shall copy all message statistic severity, id counts to the destination **uvm_report_server**. The copy is cummulative, which means only items from the source are transferred, already existing entries are not deleted, existing entries/counts are overridden when they exist in the source set.

12.4.2.12 execute_report_message

```
virtual void execute_report_message( uvm_report_message* report_message,
                                     const std::string& composed_message ) = 0;
```

The member function **execute_report_message** shall process the provided message per the actions contained within. An applicatio could overload this member function to customize action processing.

12.4.2.13 compose_report_message

```
virtual std::string compose_report_message( uvm_report_message* report_message,
                                            const std::string& report_object_name = "") const = 0;
```

The member function **compose_report_message** shall construct the actual string sent to the file or command line from the *severity*, component *name*, report *id*, and the *message* itself. An application can overload this member function to customize report formatting.

12.4.2.14 report_summarize

```
virtual void report_summarize( UVM_FILE file = 0 ) const = 0;
```

The member function **report_summarize** shall output statistical information on the reports issued by this central report server. This information is sent to the standard output (stdout) if there is no argument specified or if the argument *file* is 0; otherwise the information is send to a file using the argument *file* as file handle. The member function **uvm_root**::**run_test** shall call this member function at the end of simulation.

12.4.2.15 set_server

```
static void set_server( uvm_report_server* server );
```

The member function **set_server** shall set the global report server to use for reporting. The report server is responsible for formatting messages. This member function is provided as a convenience wrapper around setting the report server via the member function **uvm_coreservice_t::set_report_server**.

12.4.2.16 get_server

```
static uvm_report_server* get_server() = 0;
```

The member function **get_server** shall get the global report server. This member function shall always return a valid handle to a report server. This member function is provided as a convenience wrapper around retrieving the report server via the member function **uvm_coreservice_t::get_report_server**.

12.5 uvm_default_report_server

The class **uvm_default_report_server** shall define the default implementation of the UVM report server.

12.5.1 Class definition

```
namespace uvm {

  class uvm_default_report_server : public uvm_report_server
  {
   public:

    uvm_default_report_server( const std::string& name = "uvm_default_report_server" );

    // Group: Quit count

    void set_max_quit_count( int count, bool overridable = true );
    int get_max_quit_count() const;
    void set_quit_count( int quit_count );
    int get_quit_count() const;
    void incr_quit_count();
    void reset_quit_count();
    bool is_quit_count_reached();

    // Group: Severity count

    void set_severity_count( uvm_severity severity, int count );
    int get_severity_count( uvm_severity severity ) const;
    void incr_severity_count( uvm_severity severity );
    void reset_severity_counts();
    virtual void get_severity_set( std::vector<uvm_severity>& q ) const;

    // Group: id count

    void set_id_count( const std::string& id, int count );
    int get_id_count( const std::string& id ) const;
    void incr_id_count( const std::string& id );
    virtual void get_id_set( std::vector<std::string>& q ) const;

    // Group: Message processing

    virtual void execute_report_message( uvm_report_message* report_message,
                                         const std::string& composed_message );

    virtual std::string compose_report_message( uvm_report_message* report_message,
                                                const std::string& report_object_name = "" ) const;

    virtual void report_summarize( UVM_FILE file = 0 ) const;
    virtual void do_print( const uvm_printer& printer ) const;
```

```
  }; // class uvm_default_report_server

} // namespace uvm
```

12.5.2 Constructor

```
uvm_default_report_server( const std::string& name = "uvm_default_report_server" );
```

The constructor shall create a **uvm_report_server** object, if not already created. Else, it does nothing.

12.5.3 Quit count

12.5.3.1 set_max_quit_count

```
void set_max_quit_count( int count, bool overridable = true );
```

The member function **set_max_quit_count** shall set the maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The default is 0, which specifies no maximum. When argument *overridable* is set to false, the set quit count cannot be changed again.

12.5.3.2 get_max_quit_count

```
int get_max_quit_count() const;
```

The member function **get_max_quit_count** shall return the currently configured maximum number of COUNT actions that can be tolerated before a **UVM_EXIT** action is taken. The member function shall return 0 if no maximum is set.

12.5.3.3 set_quit_count

```
void set_quit_count( int quit_count );
```

The member function **set_quit_count** shall set the current number of **UVM_QUIT** actions already passed through this **uvm_report_server**.

12.5.3.4 get_quit_count

```
int get_quit_count() const;
```

The member function **get_quit_count** shall return the current number of **UVM_QUIT** actions already passed through this server.

12.5.3.5 incr_quit_count

```
void incr_quit_count();
```

The member function **incr_quit_count** shall increase the quit count with one, i.e., the number of COUNT actions.

12.5.3.6 reset_quit_count

```
void reset_quit_count();
```

The member function **reset_quit_count** shall reset the quit count, i.e., the number of COUNT actions, to 0.

12.5.3.7 is_quit_count_reached

```
bool is_quit_count_reached();
```

The member function **is_quit_count_reached** shall return *true* when the quit counter has reached the maximum.

12.5.4 Severity count

12.5.4.1 set_severity_count

```
void set_severity_count( uvm_severity severity, int count );
```

The member function **set_severity_count** shall set the counter for the given *severity* to counter value *count*.

12.5.4.2 get_severity_count

```
int get_severity_count( uvm_severity severity ) const;
```

The member function **get_severity_count** shall return the counter value for the given *severity*.

12.5.4.3 incr_severity_count

```
void incr_severity_count( uvm_severity severity );
```

The member function **incr_severity_count** shall increase the counter value for the given *severity* with one.

12.5.4.4 reset_severity_counts

```
void reset_severity_counts();
```

The member function **reset_severity_counts** shall reset all severity counters to 0.

12.5.4.5 get_severity_set

```
virtual void get_severity_set( std::vector<uvm_severity>& q ) const = 0;
```

The member function **get_severity_set** shall return the set of severities already used by this **uvm_report_server**.

12.5.5 ID count

12.5.5.1 set_id_count

```
void set_id_count( const std::string& id, int count );
```

The member function **set_id_count** shall set the counter for reports with the given *id*.

12.5.5.2 get_id_count

```
int get_id_count( const std::string& id ) const;
```

The member function **get_id_count** shall return the counter for reports with the given *id*.

12.5.5.3 incr_id_count

```
void incr_id_count( const std::string& id );
```

The member function **incr_id_count** shall increase the counter for reports with the given *id* with one.

12.5.5.4 get_id_set

```
virtual void get_id_set( std::vector<std::string>& q ) const = 0;
```

The member function **get_id_set** shall return the set of id's already used by this **uvm_report_server**.

12.5.6 Message processing

12.5.6.1 execute_report_message

```
virtual void execute_report_message( uvm_report_message* report_message,
                                     const std::string& composed_message );
```

The member function **execute_report_message** shall process the provided message per the actions contained within. An applicatio could overload this member function to customize action processing.

12.5.6.2 compose_report_message

```
virtual std::string compose_report_message( uvm_report_message* report_message,
                                            const std::string& report_object_name = "") const;
```

The member function **compose_report_message** shall construct the actual string sent to the file or command line from the *severity*, component *name*, report *id*, and the *message* itself. An application can overload this member function to customize report formatting.

12.5.6.3 report_summarize

```
virtual void report_summarize( UVM_FILE file = 0 ) const;
```

The member function **report_summarize** shall output statistical information on the reports issued by this central report server. This information is sent to the standard output (stdout) if there is no argument specified or if the argument *file* is 0; otherwise the information is send to a file using the argument *file* as file handle. The member function **uvm_root**::**run_test** shall call this member function at the end of simulation.

12.5.6.4 do_print

```
virtual void do_print( const uvm_printer& printer ) const;
```

The member function **do_print** shall provide UVM printer formatted output of the current configuration.

12.6 uvm_report_catcher

The class **uvm_report_catcher** shall be used to catch messages issued by the **uvm report server**. Catchers are objects of type **uvm_callbacks**<**uvm_report_object**, **uvm_report_catcher**>, so all facilities in the classes **uvm_callback** and **uvm_callbacks**<**T**, **CB**> are available for registering catchers and controlling catcher state.

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

User extensions of **uvm_report_catcher** need to implement the member function **catch** in which the action to be taken on catching the report is specified. The member function **catch** 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. The catchers are processed in the order in which they are registered.

On catching a report, the member function **catch** 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 member function **issue**.

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

12.6.1 Class definition

```
namespace uvm {

  class uvm_report_catcher : public uvm_callback
  {
   public:
    typedef enum { UNKNOWN_ACTION, THROW, CAUGHT} action_e;

    uvm_report_catcher( const std::string& name = "uvm_report_catcher" );

    // Group: Current Message State
    uvm_report_object* get_client() const;
    uvm_severity get_severity() const;
    int get_verbosity() const;
    std::string get_id() const;
    std::string get_message() const;
    uvm_action get_action() const;
    std::string get_fname() const;
    int get_line() const;

    // Group: Change Message State
   protected:
    void set_severity( uvm_severity severity );
    void set_verbosity( int verbosity );
    void set_id( const std::string& id );
    void set_message( const std::string& message );
    void set_action( uvm_action action );

    // Group: Debug
    static uvm_report_catcher* get_report_catcher( const std::string& name );
    static void print_catcher( UVM_FILE file = 0 );

    // Group: Callback interface
    virtual action_e do_catch°() = 0;
```

```
    // Group: Reporting
   protected:
    void uvm_report_fatal( const std::string& id,
                           const std::string& message,
                           int verbosity,
                           const std::string& fname = "",
                           int line = 0 );

    void uvm_report_error( const std::string& id,
                           const std::string& message,
                           int verbosity,
                           const std::string& fname = "",
                           int line = 0 );

    void uvm_report_warning( const std::string& id,
                             const std::string& message,
                             int verbosity,
                             const std::string& fname = "",
                             int line = 0 );

    void uvm_report_info( const std::string& id,
                          const std::string& message,
                          int verbosity,
                          const std::string& fname = "",
                          int line = 0 );

    void issue();
    static void summarize_report_catcher( UVM_FILE file );

  }; // class uvm_report_catcher

} // namespace uvm
```

12.6.2 Constructor

```
uvm_report_catcher( const std::string& name = "uvm_report_catcher" );
```

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

12.6.3 Current message state

12.6.3.1 get_client

```
uvm_report_object* get_client() const;
```

The member function **get_client** shall return the **uvm_report_object** that has generated the message that is currently being processed.

12.6.3.2 get_severity

```
uvm_severity get_severity() const;
```

The member function **get_severity** shall return the **uvm_severity** of the message that is currently being processed. If the severity was modified by a previously executed report object (which re-threw the message), then the returned severity is the modified value.

12.6.3.3 get_verbosity

```
int get_verbosity() const;
```

The member function **get_verbosity** shall return the verbosity of the message that is currently being processed. If the verbosity was modified by a previously executed report object (which re-threw the message), then the returned verbosity is the modified value.

12.6.3.4 get_id

```
std::string get_id() const;
```

The member function **get_id** shall return the string id of the message that is currently being processed. If the id was modified by a previously executed report object (which re-threw the message), then the returned id is the modified value.

12.6.3.5 get_message

```
std::string get_message() const;
```

The member function **get_message** shall return the string message of the message that is currently being processed. If the message was modified by a previously executed report object (which re-threw the message), then the returned message is the modified value.

12.6.3.6 get_action

```
uvm_action get_action() const;
```

The member function **get_action** shall return the **uvm_action** of the message that is currently being processed. If the action was modified by a previously executed report object (which re-threw the message), then the returned action is the modified value.

12.6.3.7 get_fname

```
std::string get_fname() const;
```

The member function **get_fname** shall return the file name of the message.

12.6.3.8 get_line

```
int get_line() const;
```

The member function **get_line** shall return the line number of the message.

12.6.4 Change message state

12.6.4.1 set_severity

```
void set_severity( uvm_severity severity );
```

The member function **set_severity** shall change the severity of the message to *severity*. Any other report catchers will see the modified value.

12.6.4.2 set_verbosity

```
void set_verbosity( int verbosity );
```

The member function **set_severity** shall change the verbosity of the message to *verbosity*. Any other report catchers will see the modified value.

12.6.4.3 set_id

```
void set_id( const std::string& id );
```

The member function **set_id** shall change the id of the message to *id*. Any other report catchers will see the modified value.

12.6.4.4 set_message

```
void set_message( const std::string& message );
```

The member function **set_message** shall change the text of the message to *message*. Any other report catchers will see the modified value.

12.6.4.5 set_action

```
void set_action( uvm_action action );
```

The member function **set_action** shall change the action of the message to *action*. Any other report catchers will see the modified value.

12.6.5 Debug

12.6.5.1 get_report_catcher

```
static uvm_report_catcher* get_report_catcher( const std::string& name );
```

The member function **get_report_catcher** shall return the first report catcher that has name.

12.6.5.2 print_catcher

```
static void print_catcher( UVM_FILE file = 0 );
```

The member function **print_catcher** shall print information about all of the report catchers that are registered. For finer grained detail, the member function **uvm_callbacks**<T,CB>::**display** can be used by calling **uvm_report_cb**::**display**(**uvm_report_object**).

12.6.6 Callback interface

12.6.6.1 do_catch° (catch†)

```
virtual action_e do_catch°() = 0
```

The member function **do_catch**° shall be called for each registered report catcher. The member functions in the current message state interface can be used to access information about the current message being processed (see Section 12.6.3).

12.6.7 Reporting

12.6.7.1 uvm_report_fatal

```
void uvm_report_fatal( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
                       int line = 0 );
```

The member function **uvm_report_fatal** shall issue a fatal message using the current messages report object. This message shall bypass any message catching callbacks.

12.6.7.2 uvm_report_error

```
void uvm_report_error( const std::string& id,
                       const std::string& message,
                       int verbosity,
                       const std::string& fname = "",
                       int line = 0 );
```

The member function **uvm_report_error** shall issue an error message using the current messages report object. This message shall bypass any message catching callbacks.

12.6.7.3 uvm_report_warning

```
void uvm_report_warning( const std::string& id,
                         const std::string& message,
                         int verbosity,
                         const std::string& fname = "",
                         int line = 0 );
```

The member function **uvm_report_warning** shall issue a warning message using the current messages report object. This message shall bypass any message catching callbacks.

12.6.7.4 uvm_report_info

```
void uvm_report_info( const std::string& id,
                      const std::string& message,
                      int verbosity,
                      const std::string& fname = "",
                      int line = 0 );
```

The member function **uvm_report_info** shall issue an info message using the current messages report object. This message shall bypass any message catching callbacks.

12.6.7.5 issue

```
void issue();
```

The member function **issue** shall immediately issue the message which is currently being processed. This is useful if the message is being **CAUGHT** but should still be emitted. Issuing a message shall update the report server stats, possibly multiple times if the message is not **CAUGHT**.

12.6.7.6 summarize_report_catcher

```
static void summarize_report_catcher( UVM_FILE file );
```

The member function **summarize_report_catcher** shall print the statistics for the active catchers. It shall be called automatically by the member function **uvm_report_server**::**summarize**.

13. Macros

UVM-SystemC defines macros for the following functions:

- Component and object registration.
- Reporting.
- Sequence execution.
- Callbacks.

13.1 Component and object registration macros

These macros shall register components and objects with the **uvm_factory**, using the component registry **uvm_component_registry** or **uvm_object_registry**, respectively. In addition, they shall implement the member functions **get_type** and **get_type_name** to facilitate debugging and factory configuration or overrides.

13.1.1 Macro definitions

```
namespace uvm {

  #define UVM_OBJECT_UTILS( implementation-defined ) implementation-defined
  #define UVM_OBJECT_PARAM_UTILS( implementation-defined ) implementation-defined
  #define UVM_COMPONENT_UTILS( implementation-defined ) implementation-defined
  #define UVM_COMPONENT_PARAM_UTILS( implementation-defined ) implementation-defined

} // namespace uvm
```

13.1.2 UVM_OBJECT_UTILS, UVM_OBJECT_PARAM_UTILS

```
#define UVM_OBJECT_UTILS( implementation-defined ) implementation-defined
#define UVM_OBJECT_PARAM_UTILS( implementation-defined ) implementation-defined
```

The macros **UVM_OBJECT_UTILS** and **UVM_OBJECT_PARAM_UTILS** shall implement the following functionality:

- Implement the virtual member function **get_type_name** with the following signature:
  `virtual const std::string get_type_name() const;`
  This member function shall return the name of the class, which is provided as argument to this macro, as string.
- Implement the static member function **get_type** with the following signature:
  `static uvm_object_registry<classname>* get_type();`
  This member function shall return the factory proxy object as pointer of type **uvm_object_registry**.
- Register the class with the factory.

NOTE—An implementation may use the concept of variadic macros to be able to accept a variable number of macro arguments.

13.1.3 UVM_COMPONENT_UTILS, UVM_COMPONENT_PARAM_UTILS

```
#define UVM_COMPONENT_UTILS( implementation-defined ) implementation-defined
#define UVM_COMPONENT_PARAM_UTILS( implementation-defined ) implementation-defined
```

The macros **UVM_COMPONENT_UTILS** and **UVM_COMPONENT_PARAM_UTILS** shall implement the following functionality:

- Implement the virtual member function **get_type_name** with the following signature:
  ```
  virtual const std::string get_type_name() const;
  ```
  This member function shall return the name of the class, which is provided as argument to this macro, as string.
- Implement the static member function **get_type** with the following signature:
  ```
  static uvm_component_registry<classname>* get_type();
  ```
  This member function shall return the factory proxy object as pointer of type **uvm_component_registry**.
- Register the class with the factory

NOTE—An implementation may use the concept of variadic macros to be able to accept a variable number of macro arguments.

13.2 Reporting macros

The report macros shall provide additional functionality to the UVM reporting classes to facilitate efficient filtering messages based on verbosity, id and severity information, as well as annotating file and line number information to the reported messages.

13.2.1 Macro definitions

```
namespace uvm {

  #define UVM_INFO( ID, MSG, VERBOSITY ) implementation-defined
  #define UVM_WARNING( ID, MSG ) implementation-defined
  #define UVM_ERROR( ID, MSG ) implementation-defined
  #define UVM_FATAL( ID, MSG ) implementation-defined

} // namespace uvm
```

13.2.2 UVM_INFO

```
#define UVM_INFO( ID, MSG, VERBOSITY ) implementation-defined
```

The macro **UVM_INFO** shall only call member function **uvm_report_info** if argument VERBOSITY is lower than the configured verbosity of the associated reporter. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_info** by means of using the predefined macros __FILE__ and __LINE__.

13.2.3 UVM_WARNING

```
#define UVM_WARNING( ID, MSG ) implementation-defined
```

The macro **UVM_WARNING** shall call the member function **uvm_report_warning** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter’s verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_warning** by means of using the predefined macros __FILE__ and __LINE__.

13.2.4 UVM_ERROR

```
#define UVM_ERROR( ID, MSG ) implementation-defined
```

The macro **UVM_ERROR** shall call the member function **uvm_report_error** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_error** by means of using the predefined macros __FILE__ and __LINE__.

13.2.5 UVM_FATAL

```
#define UVM_FATAL( ID, MSG ) implementation-defined
```

The macro **UVM_FATAL** shall call member function **uvm_report_fatal** with a verbosity of **UVM_NONE**. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. Argument ID is given as the message tag and argument MSG is given as the message text. The file and line number are also sent to the member function **uvm_report_fatal** by means of using the predefined macros __FILE__ and __LINE__.

13.3 Sequence execution macros

The sequence execution macros are shall provide a convenience layer to start sequences or sequence items on a default sequencer, if not specified, or on another sequencer if specified.

NOTE—It is strongly recommended not to use the sequence execution macros in an application. Instead, for a sequence item to start, it is recommended to use the member functions **start_item** (see Section 9.3.7.2) and **finish_item** (see Section 9.3.7.3). To start a sequence, it is recommended to use the member function **start** (see Section 9.3.4.1).

13.3.1 Macro definitions

```
namespace uvm {

  #define UVM_DO( SEQ_OR_ITEM ) implementation-defined
  #define UVM_DO_PRI( SEQ_OR_ITEM, PRIORITY ) implementation-defined
  #define UVM_DO_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
  #define UVM_DO_ON_PRI( SEQ_OR_ITEM, SEQR, PRIORITY ) implementation-defined
  #define UVM_CREATE( SEQ_OR_ITEM ) implementation-defined
  #define UVM_CREATE_ON( SEQ_OR_ITEM, SEQR ) implementation-defined

  #define UVM_DECLARE_P_SEQUENCER( SEQR ) implementation-defined

} // namespace uvm
```

13.3.2 UVM_DO

```
#define UVM_DO( SEQ_OR_ITEM ) implementation-defined
```

The macro **UVM_DO** shall start the execution of a sequence or sequence item. It takes as an argument *SEQ_OR_ITEM*, which is an object of type **uvm_sequence_item** or object of type **uvm_sequence**.

In the case of a sequence, the sub-sequence shall be started using member function **uvm_sequence_base**::**start** with argument *call_pre_post* set to false. In the case of a sequence item, the item shall be sent to the driver through the associated sequencer.

NOTE—Randomization is not yet supported in UVM-SystemC.

13.3.3 UVM_DO_PRI

```
#define UVM_DO_PRI( SEQ_OR_ITEM, PRIORITY ) implementation-defined
```

The macro **UVM_DO_PRI** shall implement the same functionality as **UVM_DO**, except that the sequence item or sequence is executed with the priority specified in the argument *PRIORITY*.

13.3.4 UVM_DO_ON

```
#define UVM_DO_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
```

The macro **UVM_DO_ON** shall implement the same functionality as **UVM_DO**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.5 UVM_DO_ON_PRI

```
#define UVM_DO_ON_PRI( SEQ_OR_ITEM, SEQR, PRIORITY ) implementation-defined
```

The macro **UVM_DO_ON_PRI** shall implement the same functionality as **UVM_DO_PRI**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.6 UVM_CREATE

```
#define UVM_CREATE( SEQ_OR_ITEM ) implementation-defined
```

The macro **UVM_CREATE** shall create and register the sequence item or sequence using the factory. It intentionally does not start the execution.

NOTE—After calling this member function, an application can manually set values and start the execution.

13.3.7 UVM_CREATE_ON

```
#define UVM_CREATE_ON( SEQ_OR_ITEM, SEQR ) implementation-defined
```

The macro **UVM_CREATE_ON** shall implement the same functionality as **UVM_CREATE**, except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified argument *SEQR*.

13.3.8 UVM_DECLARE_P_SEQUENCER

```
#define UVM_DECLARE_P_SEQUENCER( SEQR ) implementation-defined
```

The macro **UVM_DECLARE_P_SEQUENCER** shall declare a variable *p_sequencer* whose type is specified by the argument *SEQR*.

13.4 Callback macros

The callback macros shall register and execute callbacks which are derived from class **uvm_callbacks**.

13.4.1 Macro definitions

```
namespace uvm {

  #define UVM_REGISTER_CB( T, CB ) implementation-defined
  #define UVM_DO_CALLBACKS( T, CB, METHOD ) implementation-defined

} // namespace uvm
```

13.4.2 UVM_REGISTER_CB

```
#define UVM_REGISTER_CB( T, CB ) implementation-defined
```

The macro **UVM_REGISTER_CB** shall register the given callback type *CB* with the given object type *T*. If a type-callback pair is not registered, then a warning is issued if an attempt is made to use the pair (add, delete, etc.).

13.4.3 UVM_DO_CALLBACKS

```
#define UVM_DO_CALLBACKS( T, CB, METHOD ) implementation-defined
```

The macro **UVM_DO_CALLBACKS** shall call 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 executes all of the callbacks associated with the calling object (i.e. this object). The macro takes three arguments:

— *CB* is the class type of the callback objects to execute. The class type shall have a function signature that matches the argument *METHOD*.
— *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, with all required arguments as if they were invoked directly.

14. TLM classes

The TLM classes of UVM-SystemC shall be derived from the SystemC TLM interface definitions as defined in IEEE Std. 1666-2011. As communication between UVM components is primarily based on TLM-1 message passing semantics, dedicated ports and exports are defined compliant with these semantics.

The following TLM-1 ports are defined in UVM-SystemC:

— Ports based on TLM-1 blocking interfaces: **uvm_blocking_put_port**, **uvm_blocking_get_port**, **uvm_blocking_peek_port**, and **uvm_blocking_get_peek_port**.

— Ports based on TLM-1 non-blocking interfaces: **uvm_nonblocking_put_port**, **uvm_nonblocking_get_port**, **uvm_nonblocking_peek_port**, and **uvm_nonblocking_get_peek_port**.

— Analysis port and export classes: **uvm_analysis_port**, **uvm_analysis_export**, and **uvm_analysis_imp**.

— Request-response channel class: **uvm_tlm_req_rsp_channel**.

— Sequencer interface classes: **uvm_sqr_if_base**, **uvm_seq_item_pull_port**, **uvm_seq_item_pull_export**, and **uvm_seq_item_pull_imp**.

NOTE 1—UVM-SystemC does not define TLM-1 FIFO and FIFO interface classes. Instead, an application should use the SystemC FIFO base classes **tlm::tlm_fifo** or **tlm::tlm_analysis_fifo**, or FIFO interfaces **tlm::tlm_fifo_debug_if**, **tlm::tlm_fifo_put_if**, and **tlm::tlm_fifo_get_if**.

NOTE 2—UVM-SystemC does not define the TLM-2.0 blocking and non-blocking transport interfaces, direct memory interface (DMI), nor a debug transport interface. Instead, an application should use the SystemC TLM-2.0 interfaces.

14.1 uvm_blocking_put_port

The class **uvm_blocking_put_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_put_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.1.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_blocking_put_port : public uvm_port_base< tlm::tlm_blocking_put_if<T> >
  {
   public:
    // Constructors
    uvm_blocking_put_port();
    uvm_blocking_put_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void put( const T& val );

  }; // class uvm_blocking_put_port

} // namespace uvm
```

14.1.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.1.3 Constructor

```
uvm_blocking_put_port();
uvm_blocking_put_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking put interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.1.4 Member functions

14.1.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string “**uvm**::**uvm_blocking_put_port**”.

14.1.4.2 put

```
virtual void put( const T& val );
```

The member function **put** shall send the transaction of type T to the recipient. It shall call the member function **put** of the associated interface which is bound to this port.

According to the TLM-1 blocking put semantics, the member function **put** shall not return until the recipient has indicated that the transaction object has been processed, by calling member function **get** or **peek**. Subsequent calls to the member function **put** shall be treated as distinct transaction instances, regardless of whether or not the same transaction object or message is passed.

14.2 uvm_blocking_get_port

The class **uvm_blocking_get_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_get_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.2.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_blocking_get_port : public uvm_port_base< tlm::tlm_blocking_get_if<T> >
  {
   public:
    // Constructors
    uvm_blocking_get_port();
    uvm_blocking_get_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void get( T& val );

  }; // class uvm_blocking_get_port

} // namespace uvm
```

14.2.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.2.3 Constructor

```
uvm_blocking_get_port();
uvm_blocking_get_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking get interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.2.4 Member functions

14.2.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_get_port**".

14.2.4.2 get

```
virtual void get( T& val );
```

The member function **get** shall retrieve a transaction of type T from the sender. It shall call the member function **get** of the associated interface which is bound to this port.

According to the TLM-1 blocking get semantics, the member function **get** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **get** shall return a different transaction object. This actually means that a call to **get** shall consume the transaction from the sender.

14.3 uvm_blocking_peek_port

The class **uvm_blocking_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.3.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_blocking_peek_port : public uvm_port_base< tlm::tlm_blocking_peek_if<T> >
  {
   public:
    // Constructors
    uvm_blocking_peek_port();
    uvm_blocking_peek_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void peek( T& val ) const;

  }; // class uvm_blocking_peek_port

} // namespace uvm
```

14.3.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.3.3 Constructor

```
uvm_blocking_peek_port();
uvm_blocking_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.3.4 Member functions

14.3.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_peek_port**".

14.3.4.2 peek

```
virtual void peek( T& val ) const;
```

The member function **peek** shall retrieve a transaction of type T from the sender. It shall call the member function **peek** of the associated interface which is bound to this port.

According to the TLM-1 blocking peek semantics, the member function **peek** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **peek** shall return exactly the same transaction object. This actually means that a call to **peek** shall not consume the transaction from the sender. A transaction shall only be consumed by means of a call to **get**.

14.4 uvm_blocking_get_peek_port

The class **uvm_blocking_get_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_blocking_get_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.4.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_blocking_get_peek_port : public uvm_port_base< tlm::tlm_blocking_get_peek_if<T> >
  {
   public:
    // Constructor
    uvm_blocking_get_peek_port();
    uvm_blocking_get_peek_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void get( T& val );
    virtual void peek( T& val ) const;

  }; // class uvm_blocking_get_peek_port

} // namespace uvm
```

14.4.2 Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

14.4.3 Constructor

```
uvm_blocking_get_peek_port();
uvm_blocking_get_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 blocking get and peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.4.4 Member functions

14.4.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_blocking_get_peek_port**".

14.4.4.2 get

```
virtual void get( T& val );
```

The member function **get** shall retrieve a transaction of type T from the sender. It shall call the member function **get** of the associated interface which is bound to this port.

According to the TLM-1 blocking get semantics, the member function **get** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **get** shall return a different transaction object. This actually means that a call to **get** shall consume the transaction from the sender.

14.4.4.3 peek

```
virtual void peek( T& val ) const;
```

The member function **peek** shall retrieve a transaction of type T from the sender. It shall call the member function **peek** of the associated interface which is bound to this port.

According to the TLM-1 blocking peek semantics, the member function **peek** shall not return until a transaction object has been delivered by the sender by means of its member function **put**. Subsequent calls to the member function **peek** shall return exactly the same transaction object. This actually means that a call to **peek** shall not consume the transaction from the sender. A transaction shall only be consumed by means of a call to **get**.

14.5 uvm_nonblocking_put_port

The class **uvm_nonblocking_put_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_nonblocking_put_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.5.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_nonblocking_put_port : public uvm_port_base< tlm::tlm_nonblocking_put_if<T> >
  {
   public:
    // Constructors
    uvm_nonblocking_put_port();
    uvm_nonblocking_put_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual bool try_put( const T& val );
    virtual bool can_put() const;

  }; // class uvm_nonblocking_put_port

} // namespace uvm
```

14.5.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.5.3 Constructor

```
uvm_nonblocking_put_port();
uvm_nonblocking_put_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 non-blocking put interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.5.4 Member functions

14.5.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_nonblocking_put_port**".

14.5.4.2 try_put

```
virtual bool try_put( const T& val );
```

The member function **try_put** shall send the transaction of type T to the recipient, if possible. It shall call the corresponding non-blocking put member function of the associated interface which is bound to this port. If the recipient is able to respond immediately, then the member function shall return true. Otherwise, the member function shall return false, and shall not accept or return the next transaction.

14.5.4.3 can_put

```
virtual bool can_put() const;
```

The member function **can_put** shall return true if the recipient is able to respond immediately; otherwise it shall return false.

14.6 uvm_nonblocking_get_port

The class **uvm_nonblocking_get_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_nonblocking_get_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.6.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_nonblocking_get_port : public uvm_port_base< tlm::tlm_nonblocking_get_if<T> >
  {
   public:
    // Constructor
    uvm_nonblocking_get_port();
    uvm_nonblocking_get_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual bool try_get( T& val );
    virtual bool can_get() const;

  }; // class uvm_nonblocking_get_port

} // namespace uvm
```

14.6.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.6.3 Constructor

```
uvm_nonblocking_get_port();
uvm_nonblocking_get_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 non-blocking get interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.6.4 Member functions

14.6.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_nonblocking_get_port**".

14.6.4.2 can_get

```
virtual bool can_get() const;
```

The member function **can_get** shall return true if a new transaction can be provided immediately upon request. Otherwise it shall return false.

14.7 uvm_nonblocking_peek_port

The class **uvm_nonblocking_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_nonblocking_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.7.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_nonblocking_peek_port : public uvm_port_base< tlm::tlm_nonblocking_peek_if<T> >
  {
   public:
    // Constructors
    uvm_nonblocking_peek_port();
    uvm_nonblocking_peek_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual bool try_peek( T& val );
    virtual bool can_peek() const;

  }; // class uvm_nonblocking_peek_port

} // namespace uvm
```

14.7.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.7.3 Constructor

```
uvm_nonblocking_peek_port();
uvm_nonblocking_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 non-blocking peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.7.4 Member functions

14.7.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_nonblocking_peek_port**".

14.7.4.2 try_peek

```
virtual bool try_peek( T& val );
```

The member function **try_peek** shall retrieve a new transaction of type T without consuming it. It shall call the corresponding non-blocking peek member function of the associated interface which is bound to this port.

If a transaction is immediately available, then it is written to the argument *val* and the member function shall return true. Otherwise, the output argument is not modified and the member function shall return false.

14.7.4.3 can_peek

```
virtual bool can_peek() const;
```

The member function **can_peek** shall return true if a new transaction can be provided immediately upon request. Otherwise it shall return false.

14.8 uvm_nonblocking_get_peek_port

The class **uvm_nonblocking_get_peek_port** offers a convenience layer for UVM users to access the SystemC TLM-1 blocking interface **tlm**::**tlm_nonblocking_get_peek_if**. As this port class shall be derived from class **uvm_port_base**, it inherits the UVM specific member functions **connect**, **get_name**, **get_full_name** and **get_type_name**.

14.8.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_nonblocking_get_peek_port
  : public uvm_port_base< tlm::tlm_nonblocking_get_peek_if<T> >
  {
   public:
    // Constructors
    uvm_nonblocking_get_peek_port();
    uvm_nonblocking_get_peek_port( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual bool try_get( T& val );
    virtual bool can_get() const;
    virtual bool try_peek( T& val );
    virtual bool can_peek() const;

  }; // class uvm_nonblocking_get_peek_port

} // namespace uvm
```

14.8.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

14.8.3 Constructor

```
uvm_nonblocking_get_peek_port();
uvm_nonblocking_get_peek_port( const std::string& name );
```

The constructor shall create a new port with TLM-1 non-blocking get and peek interface semantics. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

14.8.4 Member functions

14.8.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_nonblocking_get_peek_port**".

14.8.4.2 try_get

```
virtual bool try_get( T& val );
```

The member function **try_get** shall retrieve a new transaction of type T. It shall call the corresponding non-blocking get member function of the associated interface which is bound to this port.

If a transaction is immediately available, then it is written to the argument *val* and the member function shall return true. Otherwise, the output argument is not modified and the member function shall return false.

14.8.4.3 can_get

```
virtual bool can_get() const;
```

The member function **can_get** shall return true if a new transaction can be provided immediately upon request. Otherwise it shall return false.

14.8.4.4 try_peek

```
virtual bool try_peek( T& val );
```

The member function **try_peek** shall retrieve a new transaction of type T without consuming it. It shall call the corresponding non-blocking peek member function of the associated interface which is bound to this port.

If a transaction is immediately available, then it is written to the argument *val* and the member function shall return true. Otherwise, the output argument is not modified and the member function shall return false.

14.8.4.5 can_peek

```
virtual bool can_peek() const;
```

The member function **can_peek** shall return true if a new transaction can be provided immediately upon request. Otherwise it shall return false.

14.9 uvm_analysis_port

The class **uvm_analysis_port** offers a convenience layer for UVM users and is compatible with the SystemC **tlm**::**tlm_analysis_port**, since it shall be derived from this class. Primary reason to introduce this derived port class is to offer the UVM specific member function **connect** as alternative to the SystemC **bind** and **operator()** to connect analysis ports with exports.

14.9.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_analysis_port : public tlm::tlm_analysis_port<T>
  {
   public:
    // Constructors
    uvm_analysis_port();
    uvm_analysis_port( const std::string& name );

    // member functions
    virtual const std::string get_type_name() const;
```

```
    virtual void connect( tlm::tlm_analysis_if<T>& _if );
    void write( const T& t );

  }; // class uvm_analysis_port

} // namespace uvm
```

14.9.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

14.9.3 Constructor

```
uvm_analysis_port();
uvm_analysis_port( const std::string& name );
```

The constructor shall create a new analysis port. If specified, the argument *name* shall define the name of the port. Otherwise, the name of the port is implementation-defined.

NOTE—UVM-SystemC does not define, in contrast to UVM-SystemVerilog, the constructor arguments *min_size* and *max_size* to specify the minimum and maximum number of interfaces, respectively, that are connected to this port by the end of elaboration.

14.9.4 Member functions

14.9.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string “**uvm**::**uvm_analysis_port**”.

14.9.4.2 connect

```
virtual void connect( tlm::tlm_analysis_if<T>& _if );
```

The member function **connect** shall register the subscriber passed as an argument, so that any call to the member function **write** of such analysis port instance shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis port instance.

NOTE 1—The member function **connect** implements the same functionality as the SystemC member function **bind**.

NOTE 2—There may be zero subscribers registered with any given analysis port instance, in which case calls to the member function **write** shall not be propagated.

14.9.4.3 write

```
void write( const T& t );
```

The member function **write** shall call the member function **write** of every subscriber which is bound to this analysis port, by passing on the argument as a const reference.

14.10 uvm_analysis_export

The class **uvm_analysis_export** offers a convenience layer for UVM users and is compatible with the SystemC export type **sc_core**::**sc_export**<**tlm**::**tlm_analysis_if**<T> > since it shall be derived from this class. Primary reason to introduce this export class is to offer the member function **connect** as alternative to the SystemC **bind** and **operator()** to connect analysis ports with exports.

14.10.1 Class definition

```
namespace uvm {

  template <typename T>
  class uvm_analysis_export : public sc_core::sc_export< tlm::tlm_analysis_if<T> >
  {
   public:
    // Constructors
    uvm_analysis_export();
    uvm_analysis_export( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void connect( tlm::tlm_analysis_if<T>& _if );

  }; // class uvm_analysis_export

} // namespace uvm
```

14.10.2 Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

14.10.3 Constructor

```
uvm_analysis_export();
uvm_analysis_export( const std::string& name );
```

The constructor shall create a new analysis export. If specified, the argument *name* shall define the name of the export. Otherwise, the name of the export is implementation-defined.

NOTE—UVM-SystemC does not define, in contrast to UVM-SystemVerilog, the constructor arguments *min_size* and *max_size* to specify the minimum and maximum number of interfaces, respectively, that are connected to this port by the end of elaboration.

14.10.4 Member functions

14.10.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_analysis_export**".

14.10.4.2 connect

```
virtual void connect( tlm::tlm_analysis_if<T>& _if );
```

The member function **connect** shall register the subscriber passed as an argument, so that any call to the member function **write** of such analysis export instance shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis export instance.

NOTE 1—The member function **connect** implements the same functionality as the SystemC member function **bind**.

NOTE 2—There may be zero subscribers registered with any given analysis export instance, in which case calls to the member function **write** shall not be propagated.

14.11 uvm_analysis_imp

The class **uvm_analysis_imp** shall serve as termination point of analysis port and export connections. It shall call the member function **write** of the component type passed as second template argument via its own member function **write**, without modification of the value passed to it.

14.11.1 Class definition

```
namespace uvm {

  template <typename T = int, typename IMP = int>
  class uvm_analysis_imp : public tlm::tlm_analysis_port<T>
  {
   public:
    // Constructors
    uvm_analysis_imp();
    uvm_analysis_imp( const std::string& name );

    // Member functions
    virtual const std::string get_type_name() const;
    virtual void connect( tlm::tlm_analysis_if<T>& _if );
    void write( const T& t );

  }; // class uvm_analysis_imp

} // namespace uvm
```

14.11.2 Template parameters

The template parameter T specifies the type of transaction to be communicated by the analysis port. The template parameter IMP specifies the component type which implements the member function **write**.

14.11.3 Constructors

```
uvm_analysis_imp();
uvm_analysis_imp( const std::string& name );
```

The constructor shall create a new analysis implementation. If specified, the argument *name* shall define the name of the export. Otherwise, the name of the export is implementation-defined.

14.11.4 Member functions

14.11.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string “**uvm**::**uvm_analysis_imp**”.

14.11.4.2 connect

```
virtual void connect( tlm::tlm_analysis_if<T>& _if );
```

The member function **connect** shall register the subscriber passed as an argument, so that any call to the member function **write** of such analysis implementation instance shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis export instance.

NOTE—The member function **connect** implements the same functionality as the SystemC member function **bind**.

14.11.4.3 write

```
void write( const T& t );
```

The member function **write** shall shall call the member function **write** of the associated subscriber which is specified as second template argument, by passing on the argument as a const reference.

14.12 uvm_tlm_req_rsp_channel

The class **uvm_tlm_req_rsp_channel** offers a convenience layer for UVM users and is compatible with the SystemC **tlm**::**tlm_req_rsp_channel**, since it shall be derived from this class. It offers some UVM additional capabilities such as the analysis ports for request and response monitoring.

The class **uvm_tlm_req_rsp_channel** contains a request FIFO of default type **tlm**::**tlm_fifo**<REQ> and response FIFO of default type **tlm**::**tlm_fifo**<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.

14.12.1 Class definition

```
namespace uvm {

  template < typename REQ,
             typename RSP = REQ,
             typename REQ_CHANNEL = tlm::tlm_fifo<REQ>,
             typename RSP_CHANNEL = tlm::tlm_fifo<RSP> >
  class uvm_tlm_req_rsp_channel
  : public tlm::tlm_req_rsp_channel<REQ, RSP, REQ_CHANNEL, RSP_CHANNEL>
  {
   public:

    // Ports and exports
    uvm_analysis_port<REQ> request_ap;
    uvm_analysis_port<RSP> response_ap;
    sc_core::sc_export< tlm::tlm_fifo_put_if<REQ> > put_request_export;
    sc_core::sc_export< tlm::tlm_fifo_put_if<RSP> > put_response_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_request_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_response_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_peek_request_export;
    sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_peek_response_export;
    sc_core::sc_export< tlm::tlm_master_if<REQ, RSP> > master_export;
    sc_core::sc_export< tlm::tlm_slave_if<REQ, RSP> >  slave_export;

    // Constructors
    uvm_tlm_req_rsp_channel( int req_size = 1, int rsp_size = 1 );
    uvm_tlm_req_rsp_channel( uvm_component_name name, int req_size = 1, int rsp_size = 1 );

  }; // class uvm_tlm_req_rsp_channel

} // namespace uvm
```

14.12.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. The template parameters REQ_CHANNEL and RSP_CHANNEL specify the type of the request and response FIFO, respectively. If parameters REQ_CHANNEL or RSP_CHANNEL are not specified, the interface uses FIFOs of type **tlm**::**tlm_fifo**.

14.12.3 Ports and exports

14.12.3.1 request_ap

```
uvm_analysis_port<REQ> request_ap;
```

The analysis port **request_ap** shall send the request transactions, which are passed via the member function **put** or **nb_put** (via any port connected to the export **put_request_export**), via its member function **write**, to all connected analysis exports and imps.

14.12.3.2 response_ap

```
uvm_analysis_port<RSP> response_ap;
```

The analysis port **response_ap** shall send the response transactions, which are passed via the member function **put** or **nb_put** (via any port connected to the export **put_response_export**), via its member function **write**, to all connected analysis exports and imps.

14.12.3.3 put_request_export

```
sc_core::sc_export< tlm::tlm_fifo_put_if<REQ> > put_request_export;
```

The export **put_request_export** shall provide both the blocking and non-blocking **put** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_put_if**, being member functions **put**, **nb_put** and **nb_can_put**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

14.12.3.4 put_response_export

```
sc_core::sc_export< tlm::tlm_fifo_put_if<RSP> > put_response_export;
```

The export **put_response_export** shall provide both the blocking and non-blocking **put** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_put_if**, being **put**, **nb_put** and **nb_can_put**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

14.12.3.5 get_request_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_request_export;
```

The export **get_request_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

NOTE—This member function is functionally equivalent to **get_peek_request_export**.

14.12.3.6 get_response_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_response_export;
```

The export **get_response_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

NOTE—This member function is functionally equivalent to **get_peek_response_export**.

14.12.3.7 get_peek_request_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<REQ> > get_peek_request_export;
```

The export **get_peek_request_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the request FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

NOTE—This member function is functionally equivalent to **get_request_export**.

14.12.3.8 get_peek_response_export

```
sc_core::sc_export< tlm::tlm_fifo_get_if<RSP> > get_peek_response_export;
```

The export **get_peek_response_export** shall provide both the blocking and non-blocking **get** and **peek** interface member functions to the response FIFO based on interface **tlm**::**tlm_fifo_get_if**, being **get**, **nb_get**, **nb_can_get**, **peek**, **nb_peek** and **nb_can_peek**. Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

NOTE—This member function is functionally equivalent to **get_response_export**.

14.12.3.9 master_export

```
sc_core::sc_export< tlm::tlm_master_if<REQ, RSP> > master_export;
```

The export **master_export** shall provide a single interface that allows a master to put requests and get or peek responses. It is a combination of the functionality offered by the exports **put_request_export** and **get_peek_response_export**.

14.12.3.10 slave_export

```
sc_core::sc_export< tlm::tlm_slave_if<REQ, RSP> > slave_export;
```

The export **slave_export** shall provide a single interface that allows a slave to get or peek requests and to put responses. It is a combination of the functionality offered by the exports **get_peek_request_export** and **put_response_export**.

14.12.4 Constructors

```
uvm_tlm_req_rsp_channel( int req_size = 1, int rsp_size = 1 );
uvm_tlm_req_rsp_channel( uvm_component_name name, int req_size = 1, int rsp_size = 1 );
```

The constructor shall create a new TLM-1 interface containing a request and response FIFO. The argument *req_size* specifies the size of the request FIFO. The argument *rsp_size* specifies the size of the response FIFO. If not specified, default size of these FIFOs is 1. If specified, the argument *name* shall define the name of the interface. Otherwise, the name of the interface is implementation-defined.

14.13 uvm_sqr_if_base

The class **uvm_sqr_if_base** shall define an interface for sequence drivers to communicate with sequencers. The driver requires the interface via a port, and the sequencer implements it and provides it via an export.

14.13.1 Class definition

```
namespace uvm {

  template <typename REQ, typename RSP = REQ>
  class uvm_sqr_if_base : public virtual sc_core::sc_interface
  {
   public:
    // Member functions
    virtual void get_next_item( REQ& req ) = 0;
    virtual bool try_next_item( REQ& req ) = 0;
    virtual void item_done( const RSP& item ) = 0;
    virtual void item_done() = 0;
    virtual void put( const RSP& rsp ) = 0;
    virtual void get( REQ& req ) = 0;
    virtual void peek( REQ& req ) = 0;

   protected:
    // Constructor
    uvm_sqr_if_base();

}; // class uvm_sqr_if_base

} // namespace uvm
```

14.13.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types shall be a derivative of class **uvm_sequence_item**.

14.13.3 Member functions

14.13.3.1 get_next_item

```
virtual void get_next_item( REQ& req ) = 0;
```

The member function **get_next_item** shall retrieve 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, 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 member function **wait_for_grant** (see Section 9.3.7.4).

c) The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see Section 9.3.4.4).

d) The chosen sequence item is randomized.

e) The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see Section 9.3.4.7).

f) Return with a reference to the item.

Once member function **get_next_item** is called, the member function **item_done** needs to be called to indicate the completion of the request to the sequencer.

14.13.3.2 try_next_item

```
virtual bool try_next_item( REQ& req ) = 0;
```

The member function **try_next_item** shall retrieve the next available item from a sequence if one is available. If available, it shall return true. Otherwise, the member function shall return false. The following steps occur on this call:

a) Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, the member function returns false.

b) The chosen sequence returns from member function **uvm_sequence_base**::**wait_for_grant** (see Section 9.3.7.4).

c) The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see Section 9.3.4.4).

d) The chosen sequence item is randomized.

e) The chosen sequence **uvm_sequence_base**::**post_do** is called (see Section 9.3.4.7).

f) Return with a reference to the item.

Once the member function **try_next_item** is called, the member function **item_done** shall be called to indicate the completion of the request to the sequencer. This removes the request item from the sequencer FIFO.

14.13.3.3 item_done

```
virtual void item_done( const RSP& item ) = 0;
virtual void item_done() = 0;
```

The member function **item_done** shall indicate that the request is completed to the sequencer. Any **uvm_sequence_base**::**wait_for_item_done** calls made by a sequence for this item shall return.

The current item is removed from the sequencer FIFO.

If a response item is provided, then it shall be sent back to the requesting sequence. The response item shall have its sequence ID and transaction ID set correctly, using the member function **uvm_sequence_item**::**set_id_info**.

Before the member function **item_done** is called, any calls to the member function **peek** retrieves the current item that was obtained by member function **get_next_item**. After the member function **item_done** is called, member function **peek** causes the sequencer to arbitrate for a new item.

14.13.3.4 get

```
virtual void get( REQ& req) = 0;
```

The member function **get** shall retrieve 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, 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 member function **uvm_sequence_base**::**wait_for_grant** (see Section 9.3.7.4).
c) The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see Section 9.3.4.4).
d) The chosen sequence item is randomized.
e) The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see Section 9.3.4.7).
f) Indicate **item_done** to the sequencer.
g) Return with a reference to the item.

When the member function **get** is called, the member function **item_done** may not be called. A new item can be obtained by calling the member function **get** again, or a response may be sent using either member function **put**, or **uvm_driver**::**rsp_port**.**write()**.

14.13.3.5 peek

```
virtual void peek( REQ& req ) = 0;
```

The member function **peek** shall return the current request item if one is in the sequencer FIFO. If no item is in the FIFO, then the call blocks until the sequencer has a new request. The following steps shall occur if the sequencer FIFO is empty:

a) Arbitrate among requesting, unlocked, 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 member function **uvm_sequence_base**::**wait_for_grant** (see Section 9.3.7.4).
c) The chosen sequence's member function **uvm_sequence_base**::**pre_do** is called (see Section 9.3.4.4).
d) The chosen sequence item is randomized.
e) The chosen sequence's member function **uvm_sequence_base**::**post_do** is called (see Section 9.3.4.7).

Once a request item has been retrieved and is in the sequencer FIFO, subsequent calls to member function **peek** returns the same item. The item stays in the FIFO until either the member function **get** or **item_done** is called.

14.13.3.6 put

```
virtual void put( const RSP& rsp ) = 0;
```

The member function **put** shall send a response back to the sequence that issued the request. Before the response is put, it shall have its sequence ID and transaction ID set to match the request. This can be done using the member function **uvm_sequence_item**::**set_id_info**.

This member function shall not block. The response is put into the sequence response queue or it is sent to the sequence response handler.

14.14 uvm_seq_item_pull_port

The class **uvm_seq_item_pull_port** shall define the port for use in sequencer-driver communication.

14.14.1 Class definition

```
namespace uvm {

  template <typename REQ, typename RSP = REQ>
  class uvm_seq_item_pull_port : public uvm_port_base< uvm_sqr_if_base<REQ, RSP> >
  {
   public:
    // Constructor
    uvm_seq_item_pull_port( const char* name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_seq_item_pull_port

} // namespace uvm
```

14.14.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.

14.14.3 Constructor

```
uvm_seq_item_pull_port( const char *name );
```

The constructor shall create a new export. The argument *name* shall define the name of the export. Otherwise, the name of the export is implementation-defined.

14.14.4 Member functions

14.14.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string “**uvm**::**uvm_seq_item_pull_port**”.

14.15 uvm_seq_item_pull_export

The class **uvm_seq_item_pull_export** shall define the export for use in sequencer-driver communication.

14.15.1 Class definition

```
namespace uvm {

  template <typename REQ, typename RSP = REQ>
  class uvm_seq_item_pull_export : public uvm_export_base< uvm_sqr_if_base<REQ, RSP> >
  {
   public:
    // Constructor
    uvm_seq_item_pull_export( const char* name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_seq_item_pull_export

} // namespace uvm
```

14.15.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.

14.15.3 Constructor

```
uvm_seq_item_pull_export( const char* name );
```

The constructor shall create a new export. The argument *name* shall define the name of the export. Otherwise, the name of the export is implementation-defined.

14.15.4 Member functions

14.15.4.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_seq_item_pull_export**".

14.16 uvm_seq_item_pull_imp

The class **uvm_seq_item_pull_imp** shall implement the interface used in sequencer-driver communication.

14.16.1 Class definition

```
namespace uvm {

  template <typename REQ = int, typename RSP = REQ, typename IMP = int>
  class uvm_seq_item_pull_imp : public uvm_export_base< uvm_sqr_if_base<REQ, RSP> >
  {
   public:
    // Constructor
    uvm_seq_item_pull_imp( const char* name );

    // Member function
    virtual const std::string get_type_name() const;

  }; // class uvm_seq_item_pull_imp

} // namespace uvm
```

14.16.2 Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. The template parameter IMP specifies the type of the component implementing the interface.

14.16.3 Member functions

14.16.3.1 get_type_name

```
virtual const std::string get_type_name() const;
```

The member function **get_type_name** shall return the string "**uvm**::**uvm_seq_item_pull_imp**".

15. Register abstraction classes

The UVM register abstraction layer defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a DUT.

The UVM register abstraction classes are not usable as-is. They only provide generic and introspection capabilities. They need to be specialized via extensions to provide an abstract view that corresponds to the actual registers and memories in a design. Due to the large number of registers in a design and the numerous small details involved in properly configuring the UVM register layer classes, this specialization is normally done by a model generator. Model generators work from a specification of the registers and memories in a design and are thus able to provide an up-to-date, correct-by-construction register model. Model generators are outside the scope of the UVM standard.

15.1 uvm_reg_block

The class **uvm_reg_block** is the base class for regisiter blocks. A register 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.

15.1.1 Class definition

```
namespace uvm {

  class uvm_reg_block : public uvm_object
  {
   public:

    // Constructor
    uvm_reg_block( const std::string& name = "",
                   int has_coverage = UVM_NO_COVERAGE );

    // Group: Initialization

    void configure( uvm_reg_block* parent = NULL,
                    const std::string& hdl_path = "" );

    virtual uvm_reg_map* create_map( const std::string& name,
                                     uvm_reg_addr_t base_addr,
                                     unsigned int n_bytes,
                                     uvm_endianness_e endian,
                                     bool byte_addressing = true );

    static bool check_data_width( unsigned int width );
    void set_default_map( uvm_reg_map* map );
    uvm_reg_map* get_default_map() const;
    virtual void lock_model();
    bool is_locked() const;

    // Group: Introspection

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_reg_block* get_parent() const;
    static void get_root_blocks( std::vector<uvm_reg_block*>& blks );

    static int find_blocks( std::string name,
                            std::vector<uvm_reg_block*>& blks,
                            uvm_reg_block* root = NULL,
                            uvm_object* accessor = NULL );

    static uvm_reg_block* find_block( const std::string& name,
                                      uvm_reg_block* root = NULL,
                                      uvm_object* accessor = NULL );

    virtual void get_blocks( std::vector<uvm_reg_block*>& blks,
                             uvm_hier_e hier = UVM_HIER ) const;
```

```
  virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;

  virtual void get_registers( std::vector<uvm_reg*>& regs,
                              uvm_hier_e hier = UVM_HIER ) const;

  virtual void get_fields( std::vector<uvm_reg_field*>& fields,
                           uvm_hier_e hier = UVM_HIER ) const;

  void get_memories( std::vector<uvm_mem*>& mems,
                     uvm_hier_e hier = UVM_HIER ) const;

  void get_virtual_registers( std::vector<uvm_vreg*>& regs,
                              uvm_hier_e hier = UVM_HIER ) const;

  void get_virtual_fields( std::vector<uvm_vreg_field*>& fields,
                           uvm_hier_e hier = UVM_HIER ) const;

  uvm_reg_block* get_block_by_name( const std::string& name ) const;
  uvm_reg_map* get_map_by_name( const std::string& name ) const;
  uvm_reg* get_reg_by_name( const std::string& name ) const;
  uvm_reg_field* get_field_by_name( const std::string& name ) const;
  uvm_mem* get_mem_by_name( const std::string& name ) const;
  uvm_vreg* get_vreg_by_name( const std::string& name ) const;
  uvm_vreg_field* get_vfield_by_name( const std::string& name ) const;

 // Group: Coverage

 protected:
  uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
  virtual void add_coverage( uvm_reg_cvr_t models );

 public:
  bool has_coverage( uvm_reg_cvr_t models ) const;
  uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
  bool get_coverage( uvm_reg_cvr_t is_on = UVM_CVR_ALL ) const;

 protected:
  virtual void sample( uvm_reg_addr_t offset,
                       bool is_read,
                       uvm_reg_map* map );
 public:
  void sample_values();

  // Group: Access

  uvm_path_e get_default_path() const;
  void reset( const std::string& kind = "HARD" );
  bool needs_update();

  virtual void update( uvm_status_e status,
                       uvm_path_e path = UVM_DEFAULT_PATH,
                       uvm_sequence_base* parent = NULL,
                       int prior = -1,
                       uvm_object* extension = NULL,
                       const std::string& fname = "",
                       int lineno = 0 );

  virtual void mirror( uvm_status_e status,
                       uvm_check_e check = UVM_NO_CHECK,
                       uvm_path_e path  = UVM_DEFAULT_PATH,
                       uvm_sequence_base* parent = NULL,
                       int prior = -1,
                       uvm_object* extension = NULL,
                       const std::string& fname = "",
                       int lineno = 0 );

  virtual void write_reg_by_name( uvm_status_e status,
                                  const std::string& name,
                                  uvm_reg_data_t data,
                                  uvm_path_e path = UVM_DEFAULT_PATH,
                                  uvm_reg_map* map = NULL,
                                  uvm_sequence_base* parent = NULL,
                                  int prior = -1,
                                  uvm_object* extension = NULL,
                                  const std::string& fname = "",
                                  int lineno = 0 );
```

```
    virtual void read_reg_by_name( uvm_status_e status,
                                   const std::string& name,
                                   uvm_reg_data_t data,
                                   uvm_path_e path = UVM_DEFAULT_PATH,
                                   uvm_reg_map* map = NULL,
                                   uvm_sequence_base* parent = NULL,
                                   int prior = -1,
                                   uvm_object* extension = NULL,
                                   const std::string& fname = "",
                                   int lineno = 0 );

    virtual void write_mem_by_name( uvm_status_e status,
                                    const std::string& name,
                                    uvm_reg_addr_t offset,
                                    uvm_reg_data_t data,
                                    uvm_path_e path = UVM_DEFAULT_PATH,
                                    uvm_reg_map* map = NULL,
                                    uvm_sequence_base* parent = NULL,
                                    int prior = -1,
                                    uvm_object* extension = NULL,
                                    const std::string& fname = "",
                                    int lineno = 0 );

    virtual void read_mem_by_name( uvm_status_e status,
                                   const std::string& name,
                                   uvm_reg_addr_t offset,
                                   uvm_reg_data_t data,
                                   uvm_path_e path = UVM_DEFAULT_PATH,
                                   uvm_reg_map* map = NULL,
                                   uvm_sequence_base* parent = NULL,
                                   int prior = -1,
                                   uvm_object* extension = NULL,
                                   const std::string& fname = "",
                                   int lineno = 0 );

    // Group: Backdoor

    uvm_reg_backdoor* get_backdoor( bool inherited = true ) const;

    void set_backdoor( uvm_reg_backdoor* bkdr,
                       const std::string& fname = "",
                       int lineno = 0);

    void clear_hdl_path( const std::string& kind = "RTL" );
    void add_hdl_path( const std::string& path, const std::string& kind = "RTL" );
    bool has_hdl_path( const std::string& kind = "" ) const;
    void get_hdl_path( std::vector<std::string>& paths, const std::string& kind = "" ) const;

    void get_full_hdl_path( std::vector<std::string>& paths,
                            std::string kind = "",
                            const std::string& separator = "." ) const;

    void set_default_hdl_path( const std::string& kind );
    std::string get_default_hdl_path() const;
    void set_hdl_path_root( const std::string& path, std::string kind = "RTL" );
    bool is_hdl_path_root( std::string kind = "" ) const;

    // Data members

    uvm_reg_map* default_map;
    uvm_path_e default_path;

  }; // class uvm_reg_block

} // namespace uvm
```

15.1.2 Constructor

```
uvm_reg_block( const std::string& name = "",
               int has_coverage = UVM_NO_COVERAGE );
```

The constructor shall create an instance of a block abstraction class with the specified name. The argument *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.

15.1.3 Initialization

15.1.3.1 configure

```
void configure( uvm_reg_block* parent = NULL,
                const std::string& hdl_path = "" );
```

The member function **configure** shall specify the parent block of this block. A block without parent is a root block. If the block file corresponds to a hierarchical RTL structure, its contribution to the HDL path is specified as the argument *hdl_path*. Otherwise, the block does not correspond to a hierarchical RTL structure (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers or memories.

15.1.3.2 create_map

```
virtual uvm_reg_map* create_map( const std::string& name,
                                 uvm_reg_addr_t base_addr,
                                 unsigned int n_bytes,
                                 uvm_endianness_e endian,
                                 bool byte_addressing = true );
```

The member function **create_map** shall create an address map with the specified name, then configures it with the following properties:

— *base_addr*: the base address for the map. All registers, memories, and sub-blocks within the map shall be at offsets to this address.

— *n_bytes*: the byte-width of the bus on which this map is used

— *endian*: the endian format. See **uvm_endianness_e** (Section 15.16.2.4) for possible values.

— *byte_addressing*: specifies whether consecutive addresses refer are 1 byte apart (true) or n_bytes apart (false). Default value is true

15.1.3.3 check_data_width

```
static bool check_data_width( unsigned int width );
```

The member function **check_data_width** shall check that the specified data width (in bits) is less than or equal to the value of **UVM_REG_DATA_WIDTH**.

NOTE—This member function is designed to be called by a static initializer.

15.1.3.4 set_default_map

```
void set_default_map( uvm_reg_map* map );
```

The member function **set_default_map** shall define the specified address map as the default_map for this block.

15.1.3.5 get_default_map

```
uvm_reg_map* get_default_map() const;
```

The member function **get_default_map** shall return the specified address map for this block.

15.1.3.6 lock_model

```
virtual void lock_model();
```

The member function **lock_model** shall recursively lock an entire register model and build the address maps to enable the member functions **uvm_reg_map::get_reg_by_offset** and **uvm_reg_map::get_mem_by_offset**. Once locked, no further structural changes, such as adding registers or memories, can be made. It is not possible to unlock a model.

15.1.3.7 is_locked

```
bool is_locked() const;
```

The member function **is_locked** shall return true if the model is locked, otherwise it shall return false.

15.1.4 Introspection

15.1.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this block.

15.1.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this block. The base of the hierarchical name is the root block.

15.1.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent block. If this a top-level block, it shall return NULL.

15.1.4.4 get_root_blocks

```
static void get_root_blocks( std::vector<uvm_reg_block*>& blks );
```

The member function **get_root_blocks** shall return an array of all root blocks.

15.1.4.5 find_blocks

```
static int find_blocks( std::string name,
                        std::vector<uvm_reg_block*>& blks,
```

```
                         uvm_reg_block* root = NULL,
                         uvm_object* accessor = NULL );
```

The member function **find_blocks** shall search for 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. The member function returns the number of blocks found.

15.1.4.6 find_block

```
static uvm_reg_block* find_block( const std::string& name,
                                  uvm_reg_block* root = NULL,
                                  uvm_object* accessor = NULL );
```

The member function **find_block** shall return 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. The member function returns the first block found or null otherwise. A warning is issued if more than one block is found.

15.1.4.7 get_blocks

```
virtual void get_blocks( std::vector<uvm_reg_block*>& blks,
                         uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_blocks** shall return the blocks instantiated in this block. If argument *hier* is set to true, it recursively includes any subblock.

15.1.4.8 get_maps

```
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
```

The member function **get_maps** shall return the address maps instantiated in this block.

15.1.4.9 get_registers

```
virtual void get_registers( std::vector<uvm_reg*>& regs,
                            uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_registers** shall return the registers instantiated in this block. If argument *hier* is set to true, it recursively includes the registers in the sub-blocks.

Note that registers may be located in different and/or multiple address maps. To get the registers in a specific address map, use member function **uvm_reg_map::get_registers** (see Section 15.2.4.13).

15.1.4.10 get_fields

```
virtual void get_fields( std::vector<uvm_reg_field*>& fields,
                         uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_fields** shall return the fields in the registers instantiated in this block. If argument *hier* is set to true, it recursively includes the fields of the registers in the sub-blocks.

15.1.4.11 get_memories

```
void get_memories( std::vector<uvm_mem*>& mems,
```

```
                    uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_memories** shall return the memories instantiated in this block. If argument *hier* is set to true, it recursively includes the memories in the sub-blocks.

Note that memories may be located in different and/or multiple address maps. To get the memories in a specific address map, use member function **uvm_reg_map::get_memories** (see Section 15.2.4.15).

15.1.4.12 get_virtual_registers

```
void get_virtual_registers( std::vector<uvm_vreg*>& regs,
                            uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_virtual_registers** shall return the virtual registers instantiated in this block. If argument *hier* is set to true, it recursively includes the virtual registers in the sub-blocks.

15.1.4.13 get_virtual_fields

```
void get_virtual_fields( std::vector<uvm_vreg_field*>& fields,
                         uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_virtual_fields** shall return the virtual fields from the virtual registers instantiated in this block. If argument *hier* is set to true, it recursively includes the virtual fields in the virtual registers in the sub-blocks.

15.1.4.14 get_block_by_name

```
uvm_reg_block* get_block_by_name( const std::string& name ) const;
```

The member function **get_block_by_name** shall search for the sub-block with the specified simple name. The argument *name* is the simple name of the block, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.15 get_map_by_name

```
uvm_reg_map* get_map_by_name( const std::string& name ) const;
```

The member function **get_map_by_name** shall search for an address map with the specified simple name. The argument *name* is the simple name of the address map, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.16 get_reg_by_name

```
uvm_reg* get_reg_by_name( const std::string& name ) const;
```

The member function **get_reg_by_name** shall search for a register with the specified simple name. The argument *name* is the simple name of the register, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.17 get_field_by_name

```
uvm_reg_field* get_field_by_name( const std::string& name ) const;
```

The member function **get_field_by_name** shall search for the field with the specified simple name. The argument *name* is the simple name of the field, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.18 get_mem_by_name

```
uvm_mem* get_mem_by_name( const std::string& name ) const;
```

The member function **get_mem_by_name** shall search for the memory with the specified simple name. The argument *name* is the simple name of the memory, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.19 get_vreg_by_name

```
uvm_vreg* get_vreg_by_name( const std::string& name ) const;
```

The member function **get_vreg_by_name** shall search for the virtual register with the specified simple name. The argument *name* is the simple name of the virtual register, not the hierarchical name. 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, the member function shall return NULL.

15.1.4.20 get_vfield_by_name

```
uvm_vreg_field* get_vfield_by_name( const std::string& name ) const;
```

The member function **get_vfield_by_name** shall search for the virtual field with the specified simple name. The argument *name* is the simple name of the virtual field, not the hierarchical name. 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, the member function shall return NULL.

15.1.5 Coverage

NOTE—Functional coverage is not yet available in UVM-SystemC.

15.1.5.1 build_coverage

```
protected: uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
```

The member function **build_coverage** shall check which 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**. Models are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e**. The member function returns the sum of all coverage models to be built in the block model.

15.1.5.2 add_coverage

```
protected: virtual void add_coverage( uvm_reg_cvr_t models );
```

The member function **add_coverage** shall specify that additional coverage models are available. Add 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**. This member function shall be called only in the constructor of subsequently derived classes.

15.1.5.3 has_coverage

```
bool has_coverage( uvm_reg_cvr_t models ) const;
```

The member function **has_coverage** shall return 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 **uvm_coverage_model_e**.

15.1.5.4 set_coverage

```
uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
```

The member function **set_coverage** shall specify the collection of functional coverage measurements 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. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. The member function returns the sum of all functional coverage models whose measurements were previously on. This member function can only control the measurement of functional coverage models that are present in the various abstraction classes, then enabled during construction. See Section 15.1.5.3 to identify the available functional coverage models.

15.1.5.5 get_coverage

```
virtual bool get_coverage( uvm_reg_cvr_t is_on = UVM_CVR_ALL ) const;
```

The member function **get_coverage** shall 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 Section 15.1.5.4 for more details.

15.1.5.6 sample

```
protected: virtual void sample( uvm_reg_addr_t offset,
                                bool is_read,
                                uvm_reg_map* map );
```

The member function **sample** shall specify the functional coverage measurement method.

This member function is invoked by the block abstraction class whenever an address within one of its address map is successfully read or written. The specified offset is the offset within the block, not an absolute address. This member function may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

15.1.5.7 sample_values

```
void sample_values();
```

The member function **sample_values** shall specify the functional coverage measurement method for field values.

This member function is invoked by the user or by the member function **uvm_reg_block::sample_values** of the parent block to trigger the sampling of the current field values in the block-level functional coverage model. It recursively invokes the member functions **uvm_reg_block::sample_values** and **uvm_reg::sample_values** in the blocks and registers in this block. This member function may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model. If this member function is extended, it shall call the member function **sample_values** of its base class.

15.1.6 Access

15.1.6.1 get_default_path

```
uvm_path_e get_default_path() const;
```

The member function **get_default_path** shall return the default access path for this block.

15.1.6.2 reset

```
void reset( const std::string& kind = "HARD" );
```

The member function **reset** shall set the mirror value of all registers in the block and sub-blocks to the reset value corresponding to the specified reset event (see also Section 15.5.5.4). This member function does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror.

15.1.6.3 needs_update

```
bool needs_update();
```

The member function **needs_update** shall check 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 member function **uvm_reg::set**, the mirror and state of the registers are outdated. The corresponding registers in the DUT need to be updated. This member function 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 Section 15.1.6.4.

15.1.6.4 update

```
virtual void update( uvm_status_e status,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **update** shall perform a batch update of the register. Using the minimum number of write operations, updates the registers in the design to match the mirrored values in this block and sub-blocks. The update can be performed using the physical interfaces (front-door access) or back-door accesses. This member function performs the reverse operation of **uvm_reg_block::mirror** (see Section 15.1.6.5).

15.1.6.5 mirror

```
virtual void mirror( uvm_status_e status,
                     uvm_check_e check = UVM_NO_CHECK,
                     uvm_path_e path  = UVM_DEFAULT_PATH,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **mirror** shall perform an update the mirrored values. Read all of the registers in this block and sub-blocks and update their mirror values to match their corresponding values in the design. The mirroring can be performed using the physical interfaces (front-door access) or back-door accesses. If the check argument is specified as **UVM_CHECK**, an error message is issued if the current mirrored value does not match the actual value in the design. This member function performs the reverse operation of **uvm_reg_block::update** (see Section 15.1.6.4).

15.1.6.6 write_reg_by_name

```
virtual void write_reg_by_name( uvm_status_e status,
                                const std::string& name,
                                uvm_reg_data_t data,
                                uvm_path_e path = UVM_DEFAULT_PATH,
                                uvm_reg_map* map = NULL,
                                uvm_sequence_base* parent = NULL,
                                int prior = -1,
                                uvm_object* extension = NULL,
                                const std::string& fname = "",
                                int lineno = 0 );
```

The member function **write_reg_by_name** shall write the named register. Equivalent to **get_reg_by_name** (see Section 15.1.4.16) followed by **uvm_reg::write** (see Section 15.4.5.9).

15.1.6.7 read_reg_by_name

```
virtual void read_reg_by_name( uvm_status_e status,
                               const std::string& name,
                               uvm_reg_data_t data,
                               uvm_path_e path = UVM_DEFAULT_PATH,
                               uvm_reg_map* map = NULL,
                               uvm_sequence_base* parent = NULL,
                               int prior = -1,
                               uvm_object* extension = NULL,
                               const std::string& fname = "",
                               int lineno = 0 );
```

The member function **read_reg_by_name** shall Read the named register. Equivalent to **get_reg_by_name** (see Section 15.1.4.16) followed by **uvm_reg::read** (see Section 15.4.5.10).

15.1.6.8 write_mem_by_name

```
virtual void write_mem_by_name( uvm_status_e status,
                                const std::string& name,
                                uvm_reg_addr_t offset,
                                uvm_reg_data_t data,
                                uvm_path_e path = UVM_DEFAULT_PATH,
                                uvm_reg_map* map = NULL,
                                uvm_sequence_base* parent = NULL,
                                int prior = -1,
                                uvm_object* extension = NULL,
                                const std::string& fname = "",
                                int lineno = 0 );
```

The member function **write_mem_by_name** shall write the named memory. Equivalent to **get_mem_by_name** (see Section 15.1.4.18) followed by **uvm_mem::write** (see Section 15.6.5.1).

15.1.6.9 read_mem_by_name

```
virtual void read_mem_by_name( uvm_status_e status,
                               const std::string& name,
                               uvm_reg_addr_t offset,
                               uvm_reg_data_t data,
                               uvm_path_e path = UVM_DEFAULT_PATH,
                               uvm_reg_map* map = NULL,
                               uvm_sequence_base* parent = NULL,
                               int prior = -1,
                               uvm_object* extension = NULL,
                               const std::string& fname = "",
                               int lineno = 0 );
```

The member function **read_mem_by_name** shall eead the named memory. Equivalent to **get_mem_by_name** (see Section 15.1.4.18) followed by **uvm_mem::read** (see Section 15.6.5.2).

15.1.7 Backdoor

NOTE—Backdoor access is not yet available in UVM-SystemC.

15.1.7.1 get_backdoor

```
uvm_reg_backdoor* get_backdoor( bool inherited = true ) const;
```

The member function **get_backdoor** shall return the user-defined backdoor for all registers in this block, unless overridden by a backdoor set in a lower-level block or in the register itself.

If no argument is given or argument *inherited* is set to true, the member function returns the backdoor of the parent block if none have been specified for this block.

15.1.7.2 set_backdoor

```
void set_backdoor( uvm_reg_backdoor* bkdr,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **set_backdoor** shall specify the user-defined backdoor for all registers in this block.

It defines the backdoor mechanism for all registers instantiated in this block and subblocks, unless overridden by a definition in a lower-level block or register.

15.1.7.3 clear_hdl_path

```
void clear_hdl_path( const std::string& kind = "RTL" );
```

The member function **clear_hdl_path** shall remove any previously specified HDL path to the block instance for the specified design abstraction.

15.1.7.4 add_hdl_path

```
void add_hdl_path( const std::string& path, const std::string& kind = "RTL" );
```

The member function **add_hdl_path** shall add the specified HDL path to the block instance for the specified design abstraction. This member function may be called more than once for the same design abstraction if the block is physically duplicated in the design abstraction.

15.1.7.5 has_hdl_path

```
bool has_hdl_path( const std::string& kind = "" ) const;
```

The member function **has_hdl_path** shall return true if the block 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 this block or the nearest block ancestor with a specified default design abstraction.

15.1.7.6 get_hdl_path

```
void get_hdl_path( std::vector<std::string>& paths, const std::string& kind = "" ) const;
```

The member function **get_hdl_path** shall return the HDL path(s) defined for the specified design abstraction in the block instance. It returns only the component of the HDL paths that corresponds to the block, not a full hierarchical path. If no design abstraction is specified, the default design abstraction for this block is used.

15.1.7.7 get_full_hdl_path

```
void get_full_hdl_path( std::vector<std::string>& paths,
                        std::string kind = "",
                        const std::string& separator = "." ) const;
```

The member function **get_full_hdl_path** shall return the full hierarchical HDL path(s) defined for the specified design abstraction in the block instance. There may be more than one path returned even if only one path was defined for the block instance, if any of the parent components have more than one path defined for the same design abstraction. If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

15.1.7.8 set_default_hdl_path

```
void set_default_hdl_path( const std::string& kind );
```

The member function **set_default_hdl_path** shall specify the default design abstraction for this block instance.

15.1.7.9 get_default_hdl_path

```
std::string get_default_hdl_path() const;
```

The member function **get_default_hdl_path** shall return the default design abstraction for this block instance. If a default design abstraction has not been explicitly set for this block instance, it returns the default design abstraction for the nearest block ancestor. It returns an empty string if no default design abstraction has been specified.

15.1.7.10 set_hdl_path_root

```
void set_hdl_path_root( const std::string& path, std::string kind = "RTL" );
```

The member function **set_hdl_path_root** shall specify the specified path as the absolute HDL path to the block instance for the specified design abstraction. This absolute root path is prepended to all hierarchical paths under this block. The HDL path of any ancestor block is ignored. This member function overrides any incremental path for the same design abstraction specified using **add_hdl_path**.

15.1.7.11 is_hdl_path_root

```
bool is_hdl_path_root( std::string kind = "" ) const;
```

The member function **is_hdl_path_root** shall return true if an absolute HDL path to the block instance for the specified design abstraction has been defined. If no design abstraction is specified, the default design abstraction for this block is used.

15.1.8 Data members (variables)

15.1.8.1 default_map

```
uvm_reg_map* default_map;
```

The data member **default_map** shall define the default address map for this block, to be used when no address map is specified for a register operation and that register is accessible from more than one address map.

It is also the implicit address map for a block with a single, unnamed address map because it has only one physical interface.

15.1.8.2 default_path

```
uvm_path_e default_path;
```

The data member **default_path** shall define the default access path for the registers and memories in this block.

15.2 uvm_reg_map

This class **uvm_reg_map** shall represent 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.

15.2.1 Class definition

```
namespace uvm {

  class uvm_reg_map : public uvm_object
  {
   public:

    // Constructor
    explicit uvm_reg_map( const std::string& name = "uvm_reg_map" );

    // Group: Initialization

    void configure( uvm_reg_block* parent,
                    uvm_reg_addr_t base_addr,
                    unsigned int n_bytes,
                    uvm_endianness_e endian,
                    bool byte_addressing = true );

    virtual void add_reg( uvm_reg* rg,
```

```
                        uvm_reg_addr_t offset,
                        const std::string& rights = "RW",
                        bool unmapped = false,
                        uvm_reg_frontdoor* frontdoor = NULL );

    virtual void add_mem( uvm_mem* mem,
                          uvm_reg_addr_t offset,
                          const std::string& rights = "RW",
                          bool unmapped = false,
                          uvm_reg_frontdoor* frontdoor = NULL );

    virtual void add_submap( uvm_reg_map* child_map,
                             uvm_reg_addr_t offset );

    virtual void set_sequencer( uvm_sequencer_base* sequencer,
                                uvm_reg_adapter* adapter = NULL );

    virtual void set_submap_offset( uvm_reg_map* submap,
                                    uvm_reg_addr_t offset);

    virtual uvm_reg_addr_t get_submap_offset( const uvm_reg_map* submap ) const;
    virtual void set_base_addr( uvm_reg_addr_t offset);
    virtual void reset( const std::string& kind = "SOFT" );

    // Group: Introspection

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_reg_map* get_root_map() const;
    virtual uvm_reg_block* get_parent() const;
    virtual uvm_reg_map* get_parent_map() const;
    virtual uvm_reg_addr_t get_base_addr( uvm_hier_e hier = UVM_HIER) const;
    virtual unsigned int get_n_bytes( uvm_hier_e hier = UVM_HIER ) const;
    virtual unsigned int get_addr_unit_bytes() const;
    virtual uvm_endianness_e get_endian( uvm_hier_e hier = UVM_HIER ) const;
    virtual uvm_sequencer_base* get_sequencer( uvm_hier_e hier = UVM_HIER ) const;
    virtual uvm_reg_adapter* get_adapter( uvm_hier_e hier = UVM_HIER ) const;

    virtual void get_submaps( std::vector&<uvm_reg_map*>& maps,
                              uvm_hier_e hier = UVM_HIER ) const;

    virtual void get_registers( std::vector&<uvm_reg*>& regs,
                                uvm_hier_e hier = UVM_HIER ) const;

    virtual void get_fields( std::vector&<uvm_reg_field*>& fields,
                             uvm_hier_e hier = UVM_HIER ) const;

    virtual void get_memories( std::vector&<uvm_mem*>& mems,
                               uvm_hier_e hier = UVM_HIER ) const;

    virtual void get_virtual_registers( std::vector&<uvm_vreg*>& vregs,
                                        uvm_hier_e hier = UVM_HIER) const;

    virtual void get_virtual_fields( std::vector&<uvm_vreg_field*>& fields,
                                     uvm_hier_e hier = UVM_HIER) const;

    virtual int get_physical_addresses( uvm_reg_addr_t base_addr,
                                        uvm_reg_addr_t mem_offset,
                                        unsigned int n_bytes,
                                        std::vector&<uvm_reg_addr_t>& addr ) const;

    virtual uvm_reg* get_reg_by_offset( uvm_reg_addr_t offset,
                                        bool read = true ) const;

    virtual uvm_mem* get_mem_by_offset( uvm_reg_addr_t offset ) const;

    // Group: Bus Access

    void set_auto_predict( bool on = true );
    bool get_auto_predict() const;
    void set_check_on_read( bool on = true );
    bool get_check_on_read() const;

    virtual void do_bus_write( uvm_reg_item* rw,
                               uvm_sequencer_base* sequencer,
                               uvm_reg_adapter* adapter );
```

```
    virtual void do_bus_read( uvm_reg_item* rw,
                              uvm_sequencer_base* sequencer,
                              uvm_reg_adapter* adapter );

    virtual void do_write( uvm_reg_item* rw );
    virtual void do_read( uvm_reg_item* rw );

    // Group: Backdoor

    static uvm_reg_map* backdoor();


  }; // class uvm_reg_map

} // namespace uvm
```

15.2.2 Constructor

```
explicit uvm_reg_map( const std::string& name = "uvm_reg_map" );
```

The constructor shall create an instance of an address map with the specified name.

15.2.3 Initialization

15.2.3.1 configure

```
void configure( uvm_reg_block* parent,
                uvm_reg_addr_t base_addr,
                unsigned int n_bytes,
                uvm_endianness_e endian,
                bool byte_addressing = true );
```

The member function **configure** shall configure this map with the following properties:

— *parent*: the block in which this map is created and applied.

— *base_addr*: the base address for this map. All registers, memories, and sub-blocks shall be at offsets to this address.

— *n_bytes*: the byte-width of the bus on which this map is used.

— *endian*: the endian format, see Section 15.16.2.4.

— *byte_addressing*: specifies whether the address increment is on a per-byte basis. For example, consecutive memory locations with *n_bytes*=4 (32-bit bus) are 4 apart: 0, 4, 8, and so on. Default value is true.

15.2.3.2 add_reg

```
virtual void add_reg( uvm_reg* rg,
                      uvm_reg_addr_t offset,
                      const std::string& rights = "RW",
                      bool unmapped = false,
                      uvm_reg_frontdoor* frontdoor = NULL );
```

The member function **add_reg** shall add the specified register instance *rg* to this address map.

The register is located at the specified address offset from this maps 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 **uvm_reg_field::configure**, Section 15.5.3.1) and the register’s rights in the map being used to access the field.

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 set to true, the register does not occupy any physical addresses and the base address is ignored. Unmapped registers require a user-defined frontdoor to be specified.

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.

15.2.3.3 add_mem

```
virtual void add_mem( uvm_mem* mem,
                      uvm_reg_addr_t offset,
                      const std::string& rights = "RW",
                      bool unmapped = false,
                      uvm_reg_frontdoor* frontdoor = NULL );
```

The member function **add_mem** shall add the specified memory instance to this address map. The memory is located at the specified base address and has the specified access rights ("RW", "RO" or "WO"). 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 argument *unmapped* is set to true, the memory does not occupy any physical addresses and the base address is ignored. Unmapped memories require a user-defined frontdoor to be specified.

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.

15.2.3.4 add_submap

```
virtual void add_submap( uvm_reg_map* child_map,
                         uvm_reg_addr_t offset );
```

The member function **add_submap** shall add 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.

15.2.3.5 set_sequencer

```
virtual void set_sequencer( uvm_sequencer_base* sequencer,
                            uvm_reg_adapter* adapter = NULL );
```

The member function **set_sequencer** shall set the sequencer and adapter associated with this map. This member function shall be called before starting any sequences based on **uvm_reg_sequence**.

15.2.3.6 set_submap_offset

```
virtual void set_submap_offset( uvm_reg_map* submap,
                                uvm_reg_addr_t offset);
```

The member function **set_submap_offset** shall set the offset of the given *submap* to *offset*.

15.2.3.7 get_submap_offset

```
virtual uvm_reg_addr_t get_submap_offset( const uvm_reg_map* submap ) const;
```

The member function **get_submap_offset** shall return the offset of the given *submap*.

15.2.3.8 set_base_addr

```
virtual void set_base_addr( uvm_reg_addr_t offset);
```

The member function **set_base_addr** shall set the base address of this map.

15.2.3.9 reset

```
virtual void reset( const std::string& kind = "SOFT" );
```

The member function **reset** shall set 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 also Section 15.5.5.4). 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 member functions **reset**, the default reset event for this member functions is “SOFT”.

15.2.4 Introspection

15.2.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this address map.

15.2.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this address map. The base of the hierarchical name is the root block.

15.2.4.3 get_root_map

```
virtual uvm_reg_map* get_root_map() const;
```

The member function **get_root_map** shall return the top-most address map where this address map is instantiated. It corresponds to the externally-visible address map that can be accessed by the verification environment.

15.2.4.4 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the block that is the parent of this address map.

15.2.4.5 get_parent_map

```
virtual uvm_reg_map* get_parent_map() const;
```

The member function **get_parent_map** shall return the address map in which this address map is mapped. The member function returns NULL if this is a top-level address map.

15.2.4.6 get_base_addr

```
virtual uvm_reg_addr_t get_base_addr( uvm_hier_e hier = UVM_HIER) const;
```

The member function **get_base_addr** shall return the base offset address for this map. If this map is the root map, the base address is that set with the argument *base_addr* to **uvm_reg_block::create_map**. If this map is a submap of a higher-level map, the base address is offset given this submap by the parent map. See Section 15.2.3.6.

15.2.4.7 get_n_bytes

```
virtual unsigned int get_n_bytes( uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_n_bytes** shall return the width in bytes of the bus associated with this map. If the argument *hier* is **UVM_HIER**, it returns 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 shall be limited to this bus width.

15.2.4.8 get_addr_unit_bytes

```
virtual unsigned int get_addr_unit_bytes() const;
```

The member function **get_addr_unit_bytes** shall return the number of bytes in the smallest addressable unit in the map. It shall returns 1 if the address map was configured using byte-level addressing, otherwise it shall return **get_n_bytes** (see Section 15.2.4.7).

15.2.4.9 get_endian

```
virtual uvm_endianness_e get_endian( uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_endian** shall return the endianness of the bus associated with this map (see Section 15.16.2.4). If argument *hier* is set to **UVM_HIER**, it shall return the system-level endianness.

15.2.4.10 get_sequencer

```
virtual uvm_sequencer_base* get_sequencer( uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_sequencer** shall return the sequencer for the bus associated with this map. If argument *hier* is set to **UVM_HIER**, it shall get the sequencer for the bus at the system-level. (See Section 15.2.3.5).

15.2.4.11 get_adapter

```
virtual uvm_reg_adapter* get_adapter( uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_adapter** shall return the bus adapter for the bus associated with this map. If argument *hier* is set to **UVM_HIER**, it shall get the adapter for the bus used at the system-level. (See Section 15.2.3.5).

15.2.4.12 get_submaps

```
virtual void get_submaps( std::vector&<uvm_reg_map*>& maps,
                          uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_submaps** shall return the address maps instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the address maps in the sub-maps.

15.2.4.13 get_registers

```
virtual void get_registers( std::vector&<uvm_reg*>& regs,
                            uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_registers** shall return the registers instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the registers in the sub-maps.

15.2.4.14 get_fields

```
virtual void get_fields( std::vector&<uvm_reg_field*>& fields,
                         uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_fields** shall return the fields in the registers instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the fields of the registers in the sub-maps.

15.2.4.15 get_memories

```
virtual void get_memories( std::vector&<uvm_mem*>& mems,
                           uvm_hier_e hier = UVM_HIER ) const;
```

The member function **get_memories** shall return the memories instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the memories in the sub-maps.

15.2.4.16 get_virtual_registers

```
virtual void get_virtual_registers( std::vector&<uvm_vreg*>& vregs,
                                    uvm_hier_e hier = UVM_HIER) const;
```

The member function **get_virtual_registers** shall return the virtual registers instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the virtual registers in the sub-maps.

15.2.4.17 get_virtual_fields

```
virtual void get_virtual_fields( std::vector&<uvm_vreg_field*>& fields,
                                 uvm_hier_e hier = UVM_HIER) const;
```

The member function **get_virtual_fields** shall return the virtual fields from the virtual registers instantiated in this address map. If argument *hier* is set to **UVM_HIER**, it recursively includes the virtual fields in the virtual registers in the sub-maps.

15.2.4.18 get_physical_addresses

```
virtual int get_physical_addresses( uvm_reg_addr_t base_addr,
                                    uvm_reg_addr_t mem_offset,
                                    unsigned int n_bytes,
                                    std::vector&<uvm_reg_addr_t>& addr ) const;
```

The member function **get_physical_addresses** shall translate a local address into external addresses.

It shall identify 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. It returns the number of bytes of valid data in each access.

Argument *addr* shall return a list of address in little endian order, with the granularity of the toplevel address map.

A register is specified using a base address with mem_offset as 0. A location within a memory is specified using the base address of the memory and the index of the location within that memory.

15.2.4.19 get_reg_by_offset

```
virtual uvm_reg* get_reg_by_offset( uvm_reg_addr_t offset,
                                    bool read = true ) const;
```

The member function **get_reg_by_offset** shall return the register mapped at the given *offset*. It shall identify the register located at the specified offset within this address map for the specified type of access. The member function shall return NULL if no such register is found.

The model needs to be locked using member function **uvm_reg_block::lock_model** to enable this functionality (see Section 15.1.3.6).

15.2.4.20 get_mem_by_offset

```
virtual uvm_mem* get_mem_by_offset( uvm_reg_addr_t offset ) const;
```

The member function **get_mem_by_offset** shall return the memory mapped at the given *offset*. It shall identify the memory located at the specified offset within this address map. The offset may refer to any memory location in that memory. The member function shall return NULL if no such memory is found.

The model needs to be locked using member function **uvm_reg_block::lock_model** to enable this functionality (see Section 15.1.3.6).

15.2.5 Bus access

15.2.5.1 set_auto_predict

```
void set_auto_predict( bool on = true );
```

The member function **set_auto_predict** shall specify the auto-predict mode for this map.

When the argument *on* is set to true, the register model shall automatically update 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 Section 15.4.5.9) or **uvm_reg::read** (see Section 15.4.5.10) operation returns, the register's member function **uvm_reg::predict** (see Section 15.4.5.15) is called to update the mirrored value in the register.

When the argument *on* is set to false, bus reads and writes via this map do not automatically update the mirror. For real-time updates to the mirror in this mode, an application shall connect a **uvm_reg_predictor** (see Section 16.5) instance to the bus monitor. The predictor takes observed bus transactions from the bus monitor, looks up the associated uvm_reg register given the address, then calls that register's member function **uvm_reg::predict**. While more complex, this mode shall capture all register read/write activity, including that not directly descendant from calls to **uvm_reg::write** and **uvm_reg::read**.

By default, auto-prediction is turned off.

15.2.5.2 get_auto_predict

```
bool get_auto_predict() const;
```

The member function **get_auto_predict** shall return the auto-predict mode setting for this map.

15.2.5.3 set_check_on_read

```
void set_check_on_read( bool on = true );
```

The member function **set_check_on_read** shall specify the check-on-read mode for his map and all of its submaps.

When the argument *on* is set to true, the register model shall automatically check 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 member functions **uvm_reg::read** (see Section 15.4.5.10) and **uvm_reg::mirror**(UVM_CHECK) (see Section 15.4.5.14). This mode is useful when the register model is used passively.

When the argument *on* is set to false, 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.

By default, auto-prediction is turned off.

15.2.5.4 get_check_on_read

```
bool get_check_on_read() const;
```

The member function **get_check_on_read** shall return the check-on-read mode setting for this map.

15.2.5.5 do_bus_write

```
virtual void do_bus_write( uvm_reg_item* rw,
                           uvm_sequencer_base* sequencer,
                           uvm_reg_adapter* adapter );
```

The member function **do_bus_write** shall perform a bus write operation.

15.2.5.6 do_bus_read

```
virtual void do_bus_read( uvm_reg_item* rw,
                          uvm_sequencer_base* sequencer,
                          uvm_reg_adapter* adapter );
```

The member function **do_bus_read** shall perform a bus read operation.

15.2.5.7 do_write

```
virtual void do_write( uvm_reg_item* rw );
```

The member function **do_write** shall perform a write operation.

15.2.5.8 do_read

```
virtual void do_read( uvm_reg_item* rw );
```

The member function **do_read** shall perform a read operation.

15.2.6 Backdoor

NOTE—Backdoor access is not yet available in UVM-SystemC.

15.2.6.1 backdoor

```
static uvm_reg_map* backdoor();
```

The member function **backdoor** shall return the backdoor pseudo-map singleton. This pseudo-map is used to specify or configure the backdoor instead of a real address map.

15.3 uvm_reg_file

The class **uvm_reg_file** defines the abstraction base class for a register file. A register file is a collection of register files and registers used to create regular repeated structures.

15.3.1 Class definition

```
namespace uvm {

  class uvm_reg_file : public uvm_object
  {
   public:

    // Constructor
    explicit uvm_reg_file( const std::string& name = "" );

    // Group: Initialization

    void configure( uvm_reg_block* blk_parent,
                    uvm_reg_file* regfile_parent,
                    const std::string& hdl_path = "" );

    // Group: Introspection

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_reg_block* get_parent() const;
```

```
    virtual uvm_reg_file* get_regfile() const;

    // Group: Backdoor

    void clear_hdl_path( const std::string& kind = "RTL" );
    void add_hdl_path( const std::string& path, const std::string& kind = "RTL" );
    bool has_hdl_path( const std::string& kind = "" ) const;
    void get_hdl_path( std::vector<std::string>& paths, const std::string& kind = "" ) const;

    void get_full_hdl_path( std::vector<std::string>& paths,
                            const std::string& kind = "",
                            const std::string& separator = "." ) const;

    void set_default_hdl_path( const std::string& kind );
    std::string get_default_hdl_path() const;

  }; // class uvm_reg_file

} // namespace uvm
```

15.3.2 Constructor

```
uvm_reg_block( const std::string& name = "",
               int has_coverage = UVM_NO_COVERAGE );
```

The constructor shall create an instance of a register file abstraction class with the specified name.

15.3.3 Initialization

15.3.3.1 configure

```
void configure( uvm_reg_block* blk_parent,
                uvm_reg_file* regfile_parent,
                const std::string& hdl_path = "" );
```

The member function **configure** shall specify 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 (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers.

15.3.4 Introspection

15.3.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this register file.

15.3.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this register file. The base of the hierarchical name is the root block.

15.3.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent block.

15.3.4.4 get_regfile

```
virtual uvm_reg_file* get_regfile() const;
```

The member function **get_regfile** shall return the parent register file. It returns NULL if this register file is instantiated in a block.

15.3.5 Backdoor

NOTE—Backdoor access is not yet available in UVM-SystemC.

15.3.5.1 clear_hdl_path

```
void clear_hdl_path( const std::string& kind = "RTL" );
```

The member function **clear_hdl_path** shall remove any previously specified HDL path to the register file instance for the specified design abstraction.

15.3.5.2 add_hdl_path

```
void add_hdl_path( const std::string& path, const std::string& kind = "RTL" );
```

The member function **add_hdl_path** shall add the specified HDL path to the register file instance for the specified design abstraction. This member function may be called more than once for the same design abstraction if the register file is physically duplicated in the design abstraction.

15.3.5.3 has_hdl_path

```
bool has_hdl_path( const std::string& kind = "" ) const;
```

The member function **has_hdl_path** shall return 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 If no design abstraction is specified, the default design abstraction for this register file is used.

15.3.5.4 get_hdl_path

```
void get_hdl_path( std::vector<std::string>& paths, const std::string& kind = "" ) const;
```

The member function **get_hdl_path** shall return 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. It returns only the component of the HDL paths that corresponds to the register file, not a full hierarchical path If no design abstraction is specified, the default design abstraction for this register file is used.

15.3.5.5 get_full_hdl_path

```
void get_full_hdl_path( std::vector<std::string>& paths,
                        const std::string& kind = "",
                        const std::string& separator = "." ) const;
```

The member function **get_full_hdl_path** shall return the full hierarchical HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block. There may be more than one path returned even if only one path was defined for the register file instance, if any of the parent components have more than one path defined for the same design abstraction. If no design abstraction is specified, the default design abstraction for each ancestor register file or block is used to get each incremental path.

15.3.5.6 set_default_hdl_path

```
void set_default_hdl_path( const std::string& kind );
```

The member function **set_default_hdl_path** shall specify the default design abstraction for this register file instance.

15.3.5.7 get_default_hdl_path

```
std::string get_default_hdl_path() const;
```

The member function **get_default_hdl_path** shall return the default design abstraction for this register file instance. If a default design abstraction has not been explicitly set for this register file instance, it returns the default design abstraction for the nearest register file or block ancestor. It returns an empty string if no default design abstraction has been specified.

15.4 uvm_reg

The class **uvm_reg** defines 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.

15.4.1 Class definition

```
namespace uvm {

  class uvm_reg : public uvm_object
  {
   public:

    uvm_reg( const std::string& name,
             unsigned int,
             int has_coverage );

    // Group: Initialization

    void configure( uvm_reg_block* blk_parent,
                    uvm_reg_file* regfile_parent = NULL,
                    const std::string& hdl_path = "" );

    void set_offset( uvm_reg_map* map,
                     uvm_reg_addr_t offset,
                     bool unmapped = false );

    // Group: Introspection
```

```
virtual const std::string get_name() const;
virtual const std::string get_full_name() const;
virtual uvm_reg_block* get_parent() const;
virtual uvm_reg_file* get_regfile() const;
virtual int get_n_maps() const;
bool is_in_map( uvm_reg_map* map ) const;
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
virtual std::string get_rights( uvm_reg_map* map = NULL ) const;
virtual unsigned int get_n_bits() const;
virtual unsigned int get_n_bytes() const;
static unsigned int get_max_size();
virtual void get_fields( std::vector<uvm_reg_field*>& fields ) const;
virtual uvm_reg_field* get_field_by_name( const std::string& name ) const;
virtual uvm_reg_addr_t get_offset( uvm_reg_map* map = NULL ) const;
virtual uvm_reg_addr_t get_address( const uvm_reg_map* map = NULL ) const;

virtual int get_addresses( std::vector<uvm_reg_addr_t>& addr,
                           const uvm_reg_map* map = NULL ) const;

 // Group: Access

virtual void set( uvm_reg_data_t value,
                  const std::string& fname = "",
                  int lineno = 0 );

virtual uvm_reg_data_t get( const std::string& fname = "",
                            int lineno = 0 ) const;

virtual uvm_reg_data_t get_mirrored_value( const std::string& fname = "",
                                           int lineno = 0 ) const;

virtual bool needs_update() const;
virtual void reset( const std::string& kind = "HARD" );
virtual uvm_reg_data_t get_reset( const std::string& kind = "HARD" ) const;

virtual bool has_reset( const std::string& kind = "HARD",
                        bool do_delete = false );

virtual void set_reset( uvm_reg_data_t value,
                        const std::string& kind = "HARD" );

virtual void write( uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );

virtual void read( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void poke( uvm_status_e& status,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void peek( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

```
virtual void update( uvm_status_e& status,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int  lineno = 0 );

virtual void mirror( uvm_status_e& status,
                     uvm_check_e check  = UVM_NO_CHECK,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );

virtual bool predict( uvm_reg_data_t value,
                      uvm_reg_byte_en_t be = -1,
                      uvm_predict_e kind = UVM_PREDICT_DIRECT,
                      uvm_path_e path = UVM_FRONTDOOR,
                      uvm_reg_map* map = NULL,
                      const std::string& fname = "",
                      int lineno = 0 );

bool is_busy() const;

// Group: Frontdoor

void set_frontdoor( uvm_reg_frontdoor* ftdr,
                    uvm_reg_map* map = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );

uvm_reg_frontdoor* get_frontdoor( uvm_reg_map* map = NULL ) const;

// Group: Backdoor

void set_backdoor( uvm_reg_backdoor* bkdr,
                   const std::string& fname = "",
                   int lineno = 0 );

uvm_reg_backdoor* get_backdoor( bool inherited = true ) const;

void clear_hdl_path( const std::string& kind = "RTL" );

void add_hdl_path( std::vector<uvm_hdl_path_slice> slices,
                   const std::string& kind = "RTL" );

void add_hdl_path_slice( const std::string& name,
                         int offset,
                         int size,
                         bool first = false,
                         const std::string& kind = "RTL" );

bool has_hdl_path( const std::string& kind = "" ) const;

void get_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                   const std::string& kind = "" ) const;

void get_hdl_path_kinds( std::vector<std::string>& kinds ) const;

void get_full_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                        const std::string& kind = "",
                        const std::string& separator = ".") const;

virtual void backdoor_read( uvm_reg_item* rw );

virtual void backdoor_write( uvm_reg_item* rw );

virtual void backdoor_watch();

// Group: Coverage

static void include_coverage( const std::string& scope,
```

```
                                  uvm_reg_cvr_t models,
                                  uvm_object* accessor = NULL );

   protected:
    uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
    virtual void add_coverage( uvm_reg_cvr_t models );

   public:
    virtual bool has_coverage( uvm_reg_cvr_t models ) const;
    virtual uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
    virtual bool get_coverage( uvm_reg_cvr_t is_on ) const;

   protected:
    virtual void sample( uvm_reg_data_t data,
                         uvm_reg_data_t byte_en,
                         bool is_read,
                         uvm_reg_map* map );

   public:
    virtual void sample_values();

    // Group: Callbacks

    virtual void pre_write( uvm_reg_item* rw );
    virtual void post_write( uvm_reg_item* rw );
    virtual void pre_read( uvm_reg_item* rw );
    virtual void post_read( uvm_reg_item* rw );

  }; // class uvm_reg

} // namespace uvm
```

15.4.2 Constructor

```
uvm_reg( const std::string& name,
         unsigned int,
         int has_coverage );
```

The constructor shall create an instance of a register abstraction class with the specified name. The argument *n_bits* specifies the total number of bits in the register. Not all bits need to be implemented. This value is usually a multiple of 8. The argument *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 Section 15.16.2.9).

15.4.3 Initialization

15.4.3.1 configure

```
void configure( uvm_reg_block* blk_parent,
                uvm_reg_file* regfile_parent = NULL,
                const std::string& hdl_path = "" );
```

The member function **configure** shall specify the parent block of this register. It may also set a parent register file for this register using argument *regfile_parent*.

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 member functions **add_hdl_path** or **add_hdl_path_slice**.

15.4.3.2 set_offset

```
void set_offset( uvm_reg_map* map,
                 uvm_reg_addr_t offset,
                 bool unmapped = false );
```

The member function **set_offset** shall specify the offset of a register within an address map. It shall use the member function **uvm_reg_map::add_reg** (see Section 15.2.3.2). This member function is used to modify that offset dynamically.

Modifying the offset of a register makes the register model diverge from the specification that was used to create it.

15.4.4 Introspection

15.4.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this register.

15.4.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this register. The base of the hierarchical name is the root block.

15.4.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent block.

15.4.4.4 get_regfile

```
virtual uvm_reg_file* get_regfile() const;
```

The member function **get_regfile** shall return the parent register file. It returns NULL if this register file is instantiated in a block.

15.4.4.5 get_n_maps

```
virtual int get_n_maps() const;
```

The member function **get_n_maps** shall return the number of address maps this register is mapped in.

15.4.4.6 is_in_map

```
bool is_in_map( uvm_reg_map* map ) const;
```

The member function **is_in_map** shall return true if this register is in the specified address map, otherwise return false.

15.4.4.7 get_maps

```
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
```

The member function **get_maps** shall return all of the address maps where this register is mapped.

15.4.4.8 get_rights

```
virtual std::string get_rights( uvm_reg_map* map = NULL ) const;
```

The member function **get_rights** shall return the accessibility ("RW, "RO", or "WO") of this register in the given map.

If no address map is specified and the register is mapped in only one address map, that address map is used. If 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 Section 15.5.3.1) 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 is issued and "RW" is returned.

15.4.4.9 get_n_bits

```
virtual unsigned int get_n_bits() const;
```

The member function **get_n_bits** shall return the width, in bits, of this register.

15.4.4.10 get_n_bytes

```
virtual unsigned int get_n_bytes() const;
```

The member function **get_n_bytes** shall return the width, in bytes, of this register. Rounds up to next whole byte if register is not a multiple of 8.

15.4.4.11 get_max_size

```
static unsigned int get_max_size();
```

The member function **get_max_size** shall return the maximum width, in bits, of all registers.

15.4.4.12 get_fields

```
virtual void get_fields( std::vector<uvm_reg_field*>& fields ) const;
```

The member function **get_fields** shall return the fields in this register. Fields are ordered from least-significant position to most-significant position within the register.

15.4.4.13 get_field_by_name

```
virtual uvm_reg_field* get_field_by_name( const std::string& name ) const;
```

The member function **get_field_by_name** shall return the named field in this register. The member function shall find a field with the specified name in this register and returns its abstraction class. If no fields are found, it returns NULL.

15.4.4.14 get_offset

```
virtual uvm_reg_addr_t get_offset( uvm_reg_map* map = NULL ) const;
```

The member function **get_offset** shall return the offset of this register in an address *map*. If no address map is specified and the register is mapped in only one address map, that address map is used. If 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, an error message is issued.

15.4.4.15 get_address

```
virtual uvm_reg_addr_t get_address( const uvm_reg_map* map = NULL ) const;
```

The member function **get_address** shall return the base external physical address of this register if accessed through the specified address *map*.

If no address map is specified and the register is mapped in only one address map, that address map is used. If 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, an error message is issued.

15.4.4.16 get_addresses

```
virtual int get_addresses( std::vector<uvm_reg_addr_t>& addr,
                           const uvm_reg_map* map = NULL ) const;
```

The member function **get_addresses** shall identify the external physical address(es) of a memory location. It computes all of the external physical addresses that needs to be accessed to completely read or write the specified location in this memory. The addressed are specified in little endian order. Returns the number of bytes transferred on each access. 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 is issued.

15.4.5 Access

15.4.5.1 set

```
virtual void set( uvm_reg_data_t value,
                  const std::string& fname = "",
                  int lineno = 0 );
```

The member function **set** shall specify the desired value of the fields in the register to the specified value. It 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. The member function **uvm_reg::update** is used to update the actual register with the mirrored value or member function **uvm_reg::write** is used to set the actual register and its mirrored value.

Unless this member function is used, the desired value is equal to the mirrored value.

See Section 15.5.5.1 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 as a scoreboard for the register values in the DUT, the member function **uvm_reg::predict** is used (see Section 15.4.5.15).

15.4.5.2 get

```
virtual uvm_reg_data_t get( const std::string& fname = "",
                            int lineno = 0 ) const;
```

The member function **get** shall return the desired value of the fields in the register. It 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 the **uvm_reg::set** (see Section 15.4.5.1), the desired value and the mirrored value are identical.

Use the member function **uvm_reg::read** (see Section 15.4.5.10) or **uvm_reg::peek** (see Section 15.4.5.12) to get the actual register value.

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would something different for these fields, the returned value is the actual content.

15.4.5.3 get_mirrored_value

```
virtual uvm_reg_data_t get_mirrored_value( const std::string& fname = "",
                                           int lineno = 0 ) const;
```

The member function **get_mirrored_value** shall return the mirrored value of the fields in the register. It does not actually read the value of the register in the design.

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would something different for these fields, the returned value is the actual content.

15.4.5.4 needs_update

```
virtual bool needs_update() const;
```

The member function **needs_update** shall return true if any of the fields need updating (see Section 15.5.5.8). Use the **uvm_reg::update** to actually update the DUT register (see Section 15.4.5.13).

15.4.5.5 reset

```
virtual void reset( const std::string& kind = "HARD" );
```

The member function **reset** shall set the desired and mirror value of the fields in this register to the reset value for the specified reset kind. See Section 15.5.5.4 for more details.

Also resets the semaphore that prevents concurrent access to the register. This semaphore shall be explicitly reset if a thread accessing this register array was killed in before the access was completed.

15.4.5.6 get_reset

```
virtual uvm_reg_data_t get_reset( const std::string& kind = "HARD" ) const;
```

The member function **get_reset** shall return the reset value for this register for the specified reset *kind*.

15.4.5.7 has_reset

```
virtual bool has_reset( const std::string& kind = "HARD",
                        bool do_delete = false );
```

The member function **has_reset** shall check if any field in the register has a reset value specified for the specified reset kind. If argument *do_delete* is set to true, it removes the reset value, if any.

15.4.5.8 set_reset

```
virtual void set_reset( uvm_reg_data_t value,
                        const std::string& kind = "HARD" );
```

The member function **set_reset** shall specify or modify the reset value for all the fields in the register corresponding to the cause specified by *kind*.

15.4.5.9 write

```
virtual void write( uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **write** shall write the specifed *value* in the DUT register that corresponds to this abstraction class instance using the specified access *path*. If the register is mapped in more than one address *map*, an address map shall be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the registers shall not be written.

The mirrored value shall be updated using the member function **uvm_reg::predict** (see Section 15.4.5.15).

15.4.5.10 read

```
virtual void read( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall read and return *value* from the DUT register that corresponds to this abstraction class instance using the specified access *path*. If the register is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the register through a physical access is mimicked. For example, clear-on-read bits in the registers shall be set to zero.

The mirrored value shall be updated using the member function **uvm_reg::predict** (see Section 15.4.5.15).

15.4.5.11 poke

```
virtual void poke( uvm_status_e& status,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **poke** shall deposit the specified *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 mirrored value shall be updated using the member function **uvm_reg::predict** (see Section 15.4.5.15).

15.4.5.12 peek

```
virtual void peek( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **peek** shall read the current *value* from this register. It 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 mirrored value shall be updated using the member function **uvm_reg::predict** (see Section 15.4.5.15).

15.4.5.13 update

```
virtual void update( uvm_status_e& status,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int  lineno = 0 );
```

The member function **update** shall update the content of the register in the design to match the desired value. This member function performs the reverse operation of **uvm_reg::mirror** (see Section 15.4.5.14). Write this register if the DUT register is out-of-date with the desired/mirrored value in the abstraction class, as determined by the member function **uvm_reg::needs_update** (see Section 15.4.5.4).

The update can be performed using the using the physical interfaces (frontdoor) or **uvm_reg::poke** (see Section 15.4.5.11) (backdoor) access. If the register is mapped in multiple address maps and physical access is used (front-door), an address map shall be specified.

15.4.5.14 mirror

```
virtual void mirror( uvm_status_e& status,
                     uvm_check_e check  = UVM_NO_CHECK,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
```

```
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **mirror** shall read the register and optionally compared the readback value with the current mirrored value if argument *check* is **UVM_CHECK**. The mirrored value shall be updated using the member function **uvm_reg::predict** (see Section 15.4.5.15) based on the readback value.

The mirroring can be performed using the physical interfaces (frontdoor) or **uvm_reg::peek** (see Section 15.4.5.12) (backdoor).

If argument *check* is specified as **UVM_CHECK**, an error message is issued if the current mirrored value does not match the readback value. Any field whose check has been disabled with **uvm_reg_field::set_compare** (see Section 15.5.5.14) shall not be considered in the comparison.

If the register is mapped in multiple address maps and physical access is used (frontdoor access), an address *map* shall be specified. If the register contains write-only fields, their content is mirrored and optionally checked only if a **UVM_BACKDOOR** access path is used to read the register.

15.4.5.15 predict

```
virtual bool predict( uvm_reg_data_t value,
                      uvm_reg_byte_en_t be = -1,
                      uvm_predict_e kind = UVM_PREDICT_DIRECT,
                      uvm_path_e path = UVM_FRONTDOOR,
                      uvm_reg_map* map = NULL,
                      const std::string& fname = "",
                      int lineno = 0 );
```

The member function **predict** shall update the mirrored and desired value for this register.

It 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 Section 15.4.5.15 for more details.

The member function returns true if the prediction was successful for each field in the register.

15.4.5.16 is_busy

```
bool is_busy() const;
```

The member function **is_busy** shall returns true if register is currently being read or written.

15.4.6 Frontdoor

15.4.6.1 set_frontdoor

```
void set_frontdoor( uvm_reg_frontdoor* ftdr,
                    uvm_reg_map* map = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **set_frontdoor** shall specify a user-defined frontdoor for this register.

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 needs to be defined and associated with the corresponding register abstraction class. If the register is mapped in multiple address maps, an address map needs to be specified.

15.4.6.2 get_frontdoor

```
uvm_reg_frontdoor* get_frontdoor( uvm_reg_map* map = NULL ) const;
```

The member function **get_frontdoor** shall return the user-defined frontdoor for this register.

If the member function returns NULL, no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the member function **uvm_reg::set_frontdoor**.

If the register is mapped in multiple address maps, an address map needs to be specified.

15.4.7 Backdoor

NOTE—Backdoor access is not yet available in UVM-SystemC.

15.4.7.1 set_backdoor

```
void set_backdoor( uvm_reg_backdoor* bkdr,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **set_backdoor** shall specify a user-defined backdoor for this register.

By default, registers are accessed via the built-in string-based DPI routines if an HDL path has been specified using the member function **uvm_reg::configure** or **uvm_reg::add_hdl_path**.

If this default mechanism is not suitable (e.g. because the register is not implemented in HDL), a user-defined access mechanism needs to be defined and associated with the corresponding register abstraction class.

A user-defined backdoor is required if active update of the mirror of this register abstraction class, based on observed changes of the corresponding DUT register, is used.

15.4.7.2 get_backdoor

```
uvm_reg_backdoor* get_backdoor( bool inherited = true ) const;
```

The member function **get_backdoor** shall return the user-defined backdoor for this register.

If the member function returns NULL, no user-defined backdoor has been defined. A user-defined frontdoor is defined by using the member function **uvm_reg::set_backdoor**.

If no argument is specified or the argument *inherited* is set to true, the member function returns the backdoor of the parent block if none have been specified for this register.

15.4.7.3 clear_hdl_path

```
void clear_hdl_path( const std::string& kind = "RTL" );
```

The member function **clear_hdl_path** shall remove any previously specified HDL path to the register instance for the specified design abstraction.

15.4.7.4 add_hdl_path

```
void add_hdl_path( std::vector<uvm_hdl_path_slice> slices,
                   const std::string& kind = "RTL" );
```

The member function **add_hdl_path** shall add the specified HDL path to the register instance for the specified design abstraction. This member function may be called more than once for the same design abstraction if the register is physically duplicated in the design abstraction. If the register is implemented using a single HDL variable, The array should specify a single slice with its offset and size specified as -1.

15.4.7.5 add_hdl_path_slice

```
void add_hdl_path_slice( const std::string& name,
                         int offset,
                         int size,
                         bool first = false,
                         const std::string& kind = "RTL" );
```

The member function **add_hdl_path_slice** shall append the specified HDL slice to the HDL path of the register instance for the specified design abstraction. If the argument *first* is set to true, it starts the specification of a duplicate HDL implementation of the register.

15.4.7.6 has_hdl_path

```
bool has_hdl_path( const std::string& kind = "" ) const;
```

The member function **has_hdl_path** shall return true if the register instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, it shall use the default design abstraction specified for the parent block.

15.4.7.7 get_hdl_path

```
void get_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                   const std::string& kind = "" ) const;
```

The member function **get_hdl_path** shall return 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. If no design abstraction is specified, the default design abstraction for the parent block is used.

15.4.7.8 get_hdl_path_kinds

```
void get_hdl_path_kinds( std::vector<std::string>& kinds ) const;
```

The member function **get_hdl_path_kinds** shall return the design abstractions for which HDL paths have been defined.

15.4.7.9 get_full_hdl_path

```
void get_full_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                        const std::string& kind = "",
                        const std::string& separator = ".") const;
```

The member function **get_full_hdl_path** shall return 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. If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

15.4.7.10 backdoor_read

```
virtual void backdoor_read( uvm_reg_item* rw );
```

The member function **backdoor_read** shall offer user-defined backdoor read access. The member function overrides the default string-based DPI backdoor access read for this register type.

15.4.7.11 backdoor_write

```
virtual void backdoor_write( uvm_reg_item* rw );
```

The member function **backdoor_write** shall offer user-defined backdoor write access. The member function overrides the default string-based DPI backdoor access write for this register type.

15.4.7.12 backdoor_watch

```
virtual void backdoor_watch();
```

The member function **backdoor_watch** shall offer a user-defined DUT register change monitor. The member function watches the DUT register corresponding to this abstraction class instance for any change in value and return when a value-change occurs. There is no default implementation provided for this member function.

15.4.8 Coverage

NOTE—Functional coverage is not yet available in UVM-SystemC.

15.4.8.1 include_coverage

```
static void include_coverage( const std::string& scope,
                              uvm_reg_cvr_t models,
                              uvm_object* accessor = NULL );
```

The member function **include_coverage** shall specify which coverage model that needs to be included in various block, register or memory abstraction class instances.

Yhe coverage models are specified by OR'ing or adding the **uvm_coverage_model_e** coverage model identifiers corresponding to the coverage model to be included.

The argument *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 argument *scope* can be specified as a POSIX regular expression or simple pattern. See Section 10.5.6 for more details.

The specification of which coverage model to include in which abstraction class is stored in a **uvm_reg_cvr_t** resource in the **uvm_resource_db** resource database, in the "**uvm_reg::**" scope namespace.

15.4.8.2 build_coverage

```
protected: uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
```

The member function **build_coverage** shall check which of the specified coverage model are built in this instance of the register abstraction class, as specified by calls to **uvm_reg::include_coverage**. *models* are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e**. The member function returns the sum of all coverage models to be built in the register model.

15.4.8.3 add_coverage

```
protected: virtual void add_coverage( uvm_reg_cvr_t models );
```

The member function **add_coverage** shall specify that additional coverage models are available. Add 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**. This member function shall be called only in the constructor of subsequently derived classes.

15.4.8.4 has_coverage

```
virtual bool has_coverage( uvm_reg_cvr_t models ) const;
```

The member function **has_coverage** shall return 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**.

15.4.8.5 set_coverage

```
virtual uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
```

The member function **set_coverage** shall specify the collection of functional coverage measurements for this register. The functional coverage measurement is turned on for every coverage model specified using **uvm_coverage_model_e** symbolic identifiers. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. The member function returns the sum of all functional coverage models whose measurements were previously on.

This member function can only control the measurement of functional coverage models that are present in the register abstraction classes, then enabled during construction. See Section 15.4.8.4 to identify the available functional coverage models.

15.4.8.6 get_coverage

```
virtual bool get_coverage( uvm_reg_cvr_t is_on ) const;
```

The member function **get_coverage** shall 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 Section 15.4.8.5 for more details.

15.4.8.7 sample

```
protected: virtual void sample( uvm_reg_data_t data,
                                uvm_reg_data_t byte_en,
                                bool is_read,
                                uvm_reg_map* map );
```

The member function **sample** shall specify the Functional coverage measurement method.

This member function 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. The member function may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

15.4.8.8 sample_values

```
virtual void sample_values();
```

The member function **sample_values** shall specify the functional coverage measurement method for field values.

This member function is invoked by the application or by the member function **uvm_reg_block::sample_values** of the parent block to trigger the sampling of the current field values in the register-level functional coverage model.

This member function may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model.

15.4.9 Callbacks

15.4.9.1 pre_write

```
virtual void pre_write( uvm_reg_item* rw );
```

The member function **pre_write** shall be 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 shall be used to perform the register operation. If the status is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The registered callback member functions are invoked after the invocation of this member function. All register callbacks are executed before the corresponding field callbacks.

15.4.9.2 post_write

```
virtual void post_write( uvm_reg_item* rw );
```

The member function **post_write** shall be called after register write.

If the specified status is modified, the updated status shall be returned by the register operation.

The registered callback member functions are invoked before the invocation of this member function. All register callbacks are executed before the corresponding field callbacks.

15.4.9.3 pre_read

```
virtual void pre_read( uvm_reg_item* rw );
```

The member function **pre_read** shall be called before register read.

If the specified access path or address map are modified, the updated access path or address map shall be used to perform the register operation. If the status is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The registered callback member functions are invoked after the invocation of this member function. All register callbacks are executed before the corresponding field callbacks.

15.4.9.4 post_read

```
virtual void post_read( uvm_reg_item* rw );
```

The member function **post_read** shall be called after register read.

If the specified readback data or status is modified, the updated readback data or status shall be returned by the register operation.

The registered callback member functions are invoked before the invocation of this member function. All register callbacks are executed before the corresponding field callbacks.

15.5 uvm_reg_field

The class **uvm_reg_field** defines 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 use the access the register (thus the field).

15.5.1 Class definition

```
namespace uvm {

  class uvm_reg_field : public uvm_object
  {
   public:

    // Constructor
    uvm_reg_field( const std::string& name = "uvm_reg_field" );

    // Group: Initialization

    void configure( uvm_reg* parent,
                    unsigned int size,
                    unsigned int lsb_pos,
                    const std::string& access,
                    bool is_volatile, // changed icm UVM-SV
                    uvm_reg_data_t reset,
                    bool has_reset,
                    bool is_rand,
                    bool individually_accessible );

    // Group: Introspection

    virtual const std::string get_name();
    virtual const std::string get_full_name() const;
    virtual uvm_reg* get_parent() const;
    virtual unsigned int get_lsb_pos() const;
```

```
virtual unsigned int get_n_bits() const;
static unsigned int get_max_size();
virtual std::string set_access( const std::string& mode );
static bool define_access( std::string name );
virtual std::string get_access( uvm_reg_map* map = NULL ) const;
virtual bool is_known_access( uvm_reg_map* map = NULL ) const;
virtual void set_volatility( bool is_volatile );
virtual bool is_volatile() const;

// Group: Access

virtual void set( uvm_reg_data_t value,
                  const std::string& fname = "",
                  int lineno = 0);

virtual uvm_reg_data_t get( const std::string& fname = "",
                            int lineno = 0 ) const;

virtual uvm_reg_data_t get_mirrored_value( const std::string& fname = "",
                                           int lineno = 0 ) const;

virtual void reset( const std::string& kind = "HARD" );
virtual uvm_reg_data_t get_reset( const std::string& kind = "HARD" ) const;

virtual bool has_reset( const std::string& kind = "HARD",
                        bool do_delete = 0 );

virtual void set_reset( uvm_reg_data_t value,
                        const std::string& kind = "HARD" );

virtual bool needs_update() const;

virtual void write( uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );

virtual void read( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void poke( uvm_status_e& status,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void peek( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void mirror( uvm_status_e& status,
                     uvm_check_e check = UVM_NO_CHECK,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

```
    void set_compare( uvm_check_e check = UVM_CHECK );
    uvm_check_e get_compare() const;

    bool is_indv_accessible( uvm_path_e path,
                             uvm_reg_map* local_map) const;

    bool predict( uvm_reg_data_t value,
                  uvm_reg_byte_en_t be = -1,
                  uvm_predict_e kind = UVM_PREDICT_DIRECT,
                  uvm_path_e path = UVM_FRONTDOOR,
                  uvm_reg_map* map = NULL,
                  const std::string& fname = "",
                  int lineno = 0 );

    // Group: Callbacks

    virtual void pre_write( uvm_reg_item* rw );
    virtual void post_write( uvm_reg_item* rw );
    virtual void pre_read( uvm_reg_item* rw);
    virtual void post_read( uvm_reg_item* rw);

  }; // class uvm_reg_field

} // namespace uvm
```

15.5.2 Constructor

```
uvm_reg_field( const std::string& name = "uvm_reg_field" );
```

The constructor shall create a new field instance with the specified name. This constructor shall not be used directly. The factory member function **uvm_reg_field::type_id::create** should be used instead.

15.5.3 Initialization

15.5.3.1 configure

```
void configure( uvm_reg* parent,
                unsigned int size,
                unsigned int lsb_pos,
                const std::string& access,
                bool is_volatile, // changed icm UVM-SV
                uvm_reg_data_t reset,
                bool has_reset,
                bool is_rand,
                bool individually_accessible );
```

The member function **configure** shall specify the parent register of this field, its size in bits, the position of its least-significant bit within the register relative to the least-significant bit 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 Section 15.5.4.7 for a specification of the pre-defined field access policies.

If the field access policy is a pre-defined policy and not one of “RW”, “WRC”, “WRS”, “WO”, “W1”, or “WO1”, the value of argument *is_rand* is ignored and the **rand_mode** for the field instance is turned off since it cannot be written.

15.5.4 Introspection

15.5.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this field.

15.5.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this field. The base of the hierarchical name is the root block.

15.5.4.3 get_parent

```
virtual uvm_reg* get_parent() const;
```

The member function **get_parent** shall return the parent register.

15.5.4.4 get_lsb_pos

```
virtual unsigned int get_lsb_pos() const;
```

The member function **get_lsb_pos** shall return the index of the least significant bit of the field in the register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register.

15.5.4.5 get_n_bits

```
virtual unsigned int get_n_bits() const;
```

The member function **get_n_bits** shall return the width, in number of bits, of the field.

15.5.4.6 get_max_size

```
static unsigned int get_max_size();
```

The member function **get_max_size** shall return the width, in number of bits, of the largest field.

15.5.4.7 set_access

```
virtual std::string set_access( const std::string& mode );
```

The member function **set_access** shall modify the access policy of the field to the specified one and return the previous access policy.

The pre-defined access policies are as follows. The effect of a read operation are applied after the current value of the field is sampled. The read operation shall return the current value, not the value affected by the read operation (if any).

Table 15.1—Access policies

”RO”	W: no effect, R: no effect
”RW”	W: as-is, R: no effect
”RC”	W: no effect, R: clears all bits
”RS”	W: no effect, R: sets all bits
”WRC”	W: as-is, R: clears all bits
”WRS”	W: as-is, R: sets all bits
”WC”	W: clears all bits, R: no effect
”WS”	W: sets all bits, R: no effect
”WSRC”	W: sets all bits, R: clears all bits
”WCRS”	W: clears all bits, R: sets all bits
”W1C”	W: 1/0 clears/no effect on matching bit, R: no effect
”W1S”	W: 1/0 sets/no effect on matching bit, R: no effect
”W1T”	W: 1/0 toggles/no effect on matching bit, R: no effect
”W0C”	W: 1/0 no effect on/clears matching bit, R: no effect
”W0S”	W: 1/0 no effect on/sets matching bit, R: no effect
”W0T”	W: 1/0 no effect on/toggles matching bit, R: no effect
”W1SRC”	W: 1/0 sets/no effect on matching bit, R: clears all bits
”W1CRS”	W: 1/0 clears/no effect on matching bit, R: sets all bits
”W0SRC”	W: 1/0 no effect on/sets matching bit, R: clears all bits
”W0CRS”	W: 1/0 no effect on/clears matching bit, R: sets all bits
”WO”	W: as-is, R: error
”WOC”	W: clears all bits, R: error
”WOS”	W: sets all bits, R: error
”W1”	W: first one after HARD reset is as-is, other W have no effects, R: no effect
”WO1”	W: first one after HARD reset is as-is, other W have no effects, R: error
”NOACCESS”	W: no effect, R: no effect

Modifying the access of a field shall make the register model diverge from the specification that was used to create it.

15.5.4.8 define_access

```
static bool define_access( std::string name );
```

The member function **define_access** shall specify a new access policy value.

Because field access policies are specified using string values, there is no mechnaism to verify if a specific access value is valid or not. To help catch typing errors, user-defined access values needs to be defined using this member function to avoid begin reported as an invalid access policy.

The name of field access policies are always converted to all uppercase.

The member function shall return true if the new access policy was not previously defined. It shall return false otherwise, but does not issue an error message.

15.5.4.9 get_access

```
virtual std::string get_access( uvm_reg_map* map = NULL ) const;
```

The member function **get_access** shall return the access policy of the field. It 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 map, 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. For example, 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 (field access of WO, and map access value of RO, etc), the member functions return value is NOACCESS.

15.5.4.10 is_known_access

```
virtual bool is_known_access( uvm_reg_map* map = NULL ) const;
```

The member function **is_known_access** shall return true if the current access policy of the field, when written and read through the specified address map, is a built-in access policy. Otherwise it shall return false.

15.5.4.11 set_volatility

```
virtual void set_volatility( bool is_volatile );
```

The member function **set_volatility** shall specify the volatility of the field to the specified one. Modifying the volatility of a field shall make the register model diverge from the specification that was used to create it.

15.5.4.12 is_volatile

```
virtual bool is_volatile() const;
```

The member function **is_volatile** shall return true if the value of the register is not predictable because it may change between consecutive accesses. This typically indicates a field whose value is updated by the DUT. The nature or cause of the change is not specified. The member function returns false if the value of the register is not modified between consecutive accesses.

NOTE—UVM uses the IP-XACT definition of "volatility" as defined in IEEE Std. 1685-2014[6].

15.5.5 Access

15.5.5.1 set

```
virtual void set( uvm_reg_data_t value,
                  const std::string& fname = "",
                  int lineno = 0);
```

The member function **set** shall specify the desired value of the field to the specified value modified by the field access policy. It does not actually set the value of the field in the design, only the desired value in the abstraction class. Use the member function **uvm_reg::update** (see Section 15.4.5.13) to update the actual register with the desired value or the member function **uvm_reg_field::write** (see Section 15.5.5.9) to actually write the field and update its mirrored value.

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 member function (when eventually followed by a call to **uvm_reg::update**) is a zero-time functional replacement for the member function **uvm_reg_field::write**. For example, the desired value of a read-only field is not modified by this

[6] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, https://standards.ieee.org/standard/1685-2014.html

member function 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 Section 15.5.5.17) to modify the mirrored value of the field.

15.5.5.2 get

```
virtual uvm_reg_data_t get( const std::string& fname = "",
                            int lineno = 0 ) const;
```

The member function **get** shall return the desired value of the field. It 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 the **uvm_reg_field::set**, the desired value and the mirrored value are identical.

Use the member function **uvm_reg_field::read** or **uvm_reg_field::peek** to get the actual field value.

If the field is write-only, the desired/mirrored value is the value last written and assumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content.

15.5.5.3 get_mirrored_value

```
virtual uvm_reg_data_t get_mirrored_value( const std::string& fname = "",
                                           int lineno = 0 ) const;
```

The member function **get_mirrored_value** shall return the mirrored value of the field. It 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 assumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content.

15.5.5.4 reset

```
virtual void reset( const std::string& kind = "HARD" );
```

The member function **reset** shall set 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.

It 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.

15.5.5.5 get_reset

```
virtual uvm_reg_data_t get_reset( const std::string& kind = "HARD" ) const;
```

The member function **get_reset** shall return the reset value for this field for the specified reset *kind*. It returns the current field value if no reset value has been specified for the specified reset event.

15.5.5.6 has_reset

```
virtual bool has_reset( const std::string& kind = "HARD",
```

```
                        bool do_delete = false );
```

The member function **has_reset** shall return true if this field has a reset value specified for the specified reset kind. If argument *do_delete* is set to true, it removes the reset value, if any.

15.5.5.7 set_reset

```
virtual void set_reset( uvm_reg_data_t value,
                        const std::string& kind = "HARD" );
```

The member function **set_reset** shall specify or modify the reset value for this field corresponding to the cause specified by argument *kind*.

15.5.5.8 needs_update

```
virtual bool needs_update() const;
```

The member function **needs_update** shall check 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 (more specifically what the abstraction class thinks the state of the DUT is) is outdated. This member function shall return 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 the **uvm_reg::update** (see Section 15.4.5.13) to actually update the DUT field.

15.5.5.9 write

```
virtual void write( uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );;
```

The member function **write** shall write the specified *value* in the DUT field that corresponds to this abstraction class instance using the specified access *path*. If the register containing this field is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, read-only bits in the field shall not be written.

The mirrored value shall be updated using the member function **uvm_reg_field::predict** (see Section 15.5.5.17).

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 support 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 backdoor 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.

15.5.5.10 read

```
virtual void read( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall read and return *value* from the DUT field that corresponds to this abstraction class instance using the specified access *path*. If the register containing this field is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the field through a physical access is mimicked. For example, clear-on-read bits in the field shall be set to zero.

The mirrored value shall be updated using the member function **uvm_reg_field::predict** (see Section 15.5.5.17).

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 support 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 backdoor access is used, the entire containing register is peeked and the mirrored value of the other fields in the register is updated.

15.5.5.11 poke

```
virtual void poke( uvm_status_e& status,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **poke** shall deposit the specified *value* in the DUT field corresponding to this abstraction class instance, as-is, using a back-door access. A peek-modify-poke process is used in a best-effort not to modify the value of the other fields in the register.

The mirrored value shall be updated using the member function **uvm_reg_field::predict** (see Section 15.5.5.17).

15.5.5.12 peek

```
virtual void peek( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **peek** shall read the current value from this field. It samples the value in the DUT field corresponding to this abstraction class instance using a back-door access. The field value is sampled, not modified. It uses the HDL path for the design abstraction specified by kind.

The entire containing register is peeked and the mirrored value of the other fields in the register are updated using the **uvm_reg_field::predict** (see Section 15.5.5.17).

15.5.5.13 mirror

```
virtual void mirror( uvm_status_e& status,
                     uvm_check_e check = UVM_NO_CHECK,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **mirror** shall read the field and optionally compared the readback value with the current mirrored value if *check* is **UVM_CHECK**. The mirrored value shall be updated using the member function **predict** based on the readback value.

The argument *path* specifies whether to mirror using the **UVM_FRONTDOOR** by using member function **read** or **UVM_BACKDOOR** by using member function **peek**.

If argument *check* is specified as **UVM_CHECK**, an error message is issued if the current mirrored value does not match the readback value, unless set_compare was used disable the check.

If the containing register is mapped in multiple address maps and physical access is used (front-door access), an address map shall be specified. For write-only fields, their content is mirrored and optionally checked only if a **UVM_BACKDOOR** access path is used to read the field.

15.5.5.14 set_compare

```
void set_compare( uvm_check_e check = UVM_CHECK );
```

The member function **set_compare** shall specify the comparison policy during a mirror update. The field value is checked against its mirror only when both the argument *check* in **uvm_reg_block::mirror** (see Section 15.1.6.5), **uvm_reg::mirror** (see Section 15.4.5.14), or **uvm_reg_field::mirror** (see Section 15.5.5.13) and the comparison policy for the field is **UVM_CHECK**.

15.5.5.15 get_compare

```
uvm_check_e get_compare() const;
```

The member function **get_compare** shall return the comparison policy for this field.

15.5.5.16 is_indv_accessible

```
bool is_indv_accessible( uvm_path_e path,
                         uvm_reg_map* local_map ) const;
```

The member function **is_indv_accessible** shall return true if this field can be written individually, i.e. without affecting other fields in the containing register. Otherwise it shall return false.

15.5.5.17 predict

```
bool predict( uvm_reg_data_t value,
```

```
               uvm_reg_byte_en_t be = -1,
               uvm_predict_e kind = UVM_PREDICT_DIRECT,
               uvm_path_e path = UVM_FRONTDOOR,
               uvm_reg_map* map = NULL,
               const std::string& fname = "",
               int lineno = 0 );
```

The member function **predict** shall update the mirrored and desired value for this field. It predicts the mirror and desired value of the field based on the specified observed value on a bus using the specified address map.

If argument *kind* is specified as **UVM_PREDICT_READ**, the value was observed in a read transaction on the specified address map or backdoor (if path is **UVM_BACKDOOR**). If argument *kind* is specified as **UVM_PREDICT_WRITE**, the value was observed in a write transaction on the specified address map or backdoor (if path is **UVM_BACKDOOR**). If argument *kind* is specified as **UVM_PREDICT_DIRECT**, the value was computed and is updated as-is, without regard to any access policy. For example, the mirrored value of a read-only field is modified by this member function if kind is specified as **UVM_PREDICT_DIRECT**.

This member function does not allow an update of the mirror (or desired) 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 member function returns true if the prediction was successful.

15.5.6 Callbacks

15.5.6.1 pre_write

```
virtual void pre_write( uvm_reg_item* rw );
```

The member function **pre_write** shall be 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 shall be used to perform the register operation. If the status is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callback member functions are invoked after the invocation of this member function.

15.5.6.2 post_write

```
virtual void post_write( uvm_reg_item* rw );
```

The member function **post_write** shall be called after field write.

If the specified status is modified, the updated status shall be returned by the register operation.

The field callback member functions are invoked after the callback methods on the containing register. The registered callback member functions are invoked before the invocation of this member function.

15.5.6.3 pre_read

```
virtual void pre_read( uvm_reg_item* rw);
```

The member function **pre_read** shall be called before field read.

If the access path or address map in the rw argument are modified, the updated access path or address map shall be used to perform the register operation. If the status is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The field callback member functions are invoked after the callback member functions on the containing register. The registered callback member functions are invoked after the invocation of this member function.

15.5.6.4 post_read

```
virtual void post_read( uvm_reg_item* rw);
```

The member function **post_read** shall be called after field read.

If the specified readback data or *status* in the argument *rw* is modified, the updated readback data or status shall be returned by the register operation.

The field callback member functions are invoked after the callback member functions on the containing register. The registered callback methods are invoked before the invocation of this member function.

15.6 uvm_mem

The class **uvm_mem** defines 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 modelling techniques. Rather than relying on a mirror, it is recommended that backdoor access be used instead.

15.6.1 Class definition

```
namespace uvm {

  class uvm_mem : public uvm_object
  {
   public:

    typedef enum {UNKNOWNS, ZEROES, ONES, ADDRESS, VALUE, INCR, DECR} init_e;

    // Constructor
    explicit uvm_mem( const std::string& name,
                      unsigned long size,
                      unsigned int n_bits,
                      const std::string& access = "RW",
                      int has_coverage = UVM_NO_COVERAGE );

    // Group: Initialization

    void configure( uvm_reg_block* parent,
                    const std::string& hdl_path = "" );

    void set_offset( uvm_reg_map* map,
                     uvm_reg_addr_t offset,
                     bool unmapped = 0 );

    // Group: Introspection

    virtual const std::string get_name() const;
    virtual const std::string get_full_name() const;
    virtual uvm_reg_block* get_parent() const;
    virtual int get_n_maps() const;
    bool is_in_map( uvm_reg_map* map ) const;
    virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
```

```
virtual std::string get_rights( const uvm_reg_map* map = NULL ) const;
virtual std::string get_access( const uvm_reg_map* map = NULL ) const;
unsigned long get_size() const;
unsigned int get_n_bytes() const;
unsigned int get_n_bits() const;
static unsigned int get_max_size();
virtual void get_virtual_registers( std::vector<uvm_vreg*>& regs ) const;
virtual void get_virtual_fields( std::vector<uvm_vreg_field*>& fields) const;
virtual uvm_vreg* get_vreg_by_name( const std::string& name ) const;
virtual uvm_vreg_field* get_vfield_by_name( const std::string& name ) const;

virtual uvm_vreg* get_vreg_by_offset( uvm_reg_addr_t offset,
                                      const uvm_reg_map* map = NULL ) const;

virtual uvm_reg_addr_t get_offset( uvm_reg_addr_t offset = 0,
                                   const uvm_reg_map* map = NULL ) const;

virtual uvm_reg_addr_t get_address( uvm_reg_addr_t offset = 0,
                                    const uvm_reg_map* map = NULL ) const;

virtual int get_addresses( std::vector<uvm_reg_addr_t>& addr,
                           const uvm_reg_map* map = NULL,
                           uvm_reg_addr_t offset = 0 ) const;

// Group: HDL Access

virtual void write( uvm_status_e& status,
                    uvm_reg_addr_t offset,
                    uvm_reg_data_t value,
                    uvm_path_e path   = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );


virtual void read( uvm_status_e& status,
                   uvm_reg_addr_t offset,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void burst_write( uvm_status_e& status,
                          uvm_reg_addr_t offset,
                          std::vector<uvm_reg_data_t> value,
                          uvm_path_e path = UVM_DEFAULT_PATH,
                          uvm_reg_map* map = NULL,
                          uvm_sequence_base* parent = NULL,
                          int prior = -1,
                          uvm_object* extension = NULL,
                          const std::string& fname = "",
                          int lineno = 0);

virtual void burst_read( uvm_status_e& status,
                         uvm_reg_addr_t offset,
                         std::vector<uvm_reg_data_t>& value,
                         uvm_path_e path = UVM_DEFAULT_PATH,
                         uvm_reg_map* map = NULL,
                         uvm_sequence_base* parent = NULL,
                         int prior = -1,
                         uvm_object* extension = NULL,
                         const std::string& fname = "",
                         int lineno = 0 );

virtual void poke( uvm_status_e& status,
                   uvm_reg_addr_t offset,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
```

```
                    const std::string& fname = "",
                    int lineno = 0 );

  virtual void peek( uvm_status_e& status,
                    uvm_reg_addr_t offset,
                    uvm_reg_data_t& value,
                    const std::string& kind = "",
                    uvm_sequence_base* parent = NULL,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );

  // Group: Frontdoor

  void set_frontdoor( uvm_reg_frontdoor* ftdr,
                     uvm_reg_map* map = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );

  uvm_reg_frontdoor* get_frontdoor( const uvm_reg_map* map = NULL ) const;

  // Group: Backdoor

  void set_backdoor( uvm_reg_backdoor* bkdr,
                    const std::string& fname = "",
                    int lineno = 0 );

  uvm_reg_backdoor* get_backdoor( bool inherited = true );
  void clear_hdl_path( const std::string& kind = "RTL" );

  void add_hdl_path( std::vector<uvm_hdl_path_slice> slices,
                    const std::string& kind = "RTL" );

  void add_hdl_path_slice( const std::string& name,
                          int offset,
                          int size,
                          bool first = false,
                          const std::string& kind = "RTL" );

  bool has_hdl_path( const std::string& kind = "" ) const;

  void get_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                    const std::string& kind = "" ) const;

  void get_full_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                         const std::string& kind = "",
                         const std::string& separator = "." ) const;

  void get_hdl_path_kinds( std::vector<std::string>& kinds ) const;

 protected:
  virtual void backdoor_read( uvm_reg_item* rw );

 public:
  virtual void backdoor_write( uvm_reg_item* rw );

  // Group: Callbacks

  virtual void pre_write( uvm_reg_item* rw );
  virtual void post_write( uvm_reg_item* rw );
  virtual void pre_read( uvm_reg_item* rw );
  virtual void post_read( uvm_reg_item* rw );

  // Group: Coverage

 protected:
  uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
  virtual void add_coverage( uvm_reg_cvr_t models );

 public:
  virtual bool has_coverage( uvm_reg_cvr_t models ) const;
  virtual uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
  virtual bool get_coverage( uvm_reg_cvr_t is_on );

 protected:
  virtual void sample( uvm_reg_addr_t offset,
                      bool is_read,
```

```
                        uvm_reg_map* map );

    // Data members
    uvm_mem_mam* mam;

  }; // class uvm_mem

} // namespace uvm
```

15.6.2 Constructor

```
explicit uvm_mem( const std::string& name,
                  unsigned long size,
                  unsigned int n_bits,
                  const std::string& access = "RW",
                  int has_coverage = UVM_NO_COVERAGE );
```

The constructor shall create an instance of a memory abstraction class with the specified name.

The argument *size* specifies the total number of memory locations. The argument *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 argument *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 Section 15.16.2.9).

15.6.3 Initialization

15.6.3.1 configure

```
void configure( uvm_reg_block* parent,
                const std::string& hdl_path = "" );
```

The member function **configure** shall specify 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 needs to be specified using the member function **add_hdl_path** or **add_hdl_path_slice**.

15.6.3.2 set_offset

```
void set_offset( uvm_reg_map* map,
                 uvm_reg_addr_t offset,
                 bool unmapped = 0 );
```

The member function **set_offset** shall specify the offset of a memory within an address map. It shall use the member function **uvm_reg_map::add_reg** (see Section 15.2.3.3). This member function is used to modify that offset dynamically.

Modifying the offset of a register makes the register model diverge from the specification that was used to create it.

15.6.4 Introspection

15.6.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this memory.

15.6.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this memory. The base of the hierarchical name is the root block.

15.6.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent block.

15.6.4.4 get_n_maps

```
virtual int get_n_maps() const;
```

The member function **get_n_maps** shall return the number of address maps this memory is mapped in.

15.6.4.5 is_in_map

```
bool is_in_map( uvm_reg_map* map ) const;
```

The member function **is_in_map** shall return true if this memory is in the specified address map, otherwise return false.

15.6.4.6 get_maps

```
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
```

The member function **get_maps** shall return all of the address maps where this memory is mapped.

15.6.4.7 get_rights

```
virtual std::string get_rights( uvm_reg_map* map = NULL ) const;
```

The member function **get_rights** shall return the accessibility (“RW, “RO”, or “WO”) of this memory in the given map.

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, an error message is issued and “RW” is returned.

15.6.4.8 get_access

```
virtual std::string get_access( const uvm_reg_map* map = NULL ) const;
```

The member function **get_access** shall return 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. For example, a read-write memory accessed through a domain with read-only restrictions would return “RO”.

15.6.4.9 get_size

```
unsigned long get_size() const;
```

The member function **get_size** shall return the number of unique memory locations in this memory.

15.6.4.10 get_n_bytes

```
unsigned int get_n_bytes() const;
```

The member function **get_n_bytes** shall return the width, in number of bytes, of each memory location.

15.6.4.11 get_n_bits

```
unsigned int get_n_bits() const;
```

The member function **get_n_bits** shall return the width, in number of bits, of each memory location.

15.6.4.12 get_max_size

```
static unsigned int get_max_size();
```

The member function **get_max_size** shall return the maximum width, in number of bits, of all memories.

15.6.4.13 get_virtual_registers

```
virtual void get_virtual_registers( std::vector<uvm_vreg*>& regs ) const;
```

The member function **get_virtual_registers** shall return the virtual registers in this memory. The order in which the virtual registers are located in the vector is not specified.

15.6.4.14 get_virtual_fields

```
virtual void get_virtual_fields( std::vector<uvm_vreg_field*>& fields) const;
```

The member function **get_virtual_fields** shall return the virtual fields in the memory. The order in which the virtual fields are located in the vector is not specified.

15.6.4.15 get_vreg_by_name

```
virtual uvm_vreg* get_vreg_by_name( const std::string& name ) const;
```

The member function **get_vreg_by_name** shall search for the virtual register with the specified name implemented in this memory and shall return its abstraction class instance. If no virtual register with the specified name is found, the member function returns NULL.

15.6.4.16 get_vfield_by_name

```
virtual uvm_vreg_field* get_vfield_by_name( const std::string& name ) const;
```

The member function **get_vfield_by_name** shall search for the virtual field with the specified name implemented in this memory and shall return its abstraction class instance. If no virtual field with the specified name is found, the member function returns NULL.

15.6.4.17 get_vreg_by_offset

```
virtual uvm_vreg* get_vreg_by_offset( uvm_reg_addr_t offset,
                                      const uvm_reg_map* map = NULL ) const;
```

The member function **get_vreg_by_offset** shall search for the virtual register implemented in this memory at the specified offset in the specified address map and returns its abstraction class instance. If no virtual register at the offset is found, it returns NULL.

15.6.4.18 get_offset

```
virtual uvm_reg_addr_t get_offset( uvm_reg_addr_t offset = 0,
                                   const uvm_reg_map* map = NULL ) const;
```

The member function **get_offset** shall return the base offset of the specified location in this memory in an 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 is issued.

15.6.4.19 get_address

```
virtual uvm_reg_addr_t get_address( uvm_reg_addr_t offset = 0,
                                    const uvm_reg_map* map = NULL ) const;
```

The member function **get_address** shall return the base external physical address of the specified location in this memory if accessed through the specified 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 is issued.

15.6.4.20 get_addresses

```
virtual int get_addresses( std::vector<uvm_reg_addr_t>& addr,
                           const uvm_reg_map* map = NULL,
                           uvm_reg_addr_t offset = 0 ) const;
```

The member function **get_addresses** shall return the base external physical address of the specified location in this memory if accessed through the specified 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 is issued.

15.6.5 HDL access

15.6.5.1 write

```
virtual void write( uvm_status_e& status,
                    uvm_reg_addr_t offset,
                    uvm_reg_data_t value,
                    uvm_path_e path   = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **write** shall write the specified *value* in the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path*. If the memory is mapped in more than one address map, an address *map* needs to be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the memory through a physical access is mimicked. For example, a read-only memory will remain unchanged.

15.6.5.2 read

```
virtual void read( uvm_status_e& status,
                   uvm_reg_addr_t offset,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall read and return *value* from the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path*. If the memory is mapped in more than one address map, an address *map* needs to be specified if a physical access is used (front-door access).

15.6.5.3 burst_write

```
virtual void burst_write( uvm_status_e& status,
                          uvm_reg_addr_t offset,
                          std::vector<uvm_reg_data_t> value,
                          uvm_path_e path = UVM_DEFAULT_PATH,
                          uvm_reg_map* map = NULL,
                          uvm_sequence_base* parent = NULL,
                          int prior = -1,
                          uvm_object* extension = NULL,
                          const std::string& fname = "",
                          int lineno = 0);
```

The member function **burst_write** shall burst-write the specified *values* in the memory locations beginning at the specified *offset*. If the memory is mapped in more than one address map, an address *map* needs to be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will remain unchanged.

15.6.5.4 burst_read

```
virtual void burst_read( uvm_status_e& status,
                         uvm_reg_addr_t offset,
                         std::vector<uvm_reg_data_t>& value,
                         uvm_path_e path = UVM_DEFAULT_PATH,
                         uvm_reg_map* map = NULL,
                         uvm_sequence_base* parent = NULL,
                         int prior = -1,
                         uvm_object* extension = NULL,
                         const std::string& fname = "",
                         int lineno = 0 );
```

The member function **burst_read** shall burst-read into *values* the data the memory locations beginning at the specified *offset*. If the memory is mapped in more than one address map, an address *map* needs to be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will remain unchanged.

15.6.5.5 poke

```
virtual void poke( uvm_status_e& status,
                   uvm_reg_addr_t offset,
                   uvm_reg_data_t value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **poke** shall deposit the specified *value* in the DUT memory location corresponding to this abstraction class instance at the specified *offset*, as-is, using a back-door access. It uses the HDL path for the design abstraction specified by *kind*.

15.6.5.6 peek

```
virtual void peek( uvm_status_e& status,
                   uvm_reg_addr_t offset,
                   uvm_reg_data_t& value,
                   const std::string& kind = "",
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **peek** shall read and return the current 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. It uses the HDL path for the design abstraction specified by *kind*.

15.6.6 Frontdoor

15.6.6.1 set_frontdoor

```
void set_frontdoor( uvm_reg_frontdoor* ftdr,
                    uvm_reg_map* map = NULL,
                    const std::string& fname = "",
```

```
                     int lineno = 0 );
```

The member function **set_frontdoor** shall specify a user-defined frontdoor for this memory.

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 needs to be defined and associated with the corresponding memory abstraction class. If the memory is mapped in multiple address maps, an address map needs to be specified.

15.6.6.2 get_frontdoor

```
uvm_reg_frontdoor* get_frontdoor( const uvm_reg_map* map = NULL ) const;
```

The member function **get_frontdoor** shall return the user-defined frontdoor for this memory.

If the member function returns NULL, no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the member function **uvm_mem::set_frontdoor**.

If the memory is mapped in multiple address maps, an address map needs to be specified.

15.6.7 Backdoor

NOTE—Backdoor access is not yet available in UVM-SystemC.

15.6.7.1 set_backdoor

```
void set_backdoor( uvm_reg_backdoor* bkdr,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **set_backdoor** shall specify a user-defined backdoor for this memory.

By default, memories are accessed via the built-in string-based DPI routines if an HDL path has been specified using the member function **uvm_mem::configure** or **uvm_mem::add_hdl_path**.

If this default mechanism is not suitable (e.g. because the memory is not implemented in HDL), a user-defined access mechanism needs to be defined and associated with the corresponding memory abstraction class.

15.6.7.2 get_backdoor

```
uvm_reg_backdoor* get_backdoor( bool inherited = true ) const;
```

The member function **get_backdoor** shall return the user-defined backdoor for this memory.

If the member function returns NULL, no user-defined backdoor has been defined. A user-defined frontdoor is defined by using the member function **uvm_mem::set_backdoor**.

If no argument is specified or the argument *inherited* is set to true, the member function returns the backdoor of the parent block if none have been specified for this memory.

15.6.7.3 clear_hdl_path

```
void clear_hdl_path( const std::string& kind = "RTL" );
```

The member function **clear_hdl_path** shall remove any previously specified HDL path to the memory instance for the specified design abstraction.

15.6.7.4 add_hdl_path

```
void add_hdl_path( std::vector<uvm_hdl_path_slice> slices,
                   const std::string& kind = "RTL" );
```

The member function **add_hdl_path** shall add the specified HDL path to the memory instance for the specified design abstraction. This member function may be called more than once for the same design abstraction if the memory is physically duplicated in the design abstraction.

15.6.7.5 add_hdl_path_slice

```
void add_hdl_path_slice( const std::string& name,
                         int offset,
                         int size,
                         bool first = false,
                         const std::string& kind = "RTL" );
```

The member function **add_hdl_path_slice** shall append the specified HDL slice to the HDL path of the memory instance for the specified design abstraction. If the argument *first* is set to true, it starts the specification of a duplicate HDL implementation of the memory.

15.6.7.6 has_hdl_path

```
bool has_hdl_path( const std::string& kind = "" ) const;
```

The member function **has_hdl_path** shall return true if the memory instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, it shall use the default design abstraction specified for the parent block.

15.6.7.7 get_hdl_path

```
void get_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                   const std::string& kind = "" ) const;
```

The member function **get_hdl_path** shall return 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. If no design abstraction is specified, the default design abstraction for the parent block is used.

15.6.7.8 get_full_hdl_path

```
void get_full_hdl_path( std::vector<uvm_hdl_path_concat>& paths,
                        const std::string& kind = "",
                        const std::string& separator = ".") const;
```

The member function **get_full_hdl_path** shall return the full hierarchical HDL path(s) defined for the specified design abstraction in the memory instance. There may be more than one path returned even if only one path was defined for the memory instance, if any of the parent components have more than one path defined for the same design abstraction. If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

15.6.7.9 get_hdl_path_kinds

```
void get_hdl_path_kinds( std::vector<std::string>& kinds ) const;
```

The member function **get_hdl_path_kinds** shall return the design abstractions for which HDL paths have been defined.

15.6.7.10 backdoor_read

```
protected: virtual void backdoor_read( uvm_reg_item* rw );
```

The member function **backdoor_read** shall offer user-defined backdoor read access. The member function overrides the default string-based DPI backdoor access read for this memory type.

15.6.7.11 backdoor_write

```
virtual void backdoor_write( uvm_reg_item* rw );
```

The member function **backdoor_write** shall offer user-defined backdoor write access. The member function overrides the default string-based DPI backdoor access write for this memory type.

15.6.8 Callbacks

15.6.8.1 pre_write

```
virtual void pre_write( uvm_reg_item* rw );
```

The member function **pre_write** shall be called before memory write.

If the *offset*, *value*, access *path* or address *map* are modified, the updated offset, data value, access path or address map shall be used to perform the memory operation. If the *status* is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The registered callback member functions are invoked after the invocation of this member function.

15.6.8.2 post_write

```
virtual void post_write( uvm_reg_item* rw );
```

The member function **post_write** shall be called after register write.

If the *status* is modified, the updated status shall be returned by the memory operation.

The registered callback member functions are invoked before the invocation of this member function.

15.6.8.3 pre_read

```
virtual void pre_read( uvm_reg_item* rw );
```

The member function **pre_read** shall be called before register read.

If the offset, access path or address map are modified, the updated offset, access path or address map shall be used to perform the memory operation. If the status is modified to anything other than **UVM_IS_OK**, the operation is aborted.

The registered callback member functions are invoked after the invocation of this member function.

15.6.8.4 post_read

```
virtual void post_read( uvm_reg_item* rw );
```

The member function **post_read** shall be called after memory read.

If the specified readback data or status is modified, the updated readback data or status shall be returned by the memory operation.

The registered callback member functions are invoked before the invocation of this member function.

15.6.9 Coverage

NOTE—Functional coverage is not yet available in UVM-SystemC.

15.6.9.1 build_coverage

```
protected: uvm_reg_cvr_t build_coverage( uvm_reg_cvr_t models );
```

The member function **build_coverage** shall check 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**. *models* are specified by adding the symbolic value of individual coverage model as defined in **uvm_coverage_model_e**. The member function returns the sum of all coverage models to be built in the memory model.

15.6.9.2 add_coverage

```
protected: virtual void add_coverage( uvm_reg_cvr_t models );
```

The member function **add_coverage** shall specify that additional coverage models are available. Add 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**. This member function shall be called only in the constructor of subsequently derived classes.

15.6.9.3 has_coverage

```
virtual bool has_coverage( uvm_reg_cvr_t models ) const;
```

The member function **has_coverage** shall return 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**.

15.6.9.4 set_coverage

```
virtual uvm_reg_cvr_t set_coverage( uvm_reg_cvr_t is_on );
```

The member function **set_coverage** shall specify the collection of functional coverage measurements for this memory. The functional coverage measurement is turned on for every coverage model specified using **uvm_coverage_model_e** symbolic identifiers. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. The member function returns the sum of all functional coverage models whose measurements were previously on.

This member function can only control the measurement of functional coverage models that are present in the memory abstraction classes, then enabled during construction. See Section 15.6.9.3 to identify the available functional coverage models.

15.6.9.5 get_coverage

```
virtual bool get_coverage( uvm_reg_cvr_t is_on ) const;
```

The member function **get_coverage** shall 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 Section 15.6.9.4 for more details.

15.6.9.6 sample

```
protected: virtual void sample( uvm_reg_addr_t offset,
                                bool is_read,
                                uvm_reg_map* map );
```

The member function **sample** shall specify the functional coverage measurement method.

This member function is invoked by the memory abstraction class whenever an address within one of its address map is successfully read or written. The specified offset is the offset within the memory, not an absolute address. The member function may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

15.7 uvm_reg_indirect_data

The class **uvm_reg_indirect_data** defines the abstraction class for indirect data access.

The class shall model the behavior of a register used to indirectly access a register array, indexed by a second address register. This class shall not be instantiated directly. A type-specific class extension shall be used to provide a factory-enabled constructor and specify the n_bits and coverage models.

15.7.1 Class definition

```
namespace uvm {

  class uvm_reg_indirect_data : public uvm_reg
  {
   public:

    uvm_reg_indirect_data( const std::string& name,
                           unsigned int n_bits,
                           int has_cover );

    void configure( uvm_reg* idx,
                    std::vector<uvm_reg*> reg_a,
                    uvm_reg_block* blk_parent,
                    uvm_reg_file* regfile_parent = NULL );
```

```
  }; // class uvm_reg_indirect_data

} // namespace uvm
```

15.7.2 Constructor

```
uvm_reg_indirect_data( const std::string& name,
                       unsigned int n_bits,
                       int has_cover );
```

The constructor shall create an instance of this class. The argument *n_bits* shall match the number of bits in the indirect register array.

15.7.3 Member functions

15.7.3.1 configure

```
void configure( uvm_reg* idx,
                std::vector<uvm_reg*> reg_a,
                uvm_reg_block* blk_parent,
                uvm_reg_file* regfile_parent = NULL );
```

The member function **configure** shall configure the indirect data register. The argument *idx* register specifies the index, in the *reg_a* register array, of the register to access. The *idx* needs to be written first. A read or write operation to this register shall 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 the number of bits of this register.

15.8 uvm_reg_fifo

The class **uvm_reg_fifo** defines a special register to model a DUT FIFO accessed via write/read, where writes push to the FIFO and reads pop from it. Backdoor 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.

15.8.1 Class definition

```
namespace uvm {

  class uvm_reg_fifo : public uvm_reg
  {
   public:

    // Constructor
    uvm_reg_fifo( const std::string& name,
                  unsigned int size,
                  unsigned int n_bits,
                  int has_cover);

    // Group: Initialization

    void set_compare( uvm_check_e check = UVM_CHECK );

    // Group: Introspection

    unsigned int size();
    unsigned int capacity();

    // Group: Access

    virtual void write( uvm_status_e& status,
                        uvm_reg_data_t value,
                        uvm_path_e path = UVM_DEFAULT_PATH,
```

```
                         uvm_reg_map* map = NULL,
                         uvm_sequence_base* parent = NULL,
                         int prior = -1,
                         uvm_object* extension = NULL,
                         const std::string& fname = "",
                         int lineno = 0 );

    virtual void read( uvm_status_e& status,
                        uvm_reg_data_t& value,
                        uvm_path_e path = UVM_DEFAULT_PATH,
                        uvm_reg_map* map = NULL,
                        uvm_sequence_base* parent = NULL,
                        int prior = -1,
                        uvm_object* extension = NULL,
                        const std::string& fname = "",
                        int lineno = 0 );

    virtual void set( uvm_reg_data_t  value,
                       const std::string& fname = "",
                       int lineno = 0 );

    virtual void update( uvm_status_e& status,
                          uvm_path_e path = UVM_DEFAULT_PATH,
                          uvm_reg_map* map = NULL,
                          uvm_sequence_base* parent = NULL,
                          int prior = -1,
                          uvm_object* extension = NULL,
                          const std::string& fname = "",
                          int lineno = 0 );

    virtual void mirror( uvm_status_e& status,
                          uvm_check_e check  = UVM_NO_CHECK,
                          uvm_path_e path = UVM_DEFAULT_PATH,
                          uvm_reg_map* map = NULL,
                          uvm_sequence_base* parent = NULL,
                          int prior = -1,
                          uvm_object* extension = NULL,
                          const std::string& fname = "",
                          int lineno = 0 );

    virtual uvm_reg_data_t get( const std::string& fname = "",
                                 int lineno = 0 ) const;

    virtual void do_predict( uvm_reg_item* rw,
                              uvm_predict_e kind = UVM_PREDICT_DIRECT,
                              uvm_reg_byte_en_t be = -1 );

    // Group: Special overrides

    virtual void pre_write( uvm_reg_item* rw );
    virtual void pre_read( uvm_reg_item* rw );

    // Data members

    std::vector<uvm_reg_data_t> fifo;

  }; // class uvm_reg_fifo

} // namespace uvm
```

15.8.2 Constructor

```
uvm_reg_fifo( const std::string& name,
              unsigned int size,
              unsigned int n_bits,
              int has_cover);
```

The constructor shall create an instance of a FIFO register with the specified *name*, having *size* elements of *n_bits* each.

15.8.3 Initialization

15.8.3.1 set_compare

```
void set_compare( uvm_check_e check = UVM_CHECK );
```

The member function **set_compare** shall specify the comparison 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 and the comparison policy for the field is **UVM_CHECK**.

15.8.4 Introspection

15.8.4.1 size

```
unsigned int size();
```

The member function **size** shall return the number of entries currently in the FIFO.

15.8.4.2 capacity

```
unsigned int capacity();
```

The member function **capacity** shall return the maximum number of entries, or depth, of the FIFO.

15.8.5 Access

15.8.5.1 write

```
virtual void write( uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    int prior = -1,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **write** shall write 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**), 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. If the write is via an **update** operation, the abstract FIFO already contains the written value and is thus not affected by either prediction mode.

15.8.5.2 read

```
virtual void read( uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   int prior = -1,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall reads and return the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped.

15.8.5.3 set

```
virtual void set( uvm_reg_data_t  value,
                  const std::string& fname = "",
                  int lineno = 0 );
```

The member function **set** shall write the given value to the abstract FIFO. An application may call this member function several times before an **update** as a means of preloading the DUT FIFO. Calls to **set** to a full FIFO are ignored. An application should call **update** to update the DUT FIFO with the set values.

15.8.5.4 update

```
virtual void update( uvm_status_e& status,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **update** shall write all values preloaded using the member function **set** to the DUT. An application should call **update** after **set** before any blocking statements, else other reads/writes to the DUT FIFO may cause the mirror to become out of sync with the DUT.

15.8.5.5 mirror

```
virtual void mirror( uvm_status_e& status,
                     uvm_check_e check  = UVM_NO_CHECK,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **mirror** shall read the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped. If the check argument is set and comparison is enabled with **set_compare**.

15.8.5.6 get

```
virtual uvm_reg_data_t get( const std::string& fname = "",
                            int lineno = 0 ) const;
```

The member function **get** shall return the next value from the abstract FIFO, but does not pop it. It is used to get the expected value in a **mirror** operation.

15.8.5.7 do_predict

```
virtual void do_predict( uvm_reg_item* rw,
                         uvm_predict_e kind = UVM_PREDICT_DIRECT,
                         uvm_reg_byte_en_t be = -1 );
```

The member function **do_predict** shall update the abstract (mirror) FIFO based on **write** and **read** operations. When autoprediction is on, this member function is called before each read, write, peek, or poke operation returns. When auto-prediction is off, this member function is called by a **uvm_reg_predictor** upon receipt and conversion of an observed bus operation to this register.

If a write prediction, the observed write value is pushed to the abstract FIFO as long as it is not full and the operation did not originate from an **update**. If a read prediction, the observed read value is compared with the frontmost value in the abstract FIFO if **set_compare** enabled comparison and the FIFO is not empty.

15.8.6 Special overrides

15.8.6.1 pre_write

```
virtual void pre_write( uvm_reg_item* rw );
```

The member function **pre_write** shall be called before a FIFO **write** or **update**.

It is an error to attempt a write to a full FIFO or a write while an update is still pending. An update is pending after one or more calls to **set**. If an application allows the DUT to write to a full FIFO, the application should override **pre_write** as appropriate.

15.8.6.2 pre_read

```
virtual void pre_read( uvm_reg_item* rw );
```

The member function **pre_read** shall be called before register **read** or **update**.

It aborts the operation if the internal FIFO is empty. If in an application the DUT does not behave this way, the application should override **pre_read** as appropriate.

15.8.7 Data members

15.8.7.1 fifo

```
std::vector<uvm_reg_data_t> fifo;
```

The data member**fifo** shall define the abstract representation of the FIFO, with the constrained to be no larger than the size parameter. This data member is public to enable subtypes to add constraints on it and randomize.

15.9 uvm_vreg

The class **uvm_vreg** shall define 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.

15.9.1 Class definition

```
namespace uvm {

  class uvm_vreg : public uvm_object
  {
   public:
```

```
// Constructor
explicit uvm_vreg( const std::string& name, unsigned int n_bits );

// Group: Initialization

void configure( uvm_reg_block* parent,
                uvm_mem* mem = NULL,
                unsigned long size   = 0,
                uvm_reg_addr_t offset = 0,
                unsigned int incr   = 0);

virtual bool implement( unsigned long n,
                        uvm_mem* mem = NULL,
                        uvm_reg_addr_t offset = 0,
                        unsigned int incr = 0);

virtual uvm_mem_region* allocate( unsigned long n,
                                  uvm_mem_mam* mam );

virtual uvm_mem_region* get_region() const;
virtual void release_region();

// Group: Introspection

virtual const std::string get_name() const;
virtual const std::string get_full_name() const;
virtual uvm_reg_block* get_parent() const;
virtual uvm_mem* get_memory() const;
virtual int get_n_maps() const;
bool is_in_map( uvm_reg_map* map ) const;
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
virtual std::string get_rights( uvm_reg_map* map = NULL ) const;
virtual std::string get_access( uvm_reg_map* map = NULL ) const;
virtual unsigned int get_size() const;
virtual unsigned int get_n_bytes() const;
virtual unsigned int get_n_memlocs() const;
virtual unsigned int get_incr() const;
virtual void get_fields( std::vector<uvm_vreg_field*>& fields) const;
virtual uvm_vreg_field* get_field_by_name( const std::string& name ) const;
virtual uvm_reg_addr_t get_offset_in_memory( unsigned long idx ) const;

virtual uvm_reg_addr_t get_address( unsigned long idx,
                                    const uvm_reg_map* map = NULL ) const;

// Group: HDL Access

virtual void write( unsigned long idx,
                    uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );

virtual void read( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void poke( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t value,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );

virtual void peek( unsigned long idx,
                   uvm_status_e& status,
```

```
                        uvm_reg_data_t& value,
                        uvm_sequence_base* parent = NULL,
                        uvm_object* extension = NULL,
                        const std::string& fname = "",
                        int lineno = 0 );

    void reset( const std::string& kind = "HARD" );

    // Group: Callbacks

    virtual void pre_write( unsigned long idx,
                             uvm_reg_data_t& wdat,
                             uvm_path_e& path,
                             uvm_reg_map*& map );

    virtual void post_write( unsigned long idx,
                              uvm_reg_data_t wdat,
                              uvm_path_e path,
                              uvm_reg_map* map,
                              uvm_status_e& status);

    virtual void pre_read( unsigned long idx,
                            uvm_path_e& path,
                            uvm_reg_map*& map );

    virtual void post_read( unsigned long idx,
                             uvm_reg_data_t& rdat,
                             uvm_path_e path,
                             uvm_reg_map* map,
                             uvm_status_e& status);

  }; // class uvm_vreg

} // namespace uvm
```

15.9.2 Constructor

```
explicit uvm_vreg( const std::string& name, unsigned int n_bits );
```

The constructor shall reate an instance of a virtual register abstraction class with the specified *name*. The argument *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.

15.9.3 Initialization

15.9.3.1 configure

```
void configure( uvm_reg_block* parent,
                uvm_mem* mem = NULL,
                unsigned long size   = 0,
                uvm_reg_addr_t offset = 0,
                unsigned int incr = 0);
```

The member function **configure** shall specify the parent block of this virtual register array. If one of the other parameters are specified, the virtual register is assumed to be dynamic and can be later (re-)implemented using the member function **uvm_vreg::implement**. If argument *mem* is specified, then the virtual register array is assumed to be statically implemented in the memory corresponding to the specified memory abstraction class and *size*, *offset* and *incr* also needs to be specified. Static virtual register arrays cannot be reimplemented.

15.9.3.2 implement

```
virtual bool implement( unsigned long n,
                        uvm_mem* mem = NULL,
                        uvm_reg_addr_t offset = 0,
                        unsigned int incr = 0);
```

The member function **implement** shall implement 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.

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.

The member function shall return true if the memory can implement the number of virtual registers at the specified base offset and offset increment. Returns FALSE otherwise.

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.

15.9.3.3 allocate

```
virtual uvm_mem_region* allocate( unsigned long n,
                                  uvm_mem_mam* mam );
```

The member function **allocate** shall implement 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 member function **uvm_mem_mam::request_region**.

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.

The meber function shall return a reference to a **uvm_mem_region** memory region descriptor 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 shall returns NULL.

A region implementing a virtual register array shall not be released using the member function **uvm_mem_mam::release_region**. It shall be released using the member function **uvm_vreg::release_region**.

15.9.3.4 get_region

```
virtual uvm_mem_region* get_region() const;
```

The member function **get_region** shall return a reference to the **uvm_mem_region** memory region descriptor that implements the virtual register array. The member function shall return NULL if the virtual registers array is not currently implemented. A region implementing a virtual register array shall not be released using the member function **uvm_mem_mam::release_region**, but shall be released using the member function **uvm_vreg::release_region**.

15.9.3.5 release_region

```
virtual void release_region();
```

The member function **release_region** shall release the memory region used to implement a virtual register array and return 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.

15.9.4 Introspection

15.9.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this register.

15.9.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this register. The base of the hierarchical name is the root block.

15.9.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent block.

15.9.4.4 get_memory

```
virtual uvm_mem* get_memory() const;
```

The member function **get_memory** shall return the memory where the virtual register array is implemented.

15.9.4.5 get_n_maps

```
virtual int get_n_maps() const;
```

The member function **get_n_maps** shall return the number of address maps this virtual register array is mapped in.

15.9.4.6 is_in_map

```
bool is_in_map( uvm_reg_map* map ) const;
```

The member function **is_in_map** shall return true if this virtual register array is in the specified address map, otherwise return false.

15.9.4.7 get_maps

```
virtual void get_maps( std::vector<uvm_reg_map*>& maps ) const;
```

The member function **get_maps** shall return all of the address maps where this virtual register array is mapped.

15.9.4.8 get_rights

```
virtual std::string get_rights( uvm_reg_map* map = NULL ) const;
```

The member function **get_rights** shall return the accessibility (“RW, “RO”, or “WO”) of this virtual register array.

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 is issued and “RW” is returned.

15.9.4.9 get_access

```
virtual std::string get_access( const uvm_reg_map* map = NULL ) const;
```

The member function **get_access** shall return 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 needs to 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 a domain with read-only restrictions would return “RO”.

15.9.4.10 get_size

```
unsigned int get_size() const;
```

The member function **get_size** shall return the size of the virtual register array.

15.9.4.11 get_n_bytes

```
unsigned int get_n_bytes() const;
```

The member function **get_n_bytes** shall return 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-bytes memory locations is 4-byte wide.

15.9.4.12 get_n_memlocs

```
virtual unsigned int get_n_memlocs() const;
```

The member function **get_n_memlocs** shall return the number of memory locations used by a single virtual register.

15.9.4.13 get_incr

```
virtual unsigned int get_incr() const;
```

The member function **get_incr** shall return the number of memory locations between two individual virtual registers in the same array.

15.9.4.14 get_fields

```
virtual void get_fields( std::vector<uvm_vreg_field*>& fields) const;
```

The member function **get_fields** shall return the virtual fields in this virtual register. Fields are ordered from least-significant position to most-significant position within the register.

15.9.4.15 get_field_by_name

```
virtual uvm_vreg_field* get_field_by_name( const std::string& name ) const;
```

The member function **get_field_by_name** shall return the named virtual field in this virtual register. The member function shall find a virtual field with the specified name in this register and returns its abstraction class. If no fields are found, it returns NULL.

15.9.4.16 get_offset_in_memory

```
virtual uvm_reg_addr_t get_offset_in_memory( unsigned long idx ) const;
```

The member function **get_offset_in_memory** shall return the base offset of the specified virtual register, in the overall address space of the memory that implements the virtual register array.

15.9.4.17 get_address

```
virtual uvm_reg_addr_t get_address( unsigned long idx,
                                    const uvm_reg_map* map = NULL ) const;
```

The member function **get_address** shall return 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, an error message is issued.

15.9.5 HDL access

15.9.5.1 write

```
virtual void write( unsigned long idx,
                    uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
```

```
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );
```

The member function **write** shall write the specified *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path*.

If the memory implementing the virtual register array is mapped in more than one address map, an address map shall be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

15.9.5.2 read

```
virtual void read( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall read from the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path* and return the readback *value*.

If the memory implementing the virtual register array is mapped in more than one address *map*, an address map shall be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-read operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

15.9.5.3 poke

```
virtual void poke( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t value,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **poke** shall deposit the specified *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access.

The operation is eventually mapped into set of memory-poke operations at the location where the virtual register specified by idx in the virtual register array is implemented.

15.9.5.4 peek

```
virtual void peek( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
```

```
                    int lineno = 0 );
```

The member function **peek** shall sample the current value in a virtual register.

It samples the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access, and return the sampled *value*. The operation is eventually mapped into set of memory-peek operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

15.9.5.5 reset

```
void reset( const std::string& kind = "HARD" );
```

The member function **reset** shall reset 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 in before the access was completed

15.9.6 Callbacks

15.9.6.1 pre_write

```
virtual void pre_write( unsigned long idx,
                        uvm_reg_data_t& wdat,
                        uvm_path_e& path,
                        uvm_reg_map*& map );
```

The member function **pre_write** shall be 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 shall be used to perform the virtual register operation. The registered callback methods are invoked after the invocation of this member function. 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.

15.9.6.2 post_write

```
virtual void post_write( unsigned long idx,
                         uvm_reg_data_t wdat,
                         uvm_path_e path,
                         uvm_reg_map* map,
                         uvm_status_e& status);
```

The member function **post_write** shall be called after virtual register write.

If the specified status is modified, the updated status shall be returned by the virtual register operation. The registered callback methods are invoked before the invocation of this member function. 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.

15.9.6.3 pre_read

```
virtual void pre_read( unsigned long idx,
                           uvm_path_e& path,
                           uvm_reg_map*& map );
```

The member function **pre_read** shall be called before virtual register read.

If the specified access *path* or address *map* are modified, the updated access path or address map shall be used to perform the register operation. The registered callback methods are invoked after the invocation of this member function. 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.

15.9.6.4 post_read

```
virtual void post_read( unsigned long idx,
                        uvm_reg_data_t& rdat,
                        uvm_path_e path,
                        uvm_reg_map* map,
                        uvm_status_e& status);
```

The member function **post_read** shall be called after virtual register read.

If the specified readback data or *status* is modified, the updated readback data or status shall be returned by the register operation. The registered callback methods are invoked before the invocation of this member function. 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.

15.10 uvm_vreg_cbs

The class **uvm_vreg_cbs** shall define virtual register facade class.

15.10.1 Member functions

15.10.1.1 pre_write

```
virtual void pre_write( uvm_vreg* rg,
                        unsigned long idx,
                        uvm_reg_data_t& wdat,
                        uvm_path_e& path,
                        uvm_reg_map*& map );
```

The member function **pre_write** shall be called before virtual register write.

The registered callback methods are invoked after the invocation of the member function **uvm_vreg::pre_write**. 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.

The written value *wdat*, access *path* and address *map*, if modified, modifies the actual value, access path or address map used in the virtual register operation.

15.10.1.2 post_write

```
virtual void post_write( uvm_vreg* rg,
                         unsigned long idx,
                         uvm_reg_data_t wdat,
                         uvm_path_e path,
                         uvm_reg_map* map,
                         uvm_status_e& status);
```

The member function **post_write** shall be called after virtual register write.

The registered callback methods are invoked before the invocation of the member function **uvm_reg::post_write**. All 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.

The *status* of the operation, if modified, modifies the actual returned status.

15.10.1.3 pre_read

```
virtual void pre_read( uvm_vreg* rg,
                       unsigned long idx,
                       uvm_path_e& path,
                       uvm_reg_map*& map );
```

The member function **pre_read** shall be called before virtual register read.

The registered callback methods are invoked after the invocation of the member function **uvm_reg::pre_read**. All 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.

The access *path* and address *map*, if modified, modifies the actual access path or address map used in the register operation.

15.10.1.4 post_read

```
virtual void post_read( uvm_vreg* rg,
                        unsigned idx,
                        uvm_reg_data_t& rdat,
                        uvm_path_e path,
                        uvm_reg_map* map,
                        uvm_status_e& status);
```

The member function **post_read** shall be called after virtual register read.

The registered callback methods are invoked before the invocation of the member function **uvm_reg::post_read**. All 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.

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

15.11 uvm_vreg_field

The class **uvm_vreg_field** shall define the virtual field abstraction class. A virtual field represents a set of adjacent bits that are logically implemented in consecutive memory locations.

15.11.1 Class definition

```
namespace uvm {

  class uvm_vreg_field : public uvm_object
  {
   public:

    // Constructor
    explicit uvm_vreg_field( const std::string& name = "uvm_vreg_field" );
```

```
  // Group: Initialization

  void configure( uvm_vreg* parent,
                  unsigned int size,
                  unsigned int lsb_pos );

  // Group: Introspection

  virtual cnst std::string get_name() const;
  virtual const std::string get_full_name() const;
  virtual uvm_vreg* get_parent() const;
  virtual unsigned int get_lsb_pos_in_register() const;
  virtual unsigned int get_n_bits() const;
  virtual std::string get_access( uvm_reg_map* map = NULL ) const;

  // Group: HDL access

  virtual void write( unsigned long idx,
                      uvm_status_e&  status,
                      uvm_reg_data_t value,
                      uvm_path_e path = UVM_DEFAULT_PATH,
                      uvm_reg_map* map = NULL,
                      uvm_sequence_base*  parent = NULL,
                      uvm_object*         extension = NULL,
                      const std::string& fname = "",
                      int lineno = 0 );

  virtual void read( unsigned long idx,
                     uvm_status_e& status,
                     uvm_reg_data_t& value,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );

  virtual void poke( unsigned long idx,
                     uvm_status_e& status,
                     uvm_reg_data_t value,
                     uvm_sequence_base* parent = NULL,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );

  virtual void peek( unsigned long idx,
                     uvm_status_e& status,
                     uvm_reg_data_t& value,
                     uvm_sequence_base* parent = NULL,
                     uvm_object* extension = NULL,
                     const std::string& fname = "",
                     int lineno = 0 );

  // Group: Callbacks

  virtual void pre_write( unsigned long idx,
                          uvm_reg_data_t& wdat,
                          uvm_path_e& path,
                          uvm_reg_map*& map);

  virtual void post_write( unsigned long idx,
                           uvm_reg_data_t wdat,
                           uvm_path_e path,
                           uvm_reg_map* map,
                           uvm_status_e& status );

  virtual void pre_read( unsigned long idx,
                         uvm_path_e& path,
                         uvm_reg_map*& map);

  virtual void post_read( unsigned long idx,
                          uvm_reg_data_t& rdat,
                          uvm_path_e path,
                          uvm_reg_map* map,
                          uvm_status_e& status);

}; // class uvm_vreg_field
```

```
} // namespace uvm
```

15.11.2 Constructor

```
explicit uvm_vreg_field( const std::string& name = "uvm_vreg_field" );
```

The constructor shall reate an instance of a virtual field instance with the specified *name*. The constructor shall not be called directly. An application shall use the **uvm_vreg_field::type_id::create** member function instead.

15.11.3 Initialization

15.11.3.1 configure

```
void configure( uvm_vreg* parent,
                unsigned int size,
                unsigned int lsb_pos );
```

The member function **configure** shall specify the *parent* virtual register of this virtual field, its *size* in bits, and the position of its least-significant bit *lsb_pos* within the virtual register relative to the least-significant bit of the virtual register.

15.11.4 Introspection

15.11.4.1 get_name

```
virtual const std::string get_name() const;
```

The member function **get_name** shall return the simple object name of this this virtual field.

15.11.4.2 get_full_name

```
virtual const std::string get_full_name() const;
```

The member function **get_full_name** shall return the hierarchal name of this virtual field. The base of the hierarchical name is the root block.

15.11.4.3 get_parent

```
virtual uvm_reg_block* get_parent() const;
```

The member function **get_parent** shall return the parent virtual register.

15.11.4.4 get_lsb_pos_in_register

```
virtual unsigned int get_lsb_pos_in_register() const;
```

The member function **get_lsb_pos_in_register** shall return the index of the least significant bit of the virtual field in the virtual register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register.

15.11.4.5 get_n_bits

```
virtual unsigned int get_n_bits() const;
```

The member function **get_n_bits** shall return the width, in bits, of the virtual field.

15.11.4.6 get_access

```
virtual std::string get_access( const uvm_reg_map* map = NULL ) const;
```

The member function **get_access** shall return 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 would return “RO”.

15.11.5 HDL access

15.11.5.1 write

```
virtual void write( unsigned long idx,
                    uvm_status_e& status,
                    uvm_reg_data_t value,
                    uvm_path_e path = UVM_DEFAULT_PATH,
                    uvm_reg_map* map = NULL,
                    uvm_sequence_base* parent = NULL,
                    uvm_object* extension = NULL,
                    const std::string& fname = "",
                    int lineno = 0 );
```

The member function **write** shall write the specified *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory read-modify-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented. If a backdoor is available for the memory implementing the virtual field, it shall be used for the memory-read operation.

15.11.5.2 read

```
virtual void read( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_path_e path = UVM_DEFAULT_PATH,
                   uvm_reg_map* map = NULL,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **read** shall read from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*, and return the readback *value*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory read operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

15.11.5.3 poke

```
virtual void poke( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t value,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **poke** shall deposit the specified *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access path.

The operation is eventually mapped into memory peek-modify-poke operations at the location where the virtual register specified by idx in the virtual register array is implemented.

15.11.5.4 peek

```
virtual void peek( unsigned long idx,
                   uvm_status_e& status,
                   uvm_reg_data_t& value,
                   uvm_sequence_base* parent = NULL,
                   uvm_object* extension = NULL,
                   const std::string& fname = "",
                   int lineno = 0 );
```

The member function **peek** shall sample from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*, and return the readback *value*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* shall be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory peek operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

15.11.6 Callbacks

15.11.6.1 pre_write

```
virtual void pre_write( unsigned long idx,
                        uvm_reg_data_t& wdat,
                        uvm_path_e& path,
                        uvm_reg_map*& map);
```

The member function **pre_write** shall be 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 shall be used to perform the virtual register operation.

The virtual field callback member functions are invoked before the callback member functions on the containing virtual register. The registered callback member functions are invoked after the invocation of this member function. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks.

15.11.6.2 post_write

```
virtual void post_write( unsigned long idx,
                         uvm_reg_data_t wdat,
                         uvm_path_e path,
                         uvm_reg_map* map,
                         uvm_status_e& status );
```

The member function **post_write** shall be called after virtual field write.

If the specified *status* is modified, the updated status shall be returned by the virtual register operation.

The virtual field callback member functions are invoked after the callback member functions on the containing virtual register. The registered callback member functions are invoked before the invocation of this member function. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks.

15.11.6.3 pre_read

```
virtual void pre_read( unsigned long idx,
                       uvm_path_e& path,
                       uvm_reg_map*& map);
```

The member function **pre_read** shall be called before virtual field read.

If the specified access *path* or address *map* are modified, the updated access path or address map shall be used to perform the virtual register operation.

The virtual field callback member functions are invoked after the callback member functions on the containing virtual register. The registered callback member functions are invoked after the invocation of this member function. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

15.11.6.4 post_read

```
virtual void post_read( unsigned long idx,
                        uvm_reg_data_t& rdat,
                        uvm_path_e path,
                        uvm_reg_map* map,
                        uvm_status_e& status);
```

The member function **post_read** shall be called after virtual register read.

If the specified readback data *rdat* or *status* is modified, the updated readback data or status shall be returned by the virtual register operation.

The virtual field callback member functions are invoked after the callback member functions on the containing virtual register. The registered callback member functions are invoked before the invocation of this member function. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks.

15.12 uvm_vreg_field_cbs

The class **uvm_vreg_field_cbs** shall define virtual fields facade class.

15.12.1 Class definition

```
namespace uvm {

  class uvm_vreg_field_cbs : public uvm_callback
  {
   public:
    virtual void pre_write( uvm_vreg_field* field,
                            unsigned long idx,
                            uvm_reg_data_t& wdat,
                            uvm_path_e& path,
                            uvm_reg_map*& map );

    virtual void post_write( uvm_vreg_field* field,
                             unsigned long idx,
                             uvm_reg_data_t wdat,
                             uvm_path_e path,
                             uvm_reg_map* map,
                             uvm_status_e& status );

    virtual void pre_read( uvm_vreg_field* field,
                           unsigned long idx,
                           uvm_path_e& path,
                           uvm_reg_map*& map );

    virtual void post_read( uvm_vreg_field* field,
                            unsigned long idx,
                            uvm_reg_data_t& rdat,
                            uvm_path_e path,
                            uvm_reg_map* map,
                            uvm_status_e& status );

  }; // class uvm_vreg_field_cbs

} // namespace uvm
```

15.12.2 Member functions

15.12.2.1 pre_write

```
virtual void pre_write( uvm_vreg_field* field,
                        unsigned long idx,
                        uvm_reg_data_t& wdat,
                        uvm_path_e& path,
                        uvm_reg_map*& map );
```

The member function **pre_write** shall be called before virtual field write.

The registered callback member functions are invoked before the invocation of the virtual register pre-write callbacks and after the invocation of the member function **uvm_vreg_field::pre_write**.

The written value *wdat*, access *path* and address *map*, if modified, modifies the actual value, access path or address map used in the register operation.

15.12.2.2 post_write

```
virtual void post_write( uvm_vreg_field* field,
                         unsigned long idx,
                         uvm_reg_data_t wdat,
                         uvm_path_e path,
                         uvm_reg_map* map,
```

```
                          uvm_status_e& status );
```

The member function **post_write** shall be called after virtual field write.

The registered callback member functions are invoked after the invocation of the virtual register post-write callbacks and before the invocation of the member function **uvm_vreg_field::post_write**.

The *status* of the operation, if modified, modifies the actual returned status.

15.12.2.3 pre_read

```
virtual void pre_read( uvm_vreg_field* field,
                       unsigned long idx,
                       uvm_path_e& path,
                       uvm_reg_map*& map );
```

The member function **pre_read** shall be called before virtual field read.

The registered callback member functions are invoked after the invocation of the virtual register pre-read callbacks and after the invocation of the member function **uvm_vreg_field::pre_read**.

The access *path* and address *map*, if modified, modifies the actual access path or address map used in the register operation.

15.12.2.4 post_read

```
virtual void post_read( uvm_vreg_field* field,
                        unsigned long idx,
                        uvm_reg_data_t& rdat,
                        uvm_path_e path,
                        uvm_reg_map* map,
                        uvm_status_e& status );
```

The member function **post_read** shall be called after virtual field read.

The registered callback member functions are invoked after the invocation of the virtual register post-read callbacks and before the invocation of the member function **uvm_vreg_field::post_read**.

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

15.13 uvm_reg_cbs

The class **uvm_reg_cbs** shall define the facade class for field, register, memory and backdoor access callback member functions.

15.13.1 Class definition

```
namespace uvm {

  class uvm_reg_cbs : public uvm_callback
  {
   public:

    virtual void pre_write( uvm_reg_item* rw);
    virtual void post_write( uvm_reg_item* rw );
    virtual void pre_read( uvm_reg_item* rw );
```

```
    virtual void post_read( uvm_reg_item* rw );

    virtual void post_predict( uvm_reg_field* fld,
                               uvm_reg_data_t previous,
                               uvm_reg_data_t value,
                               uvm_predict_e  kind,
                               uvm_path_e     path,
                               uvm_reg_map*   map );

    virtual void encode( std::vector<uvm_reg_data_t>& data );
    virtual void decode( std::vector<uvm_reg_data_t>& data );

  }; // class uvm_reg_cbs

} // namespace uvm
```

15.13.2 Member functions

15.13.2.1 pre_write

```
virtual void pre_write( uvm_reg_item* rw);
```

The member function **pre_write** shall be called before a write operation.

All registered **pre_write** callback member functions are invoked after the invocation of the member function **pre_write** of associated object (**uvm_reg**, **uvm_reg_field**, **uvm_mem**, or **uvm_reg_backdoor**). If the element being written is a **uvm_reg**, all **pre_write** callback member functions are invoked before the contained **uvm_reg_fields**.

— Backdoor: **uvm_reg_backdoor::pre_write**, **uvm_reg_cbs::pre_write** callbacks for backdoor.
— Register: **uvm_reg::pre_write**, **uvm_reg_cbs::pre_write** callbacks for reg, then for each field: **uvm_reg_field::pre_write**, **uvm_reg_cbs::pre_write** callbacks for field.
— RegField: **uvm_reg_field::pre_write**, **uvm_reg_cbs::pre_write** callbacks for field
— Memory: **uvm_mem::pre_write**, **uvm_reg_cbs::pre_write** callbacks for mem.

The argument *rw* 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-backdoor 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**, the operation is aborted. See Section 16.1 for details on *rw* information.

15.13.2.2 post_write

```
virtual void post_write( uvm_reg_item* rw );
```

The member function **post_write** shall be called after a write operation.

All registered **post_write** callback member functions are invoked before the invocation of the member function **post_write** of the associated object (**uvm_reg**, **uvm_reg_field**, **uvm_mem**, or **uvm_reg_backdoor**). If the element being written is a **uvm_reg**, all **post_write** callback member functions are invoked before the contained **uvm_reg_fields**.

— Backdoor: **uvm_reg_cbs::post_write** callbacks for backdoor, **uvm_reg_backdoor::post_write**.

— Register **uvm_reg_cbs::post_write** callbacks for reg, **uvm_reg::post_write**, then for each field: **uvm_reg_cbs::post_write** callbacks for field, **uvm_reg_field::post_read**.

— RegField **uvm_reg_cbs::post_write** callbacks for field, **uvm_reg_field::post_write**.

— Memory **uvm_reg_cbs::post_write** callbacks for mem, **uvm_mem::post_write**.

The argument *rw* 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 Section 16.1 for details on *rw* information.

15.13.2.3 pre_read

```
virtual void pre_read( uvm_reg_item* rw );
```

The member function **pre_read** shall be called before a read operation.

All registered **pre_read** callback member functions are invoked after the invocation of the **pre_read** member function of associated object (**uvm_reg**, **uvm_reg_field**, **uvm_mem**, or **uvm_reg_backdoor**). If the element being read is a **uvm_reg**, all **pre_read** callback member functions are invoked before the contained **uvm_reg_fields**.

— Backdoor: **uvm_reg_backdoor::pre_read**, **uvm_reg_cbs::pre_read** callbacks for backdoor.

— Register: **uvm_reg::pre_read**, **uvm_reg_cbs::pre_read** callbacks for reg, then for each field: **uvm_reg_field::pre_read**, **uvm_reg_cbs::pre_read** callbacks for field.

— RegField: **uvm_reg_field::pre_read**, **uvm_reg_cbs::pre_read** callbacks for field.

— Memory: **uvm_mem::pre_read**, **uvm_reg_cbs::pre_read** callbacks for mem.

The argument *rw* holds information about the operation.

— The value member of rw is not used has no effect if modified.

— For memories, modifying the offset modifies the offset used in the operation.

— For non-backdoor 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**, the operation is aborted.

See Section 16.1 for details on *rw* information.

15.13.2.4 post_read

```
virtual void post_read( uvm_reg_item* rw );
```

The member function **post_read** shall be called after a read operation.

All registered **post_read** callback member functions are invoked before the invocation of the member function **post_read** of the associated object (**uvm_reg**, **uvm_reg_field**, **uvm_mem**, or **uvm_reg_backdoor**). If the element being read is a **uvm_reg**, all post_read callback member functions are invoked before the contained **uvm_reg_fields**.

— Backdoor **uvm_reg_cbs::post_read** callbacks for backdoor, **uvm_reg_backdoor::post_read**.

— Register: **uvm_reg_cbs::post_read** callbacks for reg, **uvm_reg::post_read**, then for each field: **uvm_reg_cbs::post_read** callbacks for field, **uvm_reg_field::post_read**.
— RegField: **uvm_reg_cbs::post_read** callbacks for field, **uvm_reg_field::post_read**.
— Memory: **uvm_reg_cbs::post_read** callbacks for mem, **uvm_mem::post_read**.
—

The argument *rw* holds information about the operation.

— Modifying the readback value or status modifies the actual returned value and status.
— Modifying the value or offset members has no effect, as the operation has already completed.

See Section 16.1 for details on *rw* information.

15.13.2.5 post_predict

```
virtual void post_predict( uvm_reg_field* fld,
                           uvm_reg_data_t previous,
                           uvm_reg_data_t value,
                           uvm_predict_e  kind,
                           uvm_path_e     path,
                           uvm_reg_map*   map );
```

The member function **post_predict** shall be called by the member function **uvm_reg_field::predict** after a successful **UVM_PREDICT_READ** or **UVM_PREDICT_WRITE** prediction. The argument *previous* is the previous value in the mirror and the argument *value* is the latest predicted value. Any change to *value* shall modify the predicted mirror value.

15.13.2.6 encode

```
virtual void encode( std::vector<uvm_reg_data_t>& data );
```

The member function **encode** shall encode the data.

The registered callback member functions are invoked in order of registration after all the member functions **pre_write** have been called. The encoded data is passed through each invocation in sequence. This allows the member functions **pre_write** to deal with clear-text data.

By default, the data is not modified.

15.13.2.7 decode

```
virtual void decode( std::vector<uvm_reg_data_t>& data );
```

The member function **decode** shall decode the data.

The registered callback member functions are invoked in reverse order of registration before all the member functions **post_read** are called. The decoded data is passed through each invocation in sequence. This allows the member functions **post_read** 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.

15.14 uvm_mem_mam

The class **uvm_mem_mam** manages 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.

15.14.1 Class definition

```
namespace uvm {

  class uvm_mem_mam
  {
   public:

    // Constructor

    explicit uvm_mem_mam( const std::string& name,
                          uvm_mem_mam_cfg* cfg,
                          uvm_mem* mem = NULL );

    // Group: Initialization

    uvm_mem_mam_cfg* reconfigure( uvm_mem_mam_cfg* cfg = NULL );

    // Group: Memory Management

    uvm_mem_region* reserve_region( unsigned long start_offset,
                                    unsigned int n_bytes,
                                    const std::string& fname = "",
                                    int lineno = 0 );

    uvm_mem_region* request_region( unsigned int n_bytes,
                                    uvm_mem_mam_policy* alloc = NULL,
                                    const std::string& fname = "",
                                    int lineno = 0 );

    void release_region( uvm_mem_region* region );
    void release_all_regions();

    // Group: Introspection

    std::string convert2string();
    uvm_mem_region* for_each( bool reset = false );
    uvm_mem* get_memory() const;

    // Data members

    uvm_mem_mam_policy* default_alloc;

    // Type definitions

    typedef enum { GREEDY, THRIFTY } alloc_mode_e;
    typedef enum { BROAD, NEARBY }   locality_e;

  }; // class uvm_mem_mam

} // namespace uvm
```

15.14.2 Constructor

```
explicit uvm_mem_mam( const std::string& name,
                      uvm_mem_mam_cfg* cfg,
                      uvm_mem* mem = NULL );
```

The constructor shall create an instance of a memory allocation manager with the specified *name* and configuration *cfg*. 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 member functions **uvm_mem_region::read** and **uvm_mem_region::write**.

15.14.3 Initialization

15.14.3.1 reconfigure

```
uvm_mem_mam_cfg* reconfigure( uvm_mem_mam_cfg* cfg = NULL );
```

The member function **reconfigure** shall modify 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 shall fall within the new address space.

The member function shall return the previous configuration.

If no new configuration is specified, it shall return the current configuration.

15.14.4 Memory management

15.14.4.1 reserve_region

```
uvm_mem_region* reserve_region( unsigned long start_offset,
                                unsigned int n_bytes,
                                const std::string& fname = "",
                                int lineno = 0 );
```

The member function **reserve_region** shall reserve a 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, the member function shall return NULL.

It shall 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.

15.14.4.2 request_region

```
uvm_mem_region* request_region( unsigned int n_bytes,
                                uvm_mem_mam_policy* alloc = NULL,
                                const std::string& fname = "",
                                int lineno = 0 );
```

The member function **request_region** shall request and reserve a memory region of the specified number of bytes starting at a random location. If an 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 is randomized.

A descriptor of the allocated region is returned. If no region can be allocated, the member function shall return NULL.

It shall 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 because 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.

15.14.4.3 release_region

```
void release_region( uvm_mem_region* region );
```

The member function **release_region** shall release a previously allocated memory region. An error is issued if the specified region has not been previously allocated or is no longer allocated.

15.14.4.4 release_all_regions

```
void release_all_regions();
```

The member function **release_all_regions** shall forcibly release all allocated memory regions.

15.14.5 Introspection

15.14.5.1 convert2string

```
std::string convert2string();
```

The member function **convert2string** shall return a human-readable description of the state of the memory manager and the currently allocated regions.

15.14.5.2 for_each

```
uvm_mem_region* for_each( bool reset = false );
```

The member function **for_each** shall iterate over all currently allocated regions, If argument *reset* is set to true, it shall reset the iterator and return the first allocated region. It shall return NULL when there are no additional allocated regions to iterate on.

15.14.5.3 get_memory

```
uvm_mem* get_memory() const;
```

The member function **get_memory** shall return the reference to the memory abstraction class for the memory implementing the locations managed by this instance of the allocation manager. It shall return NULL if no memory abstraction class was specified at construction time.

15.14.6 Data members

15.14.6.1 default_alloc

```
uvm_mem_mam_policy* default_alloc;
```

The data member **default_alloc** shall define the region allocation policy. This object is repeatedly randomized when allocating new regions.

15.14.7 Type definitions

15.14.7.1 alloc_mode_e

```
typedef enum { GREEDY, THRIFTY } alloc_mode_e;
```

The type definition **alloc_mode_e** shall define an enumeration type to specify how to allocate a memory region:

— **GREEDY**: Consume new, previously unallocated memory
— **THRIFTY**: Reused previously released memory as much as possible.

15.14.7.2 locality_e

```
typedef enum { BROAD, NEARBY } locality_e;
```

The type definition **locality_e** shall define an enumeration type to specify where to locate new memory regions:

— **BROAD**: Locate new regions randomly throughout the address space.
— **NEARBY**: Locate new regions adjacent to existing regions.

15.15 uvm_mem_region

The class **uvm_mem_region** shall specify the allocated memory region.

Instances of this class are created only by the memory manager, and returned by the member functions **uvm_mem_mam::reserve_region** and **uvm_mem_mam::request_region**.

15.15.1 Class definition

```
namespace uvm {

  class uvm_mem_region
  {
   public:

    unsigned long get_start_offset() const;
    unsigned long get_end_offset() const;
    unsigned int get_len() const;
    unsigned int get_n_bytes() const;
    void release_region();
    uvm_mem* get_memory() const;
    uvm_vreg* get_virtual_registers() const;

    void write( uvm_status_e& status,
                uvm_reg_addr_t offset,
                uvm_reg_data_t value,
                uvm_path_e path = UVM_DEFAULT_PATH,
                uvm_reg_map* map = NULL,
                uvm_sequence_base* parent = NULL,
                int prior = -1,
                uvm_object* extension = NULL,
                const std::string& fname = "",
                int lineno = 0 );

    void read( uvm_status_e& status,
               uvm_reg_addr_t offset,
               uvm_reg_data_t& value,
               uvm_path_e path = UVM_DEFAULT_PATH,
               uvm_reg_map* map = NULL,
               uvm_sequence_base* parent = NULL,
               int prior = -1,
               uvm_object* extension = NULL,
               const std::string& fname = "",
```

```
               int lineno = 0 );

    void burst_write( uvm_status_e& status,
                      uvm_reg_addr_t offset,
                      std::vector<uvm_reg_data_t> value,
                      uvm_path_e path = UVM_DEFAULT_PATH,
                      uvm_reg_map* map = NULL,
                      uvm_sequence_base* parent = NULL,
                      int prior  = -1,
                      uvm_object* extension = NULL,
                      const std::string& fname  = "",
                      int lineno = 0);

    void burst_read( uvm_status_e& status,
                     uvm_reg_addr_t offset,
                     std::vector<uvm_reg_data_t>& value,
                     uvm_path_e path = UVM_DEFAULT_PATH,
                     uvm_reg_map* map = NULL,
                     uvm_sequence_base* parent = NULL,
                     int prior = -1,
                     uvm_object* extension = NULL,
                     const std::string& fname  = "",
                     int lineno = 0 );

    void poke( uvm_status_e& status,
               uvm_reg_addr_t offset,
               uvm_reg_data_t value,
               uvm_sequence_base* parent = NULL,
               uvm_object* extension = NULL,
               const std::string& fname = "",
               int lineno = 0 );

    void peek( uvm_status_e& status,
               uvm_reg_addr_t offset,
               uvm_reg_data_t& value,
               uvm_sequence_base* parent = NULL,
               uvm_object* extension = NULL,
               const std::string& fname = "",
               int lineno = 0 );

  }; // class uvm_mem_region

} // namespace uvm
```

15.15.2 Member functions

15.15.2.1 get_start_offset

```
unsigned long get_start_offset() const;
```

The member function **get_start_offset** shall return the address offset, within the memory, where this memory region starts.

15.15.2.2 get_end_offset

```
unsigned long get_end_offset() const;
```

The member function **get_end_offset** shall return the address offset, within the memory, where this memory region ends.

15.15.2.3 get_len

```
unsigned int get_len() const;
```

The member function **get_len** shall return the number of consecutive memory locations (not necessarily bytes) in the allocated region.

15.15.2.4 get_n_bytes

```
unsigned int get_n_bytes() const;
```

The member function **get_n_bytes** shall return 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.

15.15.2.5 release_region

```
void release_region();
```

The member function **release_region** shall release this region.

15.15.2.6 get_memory

```
uvm_mem* get_memory() const;
```

The member function **get_memory** shall return a reference to the memory abstraction class for the memory implementing this allocated memory region. It shall return NULL if no memory abstraction class was specified for the allocation manager that allocated this region.

15.15.2.7 get_virtual_registers

```
uvm_vreg* get_virtual_registers() const;
```

The member function **get_virtual_registers** shall return a reference to the virtual register array abstraction class implemented in this region. It shall return NULL if the memory region is not known to implement virtual registers.

15.15.2.8 write

```
void write( uvm_status_e& status,
            uvm_reg_addr_t offset,
            uvm_reg_data_t value,
            uvm_path_e path = UVM_DEFAULT_PATH,
            uvm_reg_map* map = NULL,
            uvm_sequence_base* parent = NULL,
            int prior = -1,
            uvm_object* extension = NULL,
            const std::string& fname = "",
            int lineno = 0 );
```

The member function **write** shall write to the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.1 for more details.

15.15.2.9 read

```
void read( uvm_status_e& status,
```

```
           uvm_reg_addr_t offset,
           uvm_reg_data_t& value,
           uvm_path_e path = UVM_DEFAULT_PATH,
           uvm_reg_map* map = NULL,
           uvm_sequence_base* parent = NULL,
           int prior = -1,
           uvm_object* extension = NULL,
           const std::string& fname = "",
           int lineno = 0 );
```

The member function **read** shall read from the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.2 for more details.

15.15.2.10 burst_write

```
void burst_write( uvm_status_e& status,
                  uvm_reg_addr_t offset,
                  std::vector<uvm_reg_data_t> value,
                  uvm_path_e path = UVM_DEFAULT_PATH,
                  uvm_reg_map* map = NULL,
                  uvm_sequence_base* parent = NULL,
                  int prior  = -1,
                  uvm_object* extension = NULL,
                  const std::string& fname  = "",
                  int lineno = 0);
```

The member function **burst_write** shall write to the memory locations that corresponds to the specified burst within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.3 for more details.

15.15.2.11 burst_read

```
void burst_read( uvm_status_e& status,
                 uvm_reg_addr_t offset,
                 std::vector<uvm_reg_data_t>& value,
                 uvm_path_e path = UVM_DEFAULT_PATH,
                 uvm_reg_map* map = NULL,
                 uvm_sequence_base* parent = NULL,
                 int prior = -1,
                 uvm_object* extension = NULL,
                 const std::string& fname  = "",
                 int lineno = 0 );
```

The member function **burst_read** shall read from the memory locations that corresponds to the specified burst within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.4 for more details.

15.15.2.12 poke

```
void poke( uvm_status_e& status,
           uvm_reg_addr_t offset,
           uvm_reg_data_t value,
           uvm_sequence_base* parent = NULL,
           uvm_object* extension = NULL,
           const std::string& fname = "",
```

```
          int lineno = 0 );
```

The member function **poke** shall deposit the specified value in the memory location that corresponds to the specified offset within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.5 for more details.

15.15.2.13 peek

```
void peek( uvm_status_e& status,
           uvm_reg_addr_t offset,
           uvm_reg_data_t& value,
           uvm_sequence_base* parent = NULL,
           uvm_object* extension = NULL,
           const std::string& fname = "",
           int lineno = 0 );
```

The member function **peek** shall sample the memory location that corresponds to the specified offset within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See Section 15.6.5.6 for more details.

15.16 Global declarations

This subclause defines the globally available types, enums, and utility classes as part of the UVM register layer.

15.16.1 Types

15.16.1.1 uvm_reg_data_t

The type **uvm_reg_data_t** shall define a 2-state data value with **UVM_REG_DATA_WIDTH** bits. Depending on the size of **UVM_REG_DATA_WIDTH**, the appropriate SystemC data type is selected.

15.16.1.2 uvm_reg_data_logic_t

The type **uvm_reg_data_logic_t** shall define a 4-state data value with **UVM_REG_DATA_WIDTH** bits. Depending on the size of **UVM_REG_DATA_WIDTH**, the appropriate SystemC data type is selected.

15.16.1.3 uvm_reg_addr_t

The type **uvm_reg_addr_t** shall define a 2-state address value with **UVM_REG_ADDR_WIDTH** bits. Depending on the size of **UVM_REG_ADDR_WIDTH**, the appropriate SystemC data type is selected.

15.16.1.4 uvm_reg_addr_logic_t

The type **uvm_reg_addr_logic_t** shall define a 4-state address value with **UVM_REG_ADDR_WIDTH** bits. Depending on the size of **UVM_REG_ADDR_WIDTH**, the appropriate SystemC data type is selected.

15.16.1.5 uvm_reg_byte_en_t

The type **uvm_reg_byte_en_t** shall define a 2-state byte_enable value with **UVM_REG_BYTENABLE_WIDTH** bits. Depending on the size of **UVM_REG_BYTENABLE_WIDTH**, the appropriate SystemC data type is selected.

15.16.1.6 uvm_reg_cvr_t

The type **uvm_reg_cvr_t** shall define a coverage model value set with **UVM_REG_CVR_WIDTH** bits. Symbolic values for individual coverage models are defined by the **uvm_coverage_model_e** type. The following bits in the set are assigned as follows

Table 15.2—Bits

0-7	UVM pre-defined coverage models
8-15	Coverage models defined by EDA vendors, implemented in a register model generator.
8-15	Coverage models defined
16-23	User-defined coverage models
24..	Reserved

NOTE—Coverage is not yet supported in UVM-SystemC.

15.16.1.7 uvm_hdl_path_slice

```
namespace uvm {
  typedef struct
  {
    std::string path;
    int offset;
    int size;
  } uvm_hdl_path_slice;

}
```

The type **uvm_hdl_path_slice** shall define a slice of an HDL path. It shall specify the HDL variable that corresponds to all or a portion of a register:

— *path*: Path to the HDL variable.

— *offset*: Offset of the LSB in the register that this variable implements.

— *size*: Number of bits (toward the MSB) that this variable implements.

If the HDL variable implements all of the register, offset and size are specified as -1.

15.16.2 Enumerations

15.16.2.1 uvm_status_e

The enumeration **uvm_status_e** shall return the status for register operations:

— **UVM_IS_OK**: Operation completed successfully.

— **UVM_NOT_OK**: Operation completed with error.

— **UVM_HAS_X**: Operation completed successfully bit had unknown bits.

15.16.2.2 uvm_path_e

The enumeration **uvm_path_e** shall define the path used for register operation:

— **UVM_FRONTDOOR**: Use the front door.
— **UVM_BACKDOOR**: Use the back door.
— **UVM_PREDICT**: Operation derived from observations by a bus monitor via the class **uvm_reg_predictor**.
— **UVM_DEFAULT_PATH**: Operation specified by the context.

15.16.2.3 uvm_check_e

The enumeration **uvm_check_e** shall define the values for read-only or read-and-check:

— **UVM_NO_CHECK**: Read only.
— **UVM_CHECK**: Read and check.

15.16.2.4 uvm_endianness_e

The enumeration **uvm_endianness_e** shall specify the byte ordering:

— **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.

15.16.2.5 uvm_elem_kind_e

The enumeration **uvm_elem_kind_e** shall define the type of element being read or written:

— **UVM_REG**: Register.
— **UVM_FIELD**: Field.
— **UVM_MEM**: Memory location.

15.16.2.6 uvm_access_e

The enumeration **uvm_access_e** shall define the type of operation being performed:

— **UVM_READ**: Read operation.
— **UVM_WRITE**: Write operation.

15.16.2.7 uvm_hier_e

The enumeration **uvm_hier_e** shall define whether to provide the requested information from a hierarchical context:

— **UVM_NO_HIER**: Provide info from the local context.
— **UVM_HIER**: Provide info based on the hierarchical context.

15.16.2.8 uvm_predict_e

The enumeration **uvm_predict_e** shall define how the mirror is to be updated:

— **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.

15.16.2.9 uvm_coverage_model_e

The enumeration **uvm_coverage_model_e** shall define coverage models available or desired. Multiple models may be specified by bitwise OR'ing individual model identifiers:

- **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.

NOTE—Coverage is not yet supported in UVM-SystemC.

15.16.2.10 uvm_reg_mem_tests_e

The enumeration **uvm_reg_mem_tests_e** shall select which pre-defined test sequence to execute. Multiple test sequences may be selected by bitwise OR'ing their respective symbolic values:

- **UVM_DO_REG_HW_RESET**: Run **uvm_reg_hw_reset_seq**.
- **UVM_DO_REG_BIT_BASH**: Run **uvm_reg_bit_bash_seq**.
- **UVM_DO_REG_ACCESS**: Run **uvm_reg_access_seq**.
- **UVM_DO_MEM_ACCESS**: Run **uvm_mem_access_seq**.
- **UVM_DO_SHARED_ACCESS**: Run **uvm_reg_mem_shared_access_seq**.
- **UVM_DO_MEM_WALK**: Run **uvm_mem_walk_seq**.
- **UVM_DO_ALL_REG_MEM_TESTS**: Run all of the above.

Test sequences, when selected, are executed in the order in which they are specified above.

NOTE—UVM-SystemC only contains the pre-defined test sequence **uvm_reg_bit_bash_seq**.

16. Register interaction with DUT

This clause defines classes to enable generic register read-write operations and classes to convert transactions between these generic register read-write operations and physical bus accesses.

The following classes are defined:

— **uvm_reg_item**
— **uvm_reg_bus_op**
— **uvm_reg_adapter**
— **uvm_reg_tlm_adapter**
— **uvm_reg_predictor**
— **uvm_reg_sequence**
— **uvm_reg_frontdoor**

The class **uvm_reg_item** defines the abstract register transaction item. The class **uvm_reg_bus_op** defines a descriptor for a physical bus operation that is used by **uvm_reg_adapter** subtypes to convert from a protocol-specific address, data, and read-writre operation to a bus-independent, canonical read-write operation. The class **uvm_reg_adapter** defines an interface for converting between **uvm_reg_bus_op** and a specific bus transaction. The class **uvm_reg_tlm_adapter** enables conversion between **uvm_reg_bus_op** and TLM transactions of type **uvm_tlm_gp**. The class **uvm_reg_predictor** defines a predictor component, which is used to update the register model's mirror values based on transactions explicitly observed on a physical bus. The class **uvm_reg_sequence** provides the base functionality for both user-defined register model test sequences and register translation sequences. The class **uvm_reg_frontdoor** is a facade class for register and memory frontdoor access.

16.1 uvm_reg_item

The class **uvm_reg_item** shall define an abstract register transaction item. No bus-specific information is present, although a handle to a **uvm_reg_map** is provided in case a user wishes to implement a custom address translation algorithm.

16.1.1 Class definition

```
namespace uvm {

  class uvm_reg_item : public uvm_sequence_item
  {
   public:

    // Constructor
    explicit uvm_reg_item( const std::string& name = "" );

    // Member functions

    virtual std::string convert2string() const;
    virtual void do_copy( const uvm_object& rhs );

    // Data members

    uvm_elem_kind_e element_kind;
    uvm_object* element;
    uvm_access_e access_kind;
    std::vector<uvm_reg_data_t> value;
    uvm_reg_addr_t offset;
    uvm_status_e status;
    uvm_reg_map* local_map;
    uvm_reg_map* map;
    uvm_path_e path;
```

```
    uvm_sequence_base* parent;
    int prior;
    uvm_object* extension;
    std::string bd_kind;
    std::string fname;
    int lineno;

  }; // class uvm_reg_item

} // namespace uvm
```

16.1.2 Constructor

```
explicit uvm_reg_item( const std::string& name = "" );
```

The constructor shall create a new instance of this type, giving it the optional *name*.

16.1.3 Member functions

16.1.3.1 convert2string

```
virtual std::string convert2string() const;
```

The member function **convert2string** shall return a string showing the contents of this transaction.

16.1.3.2 do_copy

```
virtual void do_copy( const uvm_object& rhs );
```

The member function **do_copy** shall copy the *rhs* object into this object. The *rhs* object shall be derived from **uvm_reg_item**.

16.1.4 Data members

16.1.4.1 element_kind

```
uvm_elem_kind_e element_kind;
```

The data member **element_kind** defines the kind of element being accessed: REG, MEM, or FIELD. See Section 15.16.2.5.

16.1.4.2 element

```
uvm_object* element;
```

The data member **element** defines the handle to the register model associated with this transaction. Use **element_kind** to determine the type to cast to: **uvm_reg**, **uvm_mem**, or **uvm_reg_field**.

16.1.4.3 access_kind

```
uvm_access_e access_kind;
```

The data member **access_kind** defines the kind of access: READ or WRITE.

16.1.4.4 value

```
std::vector<uvm_reg_data_t> value;
```

The data member **value** defines the value to write to, or after completion, the value read from the DUT. Burst operations use the values property.

16.1.4.5 offset

```
uvm_reg_addr_t offset;
```

The data member **offset** defines the offset. For memory accesses, the offset address. For bursts, the starting offset address.

16.1.4.6 status

```
uvm_status_e status;
```

The data member **status** defines the result of the transaction: **IS_OK**, **HAS_X**, or **ERROR**. See **uvm_status_e** (Section 15.16.2.1).

16.1.4.7 local_map

```
uvm_reg_map* local_map;
```

The data member **local_map** defines the local map used to obtain addresses. An application may customize address-translation using this map. Access to the sequencer and bus adapter can be obtained by getting this map's root map, then calling member functions **uvm_reg_map::get_sequencer** and **uvm_reg_map::get_adapter**.

16.1.4.8 map

```
uvm_reg_map* map;
```

The data member **map** defines 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.

16.1.4.9 path

```
uvm_path_e path;
```

The data member **path** defines the path being used: **UVM_FRONTDOOR** or **UVM_BACKDOOR**.

16.1.4.10 parent

```
uvm_sequence_base* parent;
```

The data member **parent** defines the sequence from which the operation originated.

16.1.4.11 prior

```
int prior;
```

The data member **prior** defines the priority requested of this transfer, as defined by **uvm_sequence_base::start_item**.

16.1.4.12 extension

```
uvm_object* extension;
```

The data member **extension** defines the handle to optional user data, as conveyed in the call to **write**, **read**, **mirror**, or **update** used to trigger the operation.

16.1.4.13 bd_kind

```
std::string bd_kind;
```

The data member **bd_kind** specifies the abstraction kind for the backdoor access, if the data member **path** is set to **UVM_BACKDOOR**.

16.1.4.14 fname

```
std::string fname;
```

The data member **fname** specifies the file name from where this transaction originated, if provided at the call site.

16.1.4.15 lineno

```
int lineno;
```

The data member **lineno** specifies the line number from where this transaction originated, if provided at the call site.

16.2 uvm_reg_bus_op

The class **uvm_reg_bus_op** shall define 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 are multiple of these bus operations for every abstract **uvm_reg_item** transaction. In this case, data represents the portion of **uvm_reg_item::value** 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 equal to **uvm_reg_item::value**.

16.2.1 Class definition

```
namespace uvm {

  class uvm_reg_bus_op
  {
   public:

    // Data members
```

```
    uvm_access_e kind;
    uvm_reg_addr_t addr;
    uvm_reg_data_t data;
    unsigned int n_bits;
    uvm_reg_byte_en_t byte_en;
    uvm_status_e status;

  }; // class uvm_reg_bus_op

} // namespace uvm
```

16.2.2 Data members

16.2.2.1 kind

```
uvm_access_e kind;
```

The data member **kind** defines the kind of access: **READ** or **WRITE**.

16.2.2.2 addr

```
uvm_reg_addr_t addr;
```

The data member **addr** defines the bus address.

16.2.2.3 data

```
uvm_reg_data_t data;
```

The data member **data** defines the data to write. 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.

16.2.2.4 n_bits

```
unsigned int n_bits;
```

The data member **n_bits** defines the number of bits of **uvm_reg_item::value** being transferred by this transaction.

16.2.2.5 byte_en

```
uvm_reg_byte_en_t byte_en;
```

The data member **byte_en** 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.

16.2.2.6 status

```
uvm_status_e status;
```

The data member **status** defines the result of the transaction: **UVM_IS_OK**, **UVM_HAS_X**, **UVM_NOT_OK**. See **uvm_status_e** (Section 15.16.2.1).

16.3 uvm_reg_adapter

The class **uvm_reg_adapter** shall define the interface for converting between **uvm_reg_bus_op** and a specific bus transaction.

16.3.1 Class definition

```
namespace uvm {

  class uvm_reg_adapter : public uvm_object
  {
   public:

    // Constructor

    explicit uvm_reg_adapter( const std::string& name = "" );

    // Member functions

    virtual uvm_sequence_item* reg2bus( const uvm_reg_bus_op& rw ) = 0;

    virtual void bus2reg( const uvm_sequence_item* bus_item,
                          uvm_reg_bus_op& rw ) = 0;

    virtual uvm_reg_item* get_item() const;

    // Data members

    bool supports_byte_enable;
    bool provides_responses;
    uvm_sequence_base* parent_sequence;

  }; // class uvm_reg_adapter

} // namespace uvm
```

16.3.2 Constructor

```
explicit uvm_reg_adapter( const std::string& name = "" );
```

The constructor shall create a new instance of this type, giving it the optional *name*.

16.3.3 Member functions

16.3.3.1 reg2bus

```
virtual uvm_sequence_item* reg2bus( const uvm_reg_bus_op& rw ) = 0;
```

The member function **reg2bus** shall allocate a new bus-specific **uvm_sequence_item**, assign its data members from the corresponding data members from the given generic *rw* bus operation, then return it.

Extensions of this class shall implement this member function to convert the specified **uvm_reg_bus_op** to a corresponding **uvm_sequence_item** subtype that defines the bus transaction.

16.3.3.2 bus2reg

```
virtual void bus2reg( const uvm_sequence_item* bus_item,
                      uvm_reg_bus_op& rw ) = 0;
```

The member function **bus2reg** shall copy the data members of the given bus-specific *bus_item* to the corresponding data members of the provided instance *rw*.

Extensions of this class shall implement this member function. Unlike **reg2bus**, 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.

16.3.3.3 get_item

```
virtual uvm_reg_item* get_item() const;
```

The member function **get_item** shall returns the bus-independent read/write information that corresponds to the generic bus transaction currently translated to a bus-specific transaction. This member function returns a value reference only when called in the member function **uvm_reg_adapter::reg2bus**. The member function returns NULL at all other times. The content of the return **uvm_reg_item** instance shall not be modified and used strictly to obtain additional information about the operation.

16.3.4 Data members

16.3.4.1 supports_byte_enable

```
bool supports_byte_enable;
```

The data member **supports_byte_enable** is used in extensions of this class to specify if the bus protocol supports byte enables.

16.3.4.2 provides_responses

```
bool provides_responses;
```

The data member **provides_responses** is used in extensions of this class to specify if the bus driver provides separate response items.

16.3.4.3 parent_sequence

```
uvm_sequence_base* parent_sequence;
```

The data member **parent_sequence** is used 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 data member shall implement the member function **do_clone**.

16.4 uvm_reg_tlm_adapter

The class **uvm_reg_tlm_adapter** shall define the interface for converting For converting between **uvm_reg_bus_op** and **uvm_tlm_gp** items.

16.4.1 Class definition

```
namespace uvm {

  class uvm_reg_tlm_adapter : public uvm_reg_adapter
  {
   public:

    // Constructor

    uvm_reg_tlm_adapter( const std::string& name = "uvm_reg_tlm_adapter" );
```

```
    // Member functions

    virtual uvm_sequence_item* reg2bus( const uvm_reg_bus_op& rw );

    virtual void bus2reg( const uvm_sequence_item* bus_item,
                          uvm_reg_bus_op& rw );

  }; // class uvm_reg_tlm_adapter

} // namespace uvm
```

16.4.2 Constructor

```
uvm_reg_tlm_adapter( const std::string& name = "uvm_reg_tlm_adapter" );
```

The constructor shall create a new instance of this type with the specified *name*.

16.4.3 Member functions

16.4.3.1 reg2bus

```
virtual uvm_sequence_item* reg2bus( const uvm_reg_bus_op& rw );
```

The member function **reg2bus** shall convert the provided bus transaction *rw* of type **uvm_reg_bus_op** to a sequence item of type **uvm_tlm_gp**.

16.4.3.2 bus2reg

```
virtual void bus2reg( const uvm_sequence_item* bus_item,
                          uvm_reg_bus_op& rw );
```

The member function **bus2reg** shall converts a TLM transaction item *bus_item* of type **uvm_tlm_gp** to a read-write bus transaction *rw* of type **uvm_reg_bus_op**.

16.5 uvm_reg_predictor

The class **uvm_reg_predictor** shall convert the observed bus transactions of type BUSTYPE to generic registers transactions, determines the register 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 Section 15.4.5.15 for details.

NOTE—Memories can be large, so their accesses are not predicted.

16.5.1 Class definition

```
namespace uvm {

  template <typename BUSTYPE = int>
  class uvm_reg_predictor : public uvm_component,
                            public tlm::tlm_analysis_if<BUSTYPE>
  {
   public:

    // Constructor

    explicit uvm_reg_predictor( uvm_component_name name );
```

```
    // Ports

    uvm_analysis_imp< BUSTYPE, uvm_reg_predictor<BUSTYPE> > bus_in;
    uvm_analysis_port<uvm_reg_item> reg_ap;

   // Member functions

    virtual void pre_predict( uvm_reg_item* rw );
    virtual void check_phase( uvm_phase& phase );

    // data members

    uvm_reg_map* map;
    uvm_reg_adapter* adapter;

  }; // class uvm_reg_predictor

} // namespace uvm
```

16.5.2 Constructor

```
explicit uvm_reg_predictor( uvm_component_name name );
```

The constructor shall create a new instance of this type with the specified *name*.

16.5.3 Ports

16.5.3.1 bus_in

```
uvm_analysis_imp< BUSTYPE, uvm_reg_predictor<BUSTYPE> > bus_in;
```

The port **bus_in** shall implement an analysis input port which shall observe bus transactions of type **BUSTYPE**. For each incoming transaction, the predictor shall attempt to get the register or memory handle corresponding to the observed bus address. If there is a match, the predictor calls the register or memory's member function **predict**, passing in the observed bus data. The register or memory mirror shall be updated with this data, subject to its configured access behavior--RW, RO, WO, etc. The predictor shall also convert the bus transaction to a generic **uvm_reg_item** and send it out the **reg_ap** analysis port.

If the register is wider than the bus, the predictor shall collect the multiple bus transactions needed to determine the value being read or written.

16.5.3.2 reg_ap

```
uvm_analysis_port<uvm_reg_item> reg_ap;
```

The port **reg_ap** shall implement an analysis output port that publishes transactions of type **uvm_reg_item**, which are converted from bus transactions received by port **bus_in**.

16.5.4 Member functions

16.5.4.1 pre_predict

```
virtual void pre_predict( uvm_reg_item* rw );
```

The member function **pre_predict** shall override this member function to change the value or re-direct the target register.

16.5.4.2 check_phase

```
virtual void check_phase( uvm_phase& phase );
```

The member function **check_phase** shall check that no pending register transactions are still queued.

16.5.5 Data members

16.5.5.1 map

```
uvm_reg_map* map;
```

The data member **map** is used to convert a bus address to the corresponding register or memory handle. It shall be configured before the run phase.

16.5.5.2 adapter

```
uvm_reg_adapter* adapter;
```

The data member **adapter** is used to convey the parameters of a bus operation in terms of a canonical **uvm_reg_bus_op** datum. The **uvm_reg_adapter** shall be configured before the run phase.

16.6 uvm_reg_sequence

The class **uvm_reg_sequence** shall provide the base functionality for both user-defined register model test sequences and register translation sequences.

- When used as a base for user-defined register model test sequences, this class provides convenience member functions for reading and writing registers and memories. An application implements the member function **body** to interact directly with the register model (held in the model property) or indirectly via the delegation member functions in this class.
- When used as a registertranslation sequence, objects of this class are executed directly on a bus sequencer which are used in support of a layered sequencer use model, a pre-defined convert-and-execute algorithm is provided.

Register operations do not require extending this class if none of the above services are needed. Register test sequences can be extend from the base class **uvm_sequence**(REQ,RSP) or even from outside a sequence.

NOTE—The convenience API is not yet implemented.

16.6.1 Class definition

```
namespace uvm {

  template <typename BASE = uvm_sequence<uvm_reg_item> >
  class uvm_reg_sequence : public BASE
  {
   public:

    // Constructor

    explicit uvm_reg_sequence(const std::string& name = "uvm_reg_sequence_inst" );

    // Group: Sequence API

    virtual void body();
```

```
virtual void do_reg_item( uvm_reg_item* rw );

// Group: Convenience Write/Read API

virtual void write_reg( uvm_reg*            rg,
                        uvm_status_e&       status,
                        uvm_reg_data_t      value,
                        uvm_path_e          path = UVM_DEFAULT_PATH,
                        uvm_reg_map*        map = NULL,
                        int                 prior = -1,
                        uvm_object*         extension = NULL,
                        const std::string&  fname = "",
                        int                 lineno = 0 );

virtual void read_reg( uvm_reg*            rg,
                       uvm_status_e&       status,
                       uvm_reg_data_t&     value,
                       uvm_path_e          path = UVM_DEFAULT_PATH,
                       uvm_reg_map*        map = NULL,
                       int                 prior = -1,
                       uvm_object*         extension = NULL,
                       const std::string&  fname = "",
                       int                 lineno = 0 );

virtual void poke_reg( uvm_reg*           rg,
                       uvm_status_e&      status,
                       uvm_reg_data_t     value,
                       const std::string& kind = "",
                       uvm_object*        extension = NULL,
                       const std::string& fname = "",
                       int                lineno = 0 );

virtual void peek_reg( uvm_reg*            rg,
                       uvm_status_e&       status,
                       uvm_reg_data_t&     value,
                       const std::string&  kind = "",
                       uvm_object*         extension = NULL,
                       const std::string&  fname = "",
                       int                 lineno = 0 );

virtual void update_reg( uvm_reg*           rg,
                         uvm_status_e&      status,
                         uvm_path_e         path = UVM_DEFAULT_PATH,
                         uvm_reg_map*       map = NULL,
                         int                prior = -1,
                         uvm_object*        extension = NULL,
                         const std::string& fname = "",
                         int                lineno = 0 );

virtual void mirror_reg( uvm_reg*           rg,
                         uvm_status_e&      status,
                         uvm_check_e        check  = UVM_NO_CHECK,
                         uvm_path_e         path = UVM_DEFAULT_PATH,
                         uvm_reg_map*       map = NULL,
                         int                prior = -1,
                         uvm_object*        extension = NULL,
                         const std::string& fname = "",
                         int                lineno = 0 );

virtual void write_mem( uvm_mem*           mem,
                        uvm_status_e&      status,
                        uvm_reg_addr_t     offset,
                        uvm_reg_data_t     value,
                        uvm_path_e         path = UVM_DEFAULT_PATH,
                        uvm_reg_map*       map = NULL,
                        int                prior = -1,
                        uvm_object*        extension = NULL,
                        const std::string& fname = "",
                        int                lineno = 0 );

virtual void read_mem( uvm_mem*            mem,
                       uvm_status_e&       status,
                       uvm_reg_addr_t      offset,
                       uvm_reg_data_t&     value,
                       uvm_path_e          path = UVM_DEFAULT_PATH,
                       uvm_reg_map*        map = NULL,
                       int                 prior = -1,
```

```
                            uvm_object*         extension = NULL,
                            const std::string&  fname = "",
                            int                 lineno = 0 );

    virtual void poke_mem( uvm_mem*            mem,
                            uvm_status_e&       status,
                            uvm_reg_addr_t      offset,
                            uvm_reg_data_t      value,
                            const std::string&  kind = "",
                            uvm_object*         extension = NULL,
                            const std::string&  fname = "",
                            int                 lineno = 0 );

    virtual void peek_mem( uvm_mem*            mem,
                            uvm_status_e&       status,
                            uvm_reg_addr_t      offset,
                            uvm_reg_data_t&     value,
                            const std::string&  kind = "",
                            uvm_object*         extension = NULL,
                            const std::string&  fname = "",
                            int                 lineno = 0 );

    // Data members

    uvm_reg_block* model;
    uvm_reg_adapter* adapter;
    uvm_sequencer<uvm_reg_item>* reg_seqr;

  }; // class uvm_reg_sequence

} // namespace uvm
```

16.6.2 Constructor

```
explicit uvm_reg_sequence(const std::string& name = "uvm_reg_sequence_inst" );
```

The constructor shall create a new instance of this type with the specified *name*.

16.6.3 Sequence API

16.6.3.1 body

```
virtual void body();
```

The member function **body** shall continually get a register transaction from the configured upstream sequencer, **reg_seqr**, and executes the corresponding bus transaction via **do_reg_item**.

NOTE—User-defined register model test sequences should override the member function **body** and not call the member function **body** of the base class, else a warning shall be issued and the calling process not return.

16.6.3.2 do_reg_item

```
virtual void do_reg_item( uvm_reg_item* rw );
```

The member function **do_reg_item** shall execute the given register transaction, *rw*, via the sequencer on which this sequence was started (i.e. *m_sequencer*). It shall use the configured adapter to convert the register transaction into the type expected by this sequencer.

16.6.4 Convenience Write/Read API

16.6.4.1 write_reg

```
virtual void write_reg( uvm_reg*            rg,
                        uvm_status_e&       status,
                        uvm_reg_data_t      value,
                        uvm_path_e          path = UVM_DEFAULT_PATH,
                        uvm_reg_map*        map = NULL,
                        int                 prior = -1,
                        uvm_object*         extension = NULL,
                        const std::string&  fname = "",
                        int                 lineno = 0 );
```

The member function **write_reg** shall write the given register *rg* using member function **uvm_reg::write**, supplying `this` as the parent argument.

16.6.4.2 read_reg

```
virtual void read_reg( uvm_reg*            rg,
                       uvm_status_e&       status,
                       uvm_reg_data_t&     value,
                       uvm_path_e          path = UVM_DEFAULT_PATH,
                       uvm_reg_map*        map = NULL,
                       int                 prior = -1,
                       uvm_object*         extension = NULL,
                       const std::string&  fname = "",
                       int                 lineno = 0 );
```

The member function **read_reg** shall read the given register *rg* using member function **uvm_reg::read**, supplying `this` as the parent argument.

16.6.4.3 poke_reg

```
virtual void poke_reg( uvm_reg*           rg,
                       uvm_status_e&      status,
                       uvm_reg_data_t     value,
                       const std::string& kind = "",
                       uvm_object*        extension = NULL,
                       const std::string& fname = "",
                       int                lineno = 0 );
```

The member function **poke_reg** shall poke the given register *rg* using member function **uvm_reg::poke**, supplying `this` as the parent argument.

16.6.4.4 peek_reg

```
virtual void peek_reg( uvm_reg*            rg,
                       uvm_status_e&       status,
                       uvm_reg_data_t&     value,
                       const std::string&  kind = "",
                       uvm_object*         extension = NULL,
                       const std::string&  fname = "",
                       int                 lineno = 0 );
```

The member function **peek_reg** shall peek the given register *rg* using member function **uvm_reg::peek**, supplying `this` as the parent argument.

16.6.4.5 update_reg

```
virtual void update_reg( uvm_reg*           rg,
```

```
                         uvm_status_e&      status,
                         uvm_path_e         path = UVM_DEFAULT_PATH,
                         uvm_reg_map*       map = NULL,
                         int                prior = -1,
                         uvm_object*        extension = NULL,
                         const std::string& fname = "",
                         int                lineno = 0 );
```

The member function **update_reg** shall update the given register *rg* using member function **uvm_reg::update**, supplying `this` as the parent argument.

16.6.4.6 mirror_reg

```
virtual void mirror_reg( uvm_reg*           rg,
                         uvm_status_e&      status,
                         uvm_check_e        check  = UVM_NO_CHECK,
                         uvm_path_e         path = UVM_DEFAULT_PATH,
                         uvm_reg_map*       map = NULL,
                         int                prior = -1,
                         uvm_object*        extension = NULL,
                         const std::string& fname = "",
                         int                lineno = 0 );
```

The member function **mirror_reg** shall mirror the given register *rg* using member function **uvm_reg::mirror**, supplying `this` as the parent argument.

16.6.4.7 write_mem

```
virtual void write_mem( uvm_mem*           mem,
                        uvm_status_e&      status,
                        uvm_reg_addr_t     offset,
                        uvm_reg_data_t     value,
                        uvm_path_e         path = UVM_DEFAULT_PATH,
                        uvm_reg_map*       map = NULL,
                        int                prior = -1,
                        uvm_object*        extension = NULL,
                        const std::string& fname = "",
                        int                lineno = 0 );
```

The member function **write_mem** shall write the given memory *mem* using member function **uvm_mem::write**, supplying `this` as the parent argument.

16.6.4.8 read_mem

```
virtual void read_mem( uvm_mem*            mem,
                       uvm_status_e&       status,
                       uvm_reg_addr_t      offset,
                       uvm_reg_data_t&     value,
                       uvm_path_e          path = UVM_DEFAULT_PATH,
                       uvm_reg_map*        map = NULL,
                       int                 prior = -1,
                       uvm_object*         extension = NULL,
                       const std::string&  fname = "",
                       int                 lineno = 0 );
```

The member function **read_mem** shall read the given memory *mem* using member function **uvm_mem::read**, supplying `this` as the parent argument.

16.6.4.9 poke_mem

```
virtual void poke_mem( uvm_mem*           mem,
                       uvm_status_e&      status,
                       uvm_reg_addr_t     offset,
```

```
                        uvm_reg_data_t     value,
                        const std::string& kind = "",
                        uvm_object*        extension = NULL,
                        const std::string& fname = "",
                        int                lineno = 0 );
```

The member function **poke_mem** shall poke the given memory *mem* using member function **uvm_mem::poke**, supplying `this` as the parent argument.

16.6.4.10 peek_mem

```
virtual void peek_mem( uvm_mem*           mem,
                       uvm_status_e&      status,
                       uvm_reg_addr_t     offset,
                       uvm_reg_data_t&    value,
                       const std::string& kind = "",
                       uvm_object*        extension = NULL,
                       const std::string& fname = "",
                       int                lineno = 0 );
```

The member function **peek_mem** shall peek the given memory *mem* using member function **uvm_mem::peek**, supplying `this` as the parent argument.

16.6.5 Data members

16.6.5.1 model

```
uvm_reg_block* model;
```

The data member **model** shall define the register block abstraction the sequence executes on, defined only when this sequence is a user-defined test sequence.

16.6.5.2 adapter

```
uvm_reg_adapter* adapter;
```

The data member **adapter** shall define the adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.

16.6.5.3 reg_seqr

```
uvm_sequencer<uvm_reg_item>* reg_seqr;
```

The data member **reg_seqr** shall specify the upstream sequencer between abstract register transactions and physical bus transactions. This data member is only defined when the sequence is a translation sequence, enabling a “pull” from an upstream sequencer.

16.7 uvm_reg_frontdoor

The class **uvm_reg_frontdoor** shall provide a 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 exclusively through physical addresses. The frontdoor allows access using a non-linear and/or non-mapped mechanism. Users can extend this class to provide the physical access to these registers.

16.7.1 Class definition

```
namespace uvm {

  class uvm_reg_frontdoor : public uvm_reg_sequence<uvm_sequence<uvm_sequence_item> >
  {
   public:

    // Constructor

    explicit uvm_reg_frontdoor( const std::string& name = "" );

    // Data members

    uvm_reg_item* rw_info;
    uvm_sequencer_base* sequencer;

  }; // class uvm_reg_frontdoor

} // namespace uvm
```

16.7.2 Constructor

```
explicit uvm_reg_frontdoor( const std::string& name = "" );
```

The constructor shall create a new instance of this type with the specified *name*.

16.7.3 Data members

16.7.3.1 rw_info

```
uvm_reg_item* rw_info;
```

The data member **rw_info** shall specify the information about the register being read or written.

16.7.3.2 sequencer

```
uvm_sequencer_base* sequencer;
```

The data member **sequencer** shall specify the sequencer executing the operation.

17. Global functionality

UVM provides other global functionality including functions, enums, defines, and classes. Some of these are targeted towards specific aspects of the functionality described in the UVM standard, and others are useful across multiple aspects.

All global functions reside in the UVM namespace. Functions marked with the symbol § are specific to UVM-SystemC and not available in the UVM-SystemVerilog standard.

17.1 Global functions

17.1.1 uvm_set_config_int§

```
namespace uvm {

  void uvm_set_config_int§( const std::string& inst_name,
  const std::string& field_name,
  int value );

} // namespace uvm
```

The global function **uvm_set_config_int** shall create and place an integer in a configuration database. The argument *inst_name* shall define the full hierarchical pathname of the object being configured. The argument *field_name* is the specific field that is being searched for. Both arguments *inst_name* and *field_name* may contain wildcards.

NOTE—This global function is made available since there is no command line interface option to pass configuration data.

17.1.2 uvm_set_config_string§

```
namepace uvm {

  void uvm_set_config_string§( const std::string& inst_name,
  const std::string& field_name,
  const std::string& value );

} // namespace uvm
```

The global function **uvm_set_config_string** shall create and place a string in a configuration database. The argument *inst_name* shall define the full hierarchical pathname of the object being configured. The argument *field_name* is the specific field that is being searched for. Both arguments *inst_name* and *field_name* may contain wildcards.

NOTE—This global function is made available since there is no command line interface option to pass configuration data.

17.1.3 run_test

```
namespace uvm {

  void run_test( const std::string& test_name = "" );

} // namespace uvm
```

The function **run_test** is a convenience function to start member function **uvm_root**::**run_test**. (See Section 4.3).

17.2 Global defines

17.2.1 UVM_MAX_STREAMBITS

The definition **UVM_MAX_STREAMBITS** shall be used to set the maximum size for integer types. If not defined, a default size of 4096 is used.

17.2.2 UVM_PACKER_MAX_BYTES

The definition **UVM_PACKER_MAX_BYTES** shall be used to set the maximum bytes to allocate for packing an object using the uvm_packer. Default is **UVM_MAX_STREAMBITS**, in bytes.

17.2.3 UVM_DEFAULT_TIMEOUT

The definition **UVM_DEFAULT_TIMEOUT** shall be used as default timeout for the run phases. If not defined, a default timeout of 9200 seconds shall be used. The timeout can be overridden by using the member function **uvm_root**::**set_timeout** (see Section 4.3.2.3).

17.3 Global type definitions (typedefs)

17.3.1 uvm_bitstream_t

The typedef **uvm_bitstream_t** shall define an integer type with a size defined by **UVM_MAX_STREAMBITS**. An application can use this type in member functions such as **uvm_printer**::**print_field** (see Section 5.2.3.1), **uvm_packer**::**pack_field** (see Section 5.1.3.1) and **uvm_packer**::**unpack_field** (see Section 5.1.4.3).

17.3.2 uvm_integral_t

The typedef **uvm_integral_t** shall define an integer type with a size of 64 bits. An application can use this type in member functions such as **uvm_printer::print_field_int** (see Section 5.2.3.2), **uvm_packer::pack_field_int** (see Section 5.1.3.2) and **uvm_packer::unpack_field_int** (see Section 5.1.4.2).

17.3.3 UVM_FILE

The typedef **UVM_FILE** shall define the file descriptor which supports output streams.

17.3.4 uvm_report_cb

The typedef **uvm_report_cb** is the alias for **uvm_callbacks**<**uvm_report_object**, **uvm_report_catcher** >.

17.3.5 uvm_config_int

The typedef **uvm_config_int** is the alias for **uvm_config_db**<**uvm_bitstream_t** >.

17.3.6 uvm_config_string

The typedef **uvm_config_string** is the alias for **uvm_config_db**<std::string>.

17.3.7 uvm_config_object

The typedef **uvm_config_object** is the alias for **uvm_config_db**<**uvm_object***>.

17.3.8 uvm_config_wrapper

The typedef **uvm_config_wrapper** is the alias for **uvm_config_db<uvm_object_wrapper*>**.

17.4 Global enumeration

17.4.1 uvm_action

The enumeration type **uvm_action** shall define all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.

— **UVM_NO_ACTION**: No action is taken.

— **UVM_DISPLAY**: Sends the report to the standard output.

— **UVM_LOG**: Sends the report to the file(s) for this (severity, id) pair.

— **UVM_COUNT**: Counts the number of reports with the COUNT attribute. When this value reaches **max_quit_count**, the simulation terminates.

— **UVM_EXIT**: Terminates the simulation immediately.

— **UVM_CALL_HOOK**: Callback the report hook methods.

— **UVM_STOP**: Causes the simulator to stop, enabling continuation as interactive session.

17.4.2 uvm_severity

The enumeration type **uvm_severity** shall define all possible values for report severity:

— **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. The simulation shall be terminated immediately.

17.4.3 uvm_verbosity

The enumeration type **uvm_verbosity** shall define standard verbosity levels for reports.

— **UVM_NONE**: Report is always printed. Verbosity level setting cannot disable it.

— **UVM_LOW**: Report is issued if configured verbosity is set to **UVM_LOW** or above.

— **UVM_MEDIUM**: Report is issued if configured verbosity is set to **UVM_MEDIUM** or above.

— **UVM_HIGH**: Report is issued if configured verbosity is set to **UVM_HIGH** or above.

— **UVM_FULL**: Report is issued if configured verbosity is set to **UVM_FULL** or above.

17.4.4 uvm_active_passive_enum

The enumeration type **uvm_active_passive_enum** shall define whether a component, usually an agent, is in "active" mode or "passive" mode.

— **UVM_ACTIVE**: **uvm_agent** is in "active" mode, which means that the sequencer, driver and monitor are enabled.

— **UVM_PASSIVE**: **uvm_agent** is in "passive" mode, which means that only the monitor is enabled.

17.4.5 uvm_sequence_state_enum

The enumeration type **uvm_sequence_state_enum** shall define the current sequence state.

— **UVM_CREATED**: The sequence has been allocated.
— **UVM_PRE_START**: The sequence is started and the callback **uvm_sequence_base**::**pre_start** is being executed.
— **UVM_PRE_BODY**: The sequence is started and the callback **uvm_sequence_base**::**pre_body** is being executed.
— **UVM_BODY**: The sequence is started and the callback **uvm_sequence_base**::**body** is being executed.
— **UVM_ENDED**: The sequence has completed the execution of the callback **uvm_sequence_base**::**body**.
— **UVM_POST_BODY**: The sequence is started and the callback **uvm_sequence_base**::**post_body** is being executed.
— **UVM_POST_START**: The sequence is started and the callback **uvm_sequence_base**::**post_start** is being executed.
— **UVM_STOPPED**: The sequence has been forcibly ended by issuing a **uvm_sequence_base**::**kill** on the sequence.
— **UVM_FINISHED**: The sequence is completely finished executing.

17.4.6 uvm_phase_type

The typedef **uvm_phase_type** shall define an enumeration list which defines the phase type.

— **UVM_PHASE_IMP**: The phase object is used to traverse the component hierarchy and call the component phase method as well as the callbacks **phase_started** and **phase_ended**.
— **UVM_PHASE_NODE**: The object represents a simple node instance in the graph. These nodes shall contain a reference to their corresponding IMP object.
— **UVM_PHASE_SCHEDULE**: The object represents a portion of the phasing graph, typically consisting of several NODE types, in series, parallel, or both.
— **UVM_PHASE_TERMINAL**: This internal object serves as the termination NODE for a SCHEDULE phase object.
— **UVM_PHASE_DOMAIN**: This object represents an entire graph segment that executes in parallel with the run phase. Domains may define any network of NODEs and SCHEDULEs. The built-in domain called uvm consists of a single schedule of all the run-time phases, starting with pre_reset and ending with **post_shutdown**.

17.5 uvm_coreservices_t

The class **uvm_coreservice_t** shall provide a common point for all central UVM services such as **uvm_factory**, **uvm_report_server**, etc. Each service class shall provide a static member function **get** which returns an instance adhering to the corresponding service provided by **uvm_coreservice_t**.

17.5.1 Class definition

```
namespace uvm {

  class uvm_coreservice_t
  {
   public:

    virtual uvm_factory* get_factory() const = 0;
```

```
    virtual void set_factory( uvm_factory* factory ) = 0;

    virtual uvm_report_server* get_report_server() const = 0;
    virtual void set_report_server( uvm_report_server* server ) = 0;

    virtual uvm_root* get_root() const = 0;

    static uvm_default_coreservice_t* get();

  }; // class uvm_coreservice_t

} // namespace uvm
```

17.5.2 Member functions

17.5.2.1 get_factory

```
virtual uvm_factory* get_factory() const = 0;
```

The member function **get_factory** shall return the currently enabled UVM factory. (See Section 6.4).

17.5.2.2 set_factory

```
virtual void set_factory( uvm_factory* facory ) = 0;
```

The member function **set_factory** shall specify the currently used UVM factory given as argument.

17.5.2.3 get_report_server

```
virtual uvm_report_server* get_report_server() const = 0;
```

The member function **get_report_server** shall return the current global report server. (See Section 12.4).

17.5.2.4 set_report_server

```
virtual void set_report_server( uvm_report_server* server ) = 0;
```

The member function **set_report_server** shall specify the central report server to *server*.

17.5.2.5 get_root

```
virtual uvm_root* get_root() const = 0;
```

The member function **get_root** shall return the **uvm_root** instance. (See Section 4.3).

17.5.2.6 get

```
static uvm_default_coreservice_t* get();
```

The member function **get** shall return an instance providing the **uvm_coreservice_t** interface.

17.6 uvm_default_coreservices_t

The class **uvm_default_coreservice_t** shall provide a default implementation of the **uvm_coreservice_t** API. It shall instantiate the objects **uvm_default_factory** (see Section 6.5), **uvm_default_report_server** (see Section 12.5), and **uvm_root** (see Section 4.3).

17.6.1 Class definition

```
namespace uvm {

  class uvm_default_coreservice_t : public uvm_coreservice_t
  {
   public:

    virtual uvm_factory* get_factory() const;
    virtual void set_factory( uvm_factory* factory );

    virtual uvm_report_server* get_report_server() const;
    virtual void set_report_server( uvm_report_server* server );

    virtual uvm_root* get_root() const;

  }; // class uvm_default_coreservice_t

} // namespace uvm
```

17.6.2 Member functions

17.6.2.1 get_factory

```
virtual uvm_factory* get_factory() const;
```

The member function **get_factory** shall returns the currently enabled UVM factory. When no factory has been set before, it shall instantiate a **uvm_default_factory**. (See Section 6.5).

17.6.2.2 set_factory

```
virtual void set_factory( uvm_factory* factory );
```

The member function **set_factory** shall specify the current UVM factory.

NOTE—The application needs to preserve the contents of the original factory or delegate calls to the original factory.

17.6.2.3 get_report_server

```
virtual uvm_report_server* get_report_server() const;
```

The member function **get_report_server** shall return the current global report server. If no report server has been set before, it shall return an instance of **uvm_default_report_server**. (See Section 12.5).

17.6.2.4 set_report_server

```
virtual void set_report_server( uvm_report_server* server );
```

The member function **set_report_server** shall specify the central report server to *server*.

17.6.2.5 get_root

```
virtual uvm_root* get_root() const = 0;
```

The member function **get_root** shall return the **uvm_root** instance. (See Section 4.3).

17.6.2.6 get

```
static uvm_default_coreservice_t* get();
```

The member function **get** shall return an instance providing the **uvm_coreservice_t** interface.

Annex A

(informative)

Glossary

This glossary contains brief, informal descriptions for a number of terms and phrases used in this standard. Where appropriate, the complete, formal definition of each term or phrase is given in the main body of the standard.

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

application: A C++ program, written by an end user.

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

callback: A member function overridden within a class in the component hierarchy that is called back by the kernel at certain fixed points during elaboration and simulation. UVM defines pre-defined callback functions as part of the phasing mechanism, such as **end_of_elaboration_phase**, **build_phase**, **connect_phase**, **run_phase**, etc. In addition, UVM supports the creation of user-defined callback classes and functions.

child: An instance that is within a given component. Component A is a child of component B if component A is within component B. See also: **parent**.

component: A piece of VIP that provides functionality and interfaces. Also referred to as a *transactor*.

configuration: Ability to change the properties of components or objects independent from the component hierarchy and composition. Configuration parameters can be stored in and retrieved from a central database, which can be accessed at any place in the verification environment, and at any time during the simulation.

consumer: A verification component that receives **transactions** from another **component**.

driver: A component responsible for executing or otherwise processing **transactions**, 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 the implementation of methods used for communication. Used in 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.

fifo: An instance of a primitive channel that models a first-in-first-out buffer.

foreign methodology: A verification methodology that is different from the methodology being used for the majority of the verification environment.

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

implementation: A specific concrete implementation of the UVM-SystemC class library as defined in this standard. It only implements the public shell which need be exposed to the application (for example, parts may be precompiled and distributed as object code by a tool vendor). See also: **kernel**.

kernel: The core of any UVM-SystemC implementation including the underlying elaboration and simulation engines. The kernel honors the semantics defined by this standard but may also contain implementation-specific functionality outside the scope of this standard. See also: **implementation**.

member function: A function declared within a class definition, excluding friend functions. Outside of a constructor or member function of the class or of any derived class, a non-static member function can only be accessed using the dot `.` and arrow `->` operators. See also: **method**.

method: A function that implements the behavior of a class. This term is synonymous with the C++ term **member function**. In UVM-SystemC, the term **method** is used in the context of an interface method call. Throughout this standard, the term **member function** is used when defining C++ classes (for conformance to the C++ standard), and the term **method** is used in more informal contexts and when discussing interface method calls.

monitor: A passive entity that samples DUT signals, but does not drive them.

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

primary (host) methodology: The methodology that manages the top-level operation of the verification environment and with which the user/integrator is presumably more familiar.

process: A process instance belongs to an implementation-defined class derived from class **uvm_object.** Each process instance has an associated function that represents the behavior of the process. A process may be a static or a dynamic (e.g., spawned) process. See also: **spawned process**.

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

recipient: The component that implements a callback or function that receives and processes a transaction. See also: **sender**.

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

root sequence: A sequence which has no parent sequence.

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.

sender: The component that implements a callback or function that initiates the transmission of a transaction. See also: **recipient**.

sequence: A 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 which executes **sequence**s that define the **transaction**s provided to the driver for execution.

spawned process: A process instance that is dynamically created by calling the SystemC function **sc_core::sc_spawn**. See also: **process**.

test: Specific customization of an environment to exercise required functionality of the DUT.

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

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

transactor: See *component*.

virtual sequence: A conceptual term for a **sequence** that controls the execution of **sequence**s on other **sequencer**s.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W