****

**UVM-SystemC  
Language Reference Manual**

**DRAFT**

30 May 2018

**Copyright notice**

© 2012 – 2016 NXP B.V. All rights reserved.

© 2018 Coseda Technologiers GmbHAll rights reserved.

© 2011 – 2013 Accellera Systems Initiative. All rights reserved.

© 2009 – 2011 Cadence Design Systems, Inc. (Cadence). All rights reserved.

**License**

This documentation is licensed under the Apache Software Foundation‘s Apache License, Version 2.0, January 2004. The full license is available at: http://www.apache.org/licenses/

**Trademarks**

Accellera, Accellera Systems Initiative, SystemC and UVM are trademarks of Accellera Systems Initiative Inc. All other trademarks and trade names are the property of their respective owners.

**Destination Control Statement**

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader’s responsibility to determine the applicable regulations and comply with them.

**Disclaimer**

THE CONTRIBUTORS AND THEIR LICENSORS MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

**Acknowledgements**

The creation of this document 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 HETEROGENOUS RELIABLE DESIGN AND INTEGRATION' (VERDI). The research leading to this result has received funding from the European Commission under grand agreement No 287562.

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

http://cordis.europa.eu/fp7

http://www.verdi-fp7.eu

Special thanks to the Accellera Systems Initiative to grant authorization to use portions of its Universal Verification Methodology Reference Implementation (UVM version 1.1d, March 2013) in this document.

The partners in the VERDI consortium wish to thank Cadence Design Systems Inc. for the initial donation of the UVM-SC Library Reference and documentation (UVM version 1.0, June 2011). This document has been derived from this work, and further enhanced and extended to make it compatible with the UVM 1.1 standard.

**Bugs and Suggestions**

Please report bugs and suggestions about this document to:

uvm-systemc-feedback@lists.accellera.org

**Table of Contents**

[1. Introduction 15](#_Toc454918779)

[2. Terminology 16](#_Toc454918780)

[2.1 Shall, should, may, can 16](#_Toc454918781)

[2.2 Implementation, application 16](#_Toc454918782)

[2.3 Call, called from, derived from 16](#_Toc454918783)

[2.4 Implementation-defined 16](#_Toc454918784)

[3. UVM-SystemC overview 17](#_Toc454918785)

[3.1 Namespace 17](#_Toc454918786)

[3.2 Header files 17](#_Toc454918787)

[3.3 Global functions 17](#_Toc454918788)

[3.4 Base classes 17](#_Toc454918789)

[3.5 Policy classes 17](#_Toc454918790)

[3.6 Registry and factory classes 18](#_Toc454918791)

[3.7 Component hierarchy classes 18](#_Toc454918792)

[3.8 Sequencer classes 18](#_Toc454918793)

[3.9 Sequence classes 19](#_Toc454918794)

[3.10 Configuration and resource classes 19](#_Toc454918795)

[3.11 Phasing and synchronization classes 19](#_Toc454918796)

[3.12 Reporting classes 20](#_Toc454918797)

[3.13 Macros 20](#_Toc454918798)

[3.14 Existing SystemC functionality used in UVM-SystemC 21](#_Toc454918799)

[3.15 Methodology for hierarchy construction 22](#_Toc454918800)

[4. Global functions 23](#_Toc454918801)

[4.1 uvm\_set\_config\_int⭍ 23](#_Toc454918802)

[4.2 uvm\_set\_config\_string⭍ 23](#_Toc454918803)

[4.3 run\_test 23](#_Toc454918804)

[5. Base Classes 24](#_Toc454918805)

[5.1 uvm\_void 24](#_Toc454918806)

[5.1.1 Class definition 24](#_Toc454918807)

[5.2 uvm\_object 24](#_Toc454918808)

[5.2.1 Class definition 24](#_Toc454918809)

[5.2.2 Construction 26](#_Toc454918810)

[5.2.3 Identification 26](#_Toc454918811)

[5.2.4 Creation 27](#_Toc454918812)

[5.2.5 Printing 28](#_Toc454918813)

[5.2.6 Recording 29](#_Toc454918814)

[5.2.7 Copying 29](#_Toc454918815)

[5.2.8 Comparing 30](#_Toc454918816)

[5.2.9 Packing 30](#_Toc454918817)

[5.2.10 Unpacking 31](#_Toc454918818)

[5.2.11 Object macros 32](#_Toc454918819)

[5.3 uvm\_root 33](#_Toc454918820)

[5.3.1 Class definition 33](#_Toc454918821)

[5.3.2 Simulation control 34](#_Toc454918822)

[5.3.3 Topology 35](#_Toc454918823)

[5.3.4 Global variable 35](#_Toc454918824)

[5.4 uvm\_port\_base 36](#_Toc454918825)

[5.4.1 Class definition 36](#_Toc454918826)

[5.4.2 Template parameter IF 37](#_Toc454918827)

[5.4.3 Constructor 37](#_Toc454918828)

[5.4.4 Member functions 37](#_Toc454918829)

[5.5 uvm\_component\_name⭍ 38](#_Toc454918830)

[5.5.1 Class definition 38](#_Toc454918831)

[5.5.2 Constraints on usage 38](#_Toc454918832)

[5.5.3 Constructor 39](#_Toc454918833)

[5.5.4 Destructor 39](#_Toc454918834)

[5.5.5 operator const char\* 39](#_Toc454918835)

[6. Policy classes 40](#_Toc454918836)

[6.1 uvm\_packer 40](#_Toc454918837)

[6.1.1 Class definition 40](#_Toc454918838)

[6.1.2 Constraints on usage 43](#_Toc454918839)

[6.1.3 Packing 43](#_Toc454918840)

[6.1.4 Unpacking 44](#_Toc454918841)

[6.1.5 operator <<, operator>> 45](#_Toc454918842)

[6.1.6 Data members (variables) 45](#_Toc454918843)

[6.2 uvm\_printer 46](#_Toc454918844)

[6.2.1 Class definition 46](#_Toc454918845)

[6.2.2 Constraints on usage 49](#_Toc454918846)

[6.2.3 Printing types 49](#_Toc454918847)

[6.2.4 Printer subtyping 51](#_Toc454918848)

[6.2.5 Data members 52](#_Toc454918849)

[6.3 uvm\_table\_printer 52](#_Toc454918850)

[6.3.1 Class definition 53](#_Toc454918851)

[6.3.2 Constructor 53](#_Toc454918852)

[6.3.3 emit 53](#_Toc454918853)

[6.4 uvm\_tree\_printer 53](#_Toc454918854)

[6.4.1 Class definition 53](#_Toc454918855)

[6.4.2 Constructor 53](#_Toc454918856)

[6.4.3 emit 54](#_Toc454918857)

[6.5 uvm\_line\_printer 54](#_Toc454918858)

[6.5.1 Class definition 54](#_Toc454918859)

[6.5.2 Constructor 54](#_Toc454918860)

[6.5.3 emit 54](#_Toc454918861)

[6.6 uvm\_comparer 54](#_Toc454918862)

[6.6.1 Class definition 54](#_Toc454918863)

[6.6.2 Constraints on usage 56](#_Toc454918864)

[6.6.3 Member functions 56](#_Toc454918865)

[6.6.4 Comparer settings 58](#_Toc454918866)

[6.7 Default policy objects 60](#_Toc454918867)

[7. Registry and factory classes 62](#_Toc454918868)

[7.1 uvm\_object\_wrapper 62](#_Toc454918869)

[7.1.1 Class definition 62](#_Toc454918870)

[7.1.2 Member functions 63](#_Toc454918871)

[7.2 uvm\_object\_registry 63](#_Toc454918872)

[7.2.1 Class definition 63](#_Toc454918873)

[7.2.2 Template parameter T 64](#_Toc454918874)

[7.2.3 Member functions 64](#_Toc454918875)

[7.3 uvm\_component\_registry 65](#_Toc454918876)

[7.3.1 Class definition 65](#_Toc454918877)

[7.3.2 Template parameter T 66](#_Toc454918878)

[7.3.3 Member functions 66](#_Toc454918879)

[7.4 uvm\_factory 68](#_Toc454918880)

[7.4.1 Class definition 68](#_Toc454918881)

[7.4.2 Registering types 70](#_Toc454918882)

[7.4.3 Type and instance overrides 70](#_Toc454918883)

[7.4.4 Creation 71](#_Toc454918884)

[7.4.5 Debug 73](#_Toc454918885)

[8. Component hierarchy classes 75](#_Toc454918886)

[8.1 uvm\_component 75](#_Toc454918887)

[8.1.1 Class definition 75](#_Toc454918888)

[8.1.2 Construction interface 79](#_Toc454918889)

[8.1.3 Hierarchy interface 79](#_Toc454918890)

[8.1.4 Phasing interface 81](#_Toc454918891)

[8.1.5 Process control interface 88](#_Toc454918892)

[8.1.6 Configuration interface 89](#_Toc454918893)

[8.1.7 Objection interface 89](#_Toc454918894)

[8.1.8 Factory interface 90](#_Toc454918895)

[8.1.9 Hierarchical reporting interface 92](#_Toc454918896)

[8.1.10 Macros 94](#_Toc454918897)

[8.2 uvm\_driver 95](#_Toc454918898)

[8.2.1 Class definition 95](#_Toc454918899)

[8.2.2 Template parameters 95](#_Toc454918900)

[8.2.3 Ports 95](#_Toc454918901)

[8.2.4 Member functions 96](#_Toc454918902)

[8.3 uvm\_monitor 96](#_Toc454918903)

[8.3.1 Class definition 96](#_Toc454918904)

[8.3.2 Member functions 97](#_Toc454918905)

[8.4 uvm\_agent 97](#_Toc454918906)

[8.4.1 Class definition 97](#_Toc454918907)

[8.4.2 Member functions 97](#_Toc454918908)

[8.5 uvm\_env 98](#_Toc454918909)

[8.5.1 Class definition 98](#_Toc454918910)

[8.5.2 Member functions 98](#_Toc454918911)

[8.6 uvm\_test 99](#_Toc454918912)

[8.6.1 Class definition 99](#_Toc454918913)

[8.6.2 Member functions 99](#_Toc454918914)

[8.7 uvm\_scoreboard 99](#_Toc454918915)

[8.7.1 Class definition 99](#_Toc454918916)

[8.7.2 Member functions 100](#_Toc454918917)

[8.8 uvm\_subscriber 100](#_Toc454918918)

[8.8.1 Class definition 100](#_Toc454918919)

[8.8.2 Template parameter T 101](#_Toc454918920)

[8.8.3 Export 101](#_Toc454918921)

[8.8.4 Member functions 101](#_Toc454918922)

[9. Sequencer classes 102](#_Toc454918923)

[9.1 uvm\_sequencer\_base 102](#_Toc454918924)

[9.1.1 Class definition 102](#_Toc454918925)

[9.1.2 Constructor 103](#_Toc454918926)

[9.1.3 Member functions 103](#_Toc454918927)

[9.2 uvm\_sequencer\_param\_base 107](#_Toc454918928)

[9.2.1 Class definition 107](#_Toc454918929)

[9.2.2 Template parameters 108](#_Toc454918930)

[9.2.3 Constructor 108](#_Toc454918931)

[9.2.4 Requests 108](#_Toc454918932)

[9.3 uvm\_sequencer 108](#_Toc454918933)

[9.3.1 Class definition 108](#_Toc454918934)

[9.3.2 Template parameters 109](#_Toc454918935)

[9.3.3 Constructor 109](#_Toc454918936)

[9.3.4 Exports 109](#_Toc454918937)

[9.3.5 Sequencer interface 110](#_Toc454918938)

[9.3.6 Macros 111](#_Toc454918939)

[10. Sequence classes 112](#_Toc454918940)

[10.1 uvm\_transaction 112](#_Toc454918941)

[10.1.1 Class definition 112](#_Toc454918942)

[10.1.2 Constructors 113](#_Toc454918943)

[10.1.3 Constraints on usage 113](#_Toc454918944)

[10.1.4 Member functions 113](#_Toc454918945)

[10.2 uvm\_sequence\_item 113](#_Toc454918946)

[10.2.1 Class definition 113](#_Toc454918947)

[10.2.2 Constructors 114](#_Toc454918948)

[10.2.3 Member functions 114](#_Toc454918949)

[10.3 uvm\_sequence\_base 116](#_Toc454918950)

[10.3.1 Class definition 116](#_Toc454918951)

[10.3.2 Constructor 118](#_Toc454918952)

[10.3.3 Sequence state 119](#_Toc454918953)

[10.3.4 Sequence execution 119](#_Toc454918954)

[10.3.5 Sequence control 121](#_Toc454918955)

[10.3.6 Sequence item execution 123](#_Toc454918956)

[10.3.7 Response interface 124](#_Toc454918957)

[10.3.8 Data members 126](#_Toc454918958)

[10.4 uvm\_sequence 126](#_Toc454918959)

[10.4.1 Class definition 126](#_Toc454918960)

[10.4.2 Template parameters 126](#_Toc454918961)

[10.4.3 Constructor 127](#_Toc454918962)

[10.4.4 Member functions 127](#_Toc454918963)

[11. Configuration and resource classes 128](#_Toc454918964)

[11.1 uvm\_config\_db 128](#_Toc454918965)

[11.1.1 Class definition 128](#_Toc454918966)

[11.1.2 Template parameter T 129](#_Toc454918967)

[11.1.3 Constraints on usage 129](#_Toc454918968)

[11.1.4 Member functions 129](#_Toc454918969)

[11.2 uvm\_resource\_db 130](#_Toc454918970)

[11.2.1 Class definition 130](#_Toc454918971)

[11.2.2 Template parameter T 132](#_Toc454918972)

[11.2.3 Constraints on usage 132](#_Toc454918973)

[11.2.4 Member functions 132](#_Toc454918974)

[11.3 uvm\_resource\_db\_options 134](#_Toc454918975)

[11.3.1 Class definition 134](#_Toc454918976)

[11.3.2 Member functions 135](#_Toc454918977)

[11.4 uvm\_resource\_options 135](#_Toc454918978)

[11.4.1 Class definition 135](#_Toc454918979)

[11.4.2 Member functions 136](#_Toc454918980)

[11.5 uvm\_resource\_base 136](#_Toc454918981)

[11.5.1 Class definition 136](#_Toc454918982)

[11.5.2 Constructor 137](#_Toc454918983)

[11.5.3 Resource database interface 138](#_Toc454918984)

[11.5.4 Read-only interface 138](#_Toc454918985)

[11.5.5 Notification 138](#_Toc454918986)

[11.5.6 Scope interface 138](#_Toc454918987)

[11.5.7 Priority 139](#_Toc454918988)

[11.5.8 Utility functions 139](#_Toc454918989)

[11.5.9 Audit trail 139](#_Toc454918990)

[11.6 uvm\_resource\_pool 140](#_Toc454918991)

[11.6.1 Class definition 140](#_Toc454918992)

[11.6.2 get 141](#_Toc454918993)

[11.6.3 spell\_check 141](#_Toc454918994)

[11.6.4 Set interface 142](#_Toc454918995)

[11.6.5 Lookup 142](#_Toc454918996)

[11.6.6 Set priority 144](#_Toc454918997)

[11.6.7 Debug 145](#_Toc454918998)

[11.7 uvm\_resource 145](#_Toc454918999)

[11.7.1 Class definition 145](#_Toc454919000)

[11.7.2 Template parameter T 146](#_Toc454919001)

[11.7.3 Type interface 146](#_Toc454919002)

[11.7.4 Set/Get interface 147](#_Toc454919003)

[11.7.5 Read/Write interface 147](#_Toc454919004)

[11.7.6 Priority 148](#_Toc454919005)

[11.8 uvm\_resource\_types 148](#_Toc454919006)

[11.8.1 Class definition 148](#_Toc454919007)

[11.8.2 Type definitions (typedefs) 149](#_Toc454919008)

[12. Phasing and synchronization classes 150](#_Toc454919009)

[12.1 uvm\_phase 150](#_Toc454919010)

[12.1.1 Class definition 150](#_Toc454919011)

[12.1.2 Construction 152](#_Toc454919012)

[12.1.3 State 152](#_Toc454919013)

[12.1.4 Callbacks 153](#_Toc454919014)

[12.1.5 Schedule 154](#_Toc454919015)

[12.1.6 Synchronization 155](#_Toc454919016)

[12.1.7 Jumping 156](#_Toc454919017)

[12.2 uvm\_domain 156](#_Toc454919018)

[12.2.1 Class definition 156](#_Toc454919019)

[12.2.2 Constructor 157](#_Toc454919020)

[12.2.3 Member functions 157](#_Toc454919021)

[12.3 uvm\_bottomup\_phase 158](#_Toc454919022)

[12.3.1 Class definition 158](#_Toc454919023)

[12.3.2 Constructor 158](#_Toc454919024)

[12.3.3 Member functions 159](#_Toc454919025)

[12.4 uvm\_topdown\_phase 159](#_Toc454919026)

[12.4.1 Class definition 159](#_Toc454919027)

[12.4.2 Constructor 160](#_Toc454919028)

[12.4.3 Member functions 160](#_Toc454919029)

[12.5 uvm\_process\_phase° (uvm\_task\_phase*†*) 160](#_Toc454919030)

[12.5.1 Class definition 160](#_Toc454919031)

[12.5.2 Member functions 161](#_Toc454919032)

[12.6 uvm\_objection 161](#_Toc454919033)

[12.6.1 Class definition 161](#_Toc454919034)

[12.6.2 Constructor 163](#_Toc454919035)

[12.6.3 Objection control 163](#_Toc454919036)

[12.6.4 Callback hooks 165](#_Toc454919037)

[12.6.5 Objections status 166](#_Toc454919038)

[12.7 uvm\_callback 167](#_Toc454919039)

[12.7.1 Class definition 167](#_Toc454919040)

[12.7.2 Member functions 167](#_Toc454919041)

[12.8 uvm\_callback\_iter 168](#_Toc454919042)

[12.8.1 Class definition 168](#_Toc454919043)

[12.8.2 Template parameter T 168](#_Toc454919044)

[12.8.3 Template parameter CB 168](#_Toc454919045)

[12.8.4 Constructor 168](#_Toc454919046)

[12.8.5 Member functions 169](#_Toc454919047)

[12.9 uvm\_callbacks 169](#_Toc454919048)

[12.9.1 Class definition 170](#_Toc454919049)

[12.9.2 Template parameter T 171](#_Toc454919050)

[12.9.3 Template parameter CB 171](#_Toc454919051)

[12.9.4 Constructor 171](#_Toc454919052)

[12.9.5 Add/delete interface 171](#_Toc454919053)

[12.9.6 Iterator interfaces 172](#_Toc454919054)

[12.9.7 Debug 173](#_Toc454919055)

[13. Reporting classes 174](#_Toc454919056)

[13.1 uvm\_report\_object 174](#_Toc454919057)

[13.1.1 Class definition 175](#_Toc454919058)

[13.1.2 Constructors 177](#_Toc454919059)

[13.1.3 Reporting 177](#_Toc454919060)

[13.1.4 Verbosity configuration 179](#_Toc454919061)

[13.1.5 Action configuration 180](#_Toc454919062)

[13.1.6 File configuration 181](#_Toc454919063)

[13.1.7 Override configuration 182](#_Toc454919064)

[13.1.8 Report handler configuration 182](#_Toc454919065)

[13.2 uvm\_report\_handler 183](#_Toc454919066)

[13.2.1 Class definition 183](#_Toc454919067)

[13.2.2 Constructor 184](#_Toc454919068)

[13.2.3 Member functions 184](#_Toc454919069)

[13.3 uvm\_report\_server 185](#_Toc454919070)

[13.3.1 Class definition 185](#_Toc454919071)

[13.3.2 Constructor 187](#_Toc454919072)

[13.3.3 Member functions 187](#_Toc454919073)

[13.4 uvm\_report\_catcher 190](#_Toc454919074)

[13.4.1 Class definition 190](#_Toc454919075)

[13.4.2 Constructor 192](#_Toc454919076)

[13.4.3 Current message state 192](#_Toc454919077)

[13.4.4 Change message state 193](#_Toc454919078)

[13.4.5 Debug 194](#_Toc454919079)

[13.4.6 Callback interface 194](#_Toc454919080)

[13.4.7 Reporting 195](#_Toc454919081)

[14. Macros 197](#_Toc454919082)

[14.1 Component and object registration macros 197](#_Toc454919083)

[14.1.1 Macro definitions 197](#_Toc454919084)

[14.1.2 UVM\_OBJECT\_UTILS, UVM\_OBJECT\_PARAM\_UTILS 197](#_Toc454919085)

[14.1.3 UVM\_COMPONENT\_UTILS, UVM\_COMPONENT\_PARAM\_UTILS 198](#_Toc454919086)

[14.2 Reporting macros 198](#_Toc454919087)

[14.2.1 Macro definitions 198](#_Toc454919088)

[14.2.2 UVM\_INFO 198](#_Toc454919089)

[14.2.3 UVM\_WARNING 199](#_Toc454919090)

[14.2.4 UVM\_ERROR 199](#_Toc454919091)

[14.2.5 UVM\_FATAL 199](#_Toc454919092)

[14.3 Sequence execution macros 199](#_Toc454919093)

[14.3.1 Macro definitions 199](#_Toc454919094)

[14.3.2 UVM\_DO 200](#_Toc454919095)

[14.3.3 UVM\_DO\_PRI 200](#_Toc454919096)

[14.3.4 UVM\_DO\_ON 200](#_Toc454919097)

[14.3.5 UVM\_DO\_ON\_PRI 200](#_Toc454919098)

[14.3.6 UVM\_CREATE 200](#_Toc454919099)

[14.3.7 UVM\_CREATE\_ON 201](#_Toc454919100)

[14.3.8 UVM\_DECLARE\_P\_SEQUENCER 201](#_Toc454919101)

[14.4 Callback macros 201](#_Toc454919102)

[14.4.1 Macro definitions 201](#_Toc454919103)

[14.4.2 UVM\_REGISTER\_CB 201](#_Toc454919104)

[14.4.3 UVM\_DO\_CALLBACKS 201](#_Toc454919105)

[15. TLM interfaces 203](#_Toc454919106)

[15.1 uvm\_blocking\_put\_port 204](#_Toc454919107)

[15.1.1 Class definition 204](#_Toc454919108)

[15.1.2 Template parameter T 204](#_Toc454919109)

[15.1.3 Constructor 204](#_Toc454919110)

[15.1.4 Member functions 204](#_Toc454919111)

[15.2 uvm\_blocking\_get\_port 205](#_Toc454919112)

[15.2.1 Class definition 205](#_Toc454919113)

[15.2.2 Template parameter T 205](#_Toc454919114)

[15.2.3 Constructor 205](#_Toc454919115)

[15.2.4 Member functions 206](#_Toc454919116)

[15.3 uvm\_blocking\_peek\_port 206](#_Toc454919117)

[15.3.1 Class definition 206](#_Toc454919118)

[15.3.2 Template parameter T 206](#_Toc454919119)

[15.3.3 Constructor 207](#_Toc454919120)

[15.3.4 Member functions 207](#_Toc454919121)

[15.4 uvm\_blocking\_get\_peek\_port 207](#_Toc454919122)

[15.4.1 Class definition 207](#_Toc454919123)

[15.4.2 Template parameter T 208](#_Toc454919124)

[15.4.3 Constructor 208](#_Toc454919125)

[15.4.4 Member functions 208](#_Toc454919126)

[15.5 uvm\_nonblocking\_put\_port 209](#_Toc454919127)

[15.5.1 Class definition 209](#_Toc454919128)

[15.5.2 Template parameter T 209](#_Toc454919129)

[15.5.3 Constructor 209](#_Toc454919130)

[15.5.4 Member functions 209](#_Toc454919131)

[15.6 uvm\_nonblocking\_get\_port 210](#_Toc454919132)

[15.6.1 Class definition 210](#_Toc454919133)

[15.6.2 Template parameter T 210](#_Toc454919134)

[15.6.3 Constructor 210](#_Toc454919135)

[15.6.4 Member functions 211](#_Toc454919136)

[15.7 uvm\_nonblocking\_peek\_port 211](#_Toc454919137)

[15.7.1 Class definition 211](#_Toc454919138)

[15.7.2 Template parameter T 212](#_Toc454919139)

[15.7.3 Constructor 212](#_Toc454919140)

[15.7.4 Member functions 212](#_Toc454919141)

[15.8 uvm\_nonblocking\_get\_peek\_port 212](#_Toc454919142)

[15.8.1 Class definition 212](#_Toc454919143)

[15.8.2 Template parameter T 213](#_Toc454919144)

[15.8.3 Constructor 213](#_Toc454919145)

[15.8.4 Member functions 213](#_Toc454919146)

[15.9 uvm\_analysis\_port 214](#_Toc454919147)

[15.9.1 Class definition 214](#_Toc454919148)

[15.9.2 Template parameter T 215](#_Toc454919149)

[15.9.3 Constructor 215](#_Toc454919150)

[15.9.4 Member functions 215](#_Toc454919151)

[15.10 uvm\_analysis\_export 215](#_Toc454919152)

[15.10.1 Class definition 216](#_Toc454919153)

[15.10.2 Template parameter T 216](#_Toc454919154)

[15.10.3 Constructor 216](#_Toc454919155)

[15.10.4 Member functions 216](#_Toc454919156)

[15.11 uvm\_analysis\_imp 217](#_Toc454919157)

[15.11.1 Class definition 217](#_Toc454919158)

[15.11.2 Template parameters 217](#_Toc454919159)

[15.11.3 Constructor 217](#_Toc454919160)

[15.11.4 Member functions 218](#_Toc454919161)

[15.12 uvm\_tlm\_req\_rsp\_channel 218](#_Toc454919162)

[15.12.1 Class definition 218](#_Toc454919163)

[15.12.2 Template parameters 219](#_Toc454919164)

[15.12.3 Ports and exports 219](#_Toc454919165)

[15.12.4 Constructor 221](#_Toc454919166)

[15.13 uvm\_sqr\_if\_base 221](#_Toc454919167)

[15.13.1 Class definition 221](#_Toc454919168)

[15.13.2 Template parameters 222](#_Toc454919169)

[15.13.3 Member functions 222](#_Toc454919170)

[15.14 uvm\_seq\_item\_pull\_port 224](#_Toc454919171)

[15.14.1 Class definition 225](#_Toc454919172)

[15.14.2 Template parameters 225](#_Toc454919173)

[15.14.3 Constructor 225](#_Toc454919174)

[15.14.4 Member functions 225](#_Toc454919175)

[15.15 uvm\_seq\_item\_pull\_export 225](#_Toc454919176)

[15.15.1 Class definition 225](#_Toc454919177)

[15.15.2 Template parameters 226](#_Toc454919178)

[15.15.3 Constructor 226](#_Toc454919179)

[15.15.4 Member functions 226](#_Toc454919180)

[15.16 uvm\_seq\_item\_pull\_imp 226](#_Toc454919181)

[15.16.1 Class definition 226](#_Toc454919182)

[15.16.2 Template parameters 227](#_Toc454919183)

[15.16.3 Constructor 227](#_Toc454919184)

[15.16.4 Member functions 227](#_Toc454919185)

[16. Global defines, typedefs and enumerations 228](#_Toc454919186)

[16.1 Global defines 228](#_Toc454919187)

[16.1.1 UVM\_MAX\_STREAMBITS 228](#_Toc454919188)

[16.1.2 UVM\_PACKER\_MAX\_BYTES 228](#_Toc454919189)

[16.1.3 UVM\_DEFAULT\_TIMEOUT 228](#_Toc454919190)

[16.2 Type definitions (typedefs) 228](#_Toc454919191)

[16.2.1 uvm\_bitstream\_t 228](#_Toc454919192)

[16.2.2 uvm\_integral\_t 228](#_Toc454919193)

[16.2.3 UVM\_FILE 228](#_Toc454919194)

[16.2.4 uvm\_report\_cb 228](#_Toc454919195)

[16.2.5 uvm\_config\_int 228](#_Toc454919196)

[16.2.6 uvm\_config\_string 229](#_Toc454919197)

[16.2.7 uvm\_config\_object 229](#_Toc454919198)

[16.2.8 uvm\_config\_wrapper 229](#_Toc454919199)

[16.3 Enumeration 229](#_Toc454919200)

[16.3.1 uvm\_action 229](#_Toc454919201)

[16.3.2 uvm\_severity 229](#_Toc454919202)

[16.3.3 uvm\_verbosity 229](#_Toc454919203)

[16.3.4 uvm\_active\_passive\_enum 230](#_Toc454919204)

[16.3.5 uvm\_sequence\_state\_enum 230](#_Toc454919205)

[16.3.6 uvm\_phase\_type 230](#_Toc454919206)

[Annex A. UVM-SystemVerilog features not included in UVM-SystemC 232](#_Toc454919207)

[A.1 No field macros 232](#_Toc454919208)

[A.2 No automated configuration 232](#_Toc454919209)

[A.3 No transaction recording 232](#_Toc454919210)

[A.4 No register abstraction layer 232](#_Toc454919211)

[A.5 No constraint randomization and coverage classes 232](#_Toc454919212)

[A.6 No assertions 232](#_Toc454919213)

[Annex B. Renamed functions UVM-SystemC versus UVM-SystemVerilog 233](#_Toc454919214)

[Annex C. Terminology 234](#_Toc454919215)

[C.1 Definitions 234](#_Toc454919216)

[C.2 Acronyms and Abbreviations 236](#_Toc454919217)

[Index 238](#_Toc454919218)

# 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 that of the existing UVM 1.1 standard and associated base class library implementation in SystemVerilog (SV). 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 proof-of-concept library that works with any IEEE 1666-2011 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 (formerly OSCI) SystemC   
proof-of-concept library [1].

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 1.1 standard are marked with symbol ⭍. Elements marked with symbol ° are renamed in UVM-SystemC, in contrast to the UVM 1.1 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 1.1 name is given in brackets and marked with *†*. Note that these original names are not used in   
UVM-SystemC.

[1] As process control extensions are only supported in the Accellera Systems Initiative SystemC 2.3.0 release (or later) of the   
proof-of-concept library, it is required to have this library installed prior to UVM-SystemC installation.

Terminology

## 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*.

## 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.

## 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.

## 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.

UVM-SystemC overview

## Namespace

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

## 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.

## Global functions

A minimal set of global functions is defined in the global namespace offering generic UVM capabilities and convenience functions for configuration and printing. The global functions are specified in section 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\_component\_name**⭍

The base classes are specified in section 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 section 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 factory classes are:

* **uvm\_object\_wrapper**
* **uvm\_object\_registry**
* **uvm\_component\_registry**
* **uvm\_factory**

The registry and factory classes are specified in section 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**, and **uvm\_scoreboard** are derived from the class **uvm\_component**.

The UVM component classes are specified in section 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**
* **uvm\_sqr\_if\_base**

The sequencer classes are specified in section 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 section 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 section 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 section 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**: 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 section 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 section 14.

## 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.

## 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.

# Global functions

All global functions reside in the UVM namespace. Functions marked with symbol ⭍ are not compatible with the UVM 1.1 standard.

## uvm\_set\_config\_int⭍

|  |
| --- |
| void **uvm\_set\_config\_int**( const std::string& *inst\_name*,  const std::string& *field\_name*,  int *value* ); |

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.

## uvm\_set\_config\_string⭍

|  |
| --- |
| void **uvm\_set\_config\_string**( const std::string& *inst\_name*,  const std::string& *field\_name*,  const std::string& *value* ); |

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.

## run\_test

|  |
| --- |
| void run\_test( const std::string& *test\_name* = "" ); |

The function **run\_test** is a convenience function to start member function **uvm\_root**::**run\_test**. (See 5.3.2.1)

# Base Classes

## 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 will inherit none of the UVM functionality, but such classes may be placed in **uvm\_void**-typed containers along with other UVM objects.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_void** {};  } *// namespace uvm* |

## 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**.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_object** : public **uvm\_void**, public **uvm\_report\_object**  {  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* |

### Construction

#### 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.

### Identification

#### 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.

#### 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**.

#### 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.

#### get\_inst\_id

|  |
| --- |
| virtual int **get\_inst\_id**() const; |

The member function **get\_inst\_id** shall return the object’s unique, numeric instance identifier.

#### 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.

#### 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.

To enable use of this member function, a user’s subtype must implement a version that returns the subtype’s wrapper.

#### 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).

#### 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.

### Creation

#### 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 via a base handle of type **uvm\_object**. Every class deriving from **uvm\_object**, directly or indirectly, shall implement the member function **create**.

#### 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.

### Printing

#### 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 6.7.1.4)

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.

#### 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 6.7.1.4)

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 will manage all string concatenations and provide the string to **sprint** to return to the caller.

#### 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.

#### 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.

### Recording

#### 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 6.7.1.7).

NOTE−The recording mechanism is vendor-specific. By providing access via a common interface, the **uvm\_recorder** policy provides vendor-independent access to a simulator’s recording capabilities.

#### 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−Vendor-specific recording implementations are encapsulated in the recorder policy, thereby insulating user-code from vendor-specific behavior.

### Copying

#### 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**.

#### 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.

### Comparing

#### 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 6.7.1.6).

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**.

#### 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.

### Packing

#### 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 6.7.1.5).

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**.

#### 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 6.7.1.5).

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**.

#### 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 6.7.1.5).

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**.

#### do\_pack

|  |
| --- |
| virtual void **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.

### Unpacking

#### 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 6.7.1.5).

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.

#### 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 6.7.1.5).

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.

#### 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 6.7.1.5).

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.

#### 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.

### 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.

## 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**.

### 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* |

### Simulation control

#### 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** will register both the common phases as well as the UVM run-time phases. (See section 12).

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.

#### 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.

#### 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 16.1.3).

#### 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.

#### 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 5.3.2.4.)

### Topology

#### 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.

#### 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.

#### 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.

#### 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.

### Global variable

#### 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 an **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** and **find\_all**.
* *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 will not continue to the **start\_of\_simulation\_phase**.

## uvm\_port\_base

The class **uvm\_port\_base** shall provide methods to bind ports to interfaces or to other ports or exports, 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**.

### 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* |

### 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 or export which is derived from this type.

### 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.

### Member functions

#### get\_name

|  |
| --- |
| virtual const std::string **get\_name**() const; |

The member function **get\_name** shall return the leaf name of this port.

#### 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.

#### 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.

#### 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.

#### 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**.

## 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.

### 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* |

### 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**.

### 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.

### 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* ).

### 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**.

# 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**

## 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 16.1.2).

### 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* |

### Constraints on usage

An application shall not explicitly create an instance of the class **uvm\_packer**.

### Packing

#### 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.

#### 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.

#### 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.

#### 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.

#### 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 the IEEE Standard for Floating-Point Arithmetic (IEEE Std. 754-1985).

#### 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 will be 0.

### Unpacking

#### is\_null

|  |
| --- |
| virtual bool **is\_null**(); |

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.

#### 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* is 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.

#### 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**.

#### 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.

#### 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.

#### 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 the IEEE Standard for Floating-Point Arithmetic (IEEE Std. 754-1985).

#### 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* must 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.

#### get\_packed\_size

|  |
| --- |
| virtual unsigned int **get\_packed\_size**() const; |

The member function **get\_packed\_size** returns the number of bits that were packed.

### 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 6.1.6.3).
  + 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**.

### Data members (variables)

#### 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**.

#### 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**.

#### 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.

#### 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.

## 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.

### 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* |

### Constraints on usage

An application shall not explicitly create an instance of the class **uvm\_printer**.

### Printing types

#### 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).

#### 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).

#### 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.

#### 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**.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

### Printer subtyping

#### 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.

#### 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).

#### format\_header

|  |
| --- |
| virtual std::string **format\_header**(); |

The member **format\_header** shall offer a hook to override the base header with a custom header.

#### format\_footer

|  |
| --- |
| virtual std::string **format\_footer**(); |

The member **format\_footer** shall offer a hook to override the base footer with a custom footer.

#### 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.

#### 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.

#### 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 method 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.

#### 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 method lets the printer know that the array printing is complete.

### Data members

#### knobs

|  |
| --- |
| **uvm\_printer\_knobs** knobs; |

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

## uvm\_table\_printer

The class **uvm\_table\_printer** shall provide a pre-defined printing output in a tabular format.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_table\_printer** : public **uvm\_printer**  {  public:  **uvm\_table\_printer**();  virtual std::string **emit**();  }; *// class uvm\_table\_printer*  } *// namespace uvm* |

### Constructor

|  |
| --- |
| **uvm\_table\_printer**(); |

The constructor shall create a new instance of type **uvm\_table\_printer**.

### emit

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

## uvm\_tree\_printer

The class **uvm\_tree\_printer** shall provide a pre-defined printing output in a tree format.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_tree\_printer** : public **uvm\_printer**  {  public:  **uvm\_tree\_printer**();  virtual std::string **emit**();  }; *// class uvm\_tree\_printer*  } *// namespace uvm* |

### Constructor

|  |
| --- |
| **uvm\_tree\_printer**(); |

The constructor shall create a new instance of type **uvm\_tree\_printer**.

### emit

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

## uvm\_line\_printer

The class **uvm\_table\_printer** shall provide a pre-defined printing output in a line format.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_line\_printer** : public **uvm\_printer**  {  public:  **uvm\_line\_printer**();  virtual std::string **emit**();  }; *// class uvm\_line\_printer*  } *// namespace uvm* |

### Constructor

|  |
| --- |
| **uvm\_line\_printer**(); |

The constructor shall create a new instance of type **uvm\_line\_printer**.

### emit

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

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_comparer**  {  public:  *// Group: 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* |

### Constraints on usage

An application shall not explicitly create an instance of the class **uvm\_comparer**.

### Member functions

#### 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 size variable indicates the number of bits to compare; size must be less than or equal to **UVM\_MAX\_STREAMBITS**. The argument *radix* is used for reporting purposes, the default radix is hex.

#### 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 size variable indicates the number of bits to compare; size must be less than or equal to 64. The argument *radix* is used for reporting purposes, the default radix is hex.

#### compare\_field\_real, compare\_field\_double

|  |
| --- |
| 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. The left-hand-side *lhs* and right-hand-side *rhs* arguments are used for comparison.

#### 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**() ).

#### 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.

#### 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 6.6.4.5) and severity (see 6.6.4.7) settings.

### Comparer settings

#### set\_policy

|  |
| --- |
| **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**.

#### get\_policy

|  |
| --- |
| **uvm\_recursion\_policy\_enum** **get\_policy**() const; |

The member function **get\_policy** shall return the comparison policy.

#### 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.

#### 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.

#### 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**.

#### get\_verbosity

|  |
| --- |
| unsigned int **get\_verbosity**() const; |

The member function **get\_verbosity** shall return the verbosity for printed messages.

#### 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**.

#### get\_severity

|  |
| --- |
| **uvm\_severity** **get\_severity**() const; |

The member function **get\_severity** shall return the severity for printed messages.

#### 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.

#### 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.

#### 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.

#### 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**.

#### 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.

#### 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.

## Default policy objects

This section lists the default policy objects.

#### 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**, whichcan be used with **uvm\_object**::**do\_print** to get tabular style printing.

#### 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**, whichcan be used with **uvm\_object**::**do\_print** to get a multi-line tree style printing.

#### 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**, whichcan be used with **uvm\_object**::**do\_print** to get a single-line style printing.

#### 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.

#### uvm\_default\_packer

|  |
| --- |
| extern **uvm\_printer**\* **uvm\_default\_printer**; |

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.

#### 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.

#### 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.

# 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**

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 will determine what type of object to create based on its configuration, and will ask that type’s proxy to create an instance of the type, which is returned to the application.

## 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.

### 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* |

### Member functions

#### 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 7.2).

#### 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 7.3).

#### 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.

## 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 wth the factory.

### 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 **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* |

### Template parameter T

The template parameter T specifies the object type of the objects being registered. The object type must be a derivative of class **uvm\_object**.

### Member functions

#### 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**.

#### 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**.

#### 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.

#### 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.

#### 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 replace is true, a previous override on *original\_type* is replaced, otherwise a previous override, if any, remains intact.

#### 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.

## 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.

### 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 **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* |

### Template parameter T

The template parameter T specifies the object type of the components being registered. The object type must be a derivative of class **uvm\_component**.

### Member functions

#### 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**.

#### 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**.

#### 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.

#### 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.

#### 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 replace is true, a previous override on *original\_type* is replaced, otherwise a previous override, if any, remains intact.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_factory** {  public:  **uvm\_factory**();    *// Group: Registering types*  void **do\_register**° ( **uvm\_object\_wrapper**\* *obj* );  *// Group: Type & instance overrides*  void **set\_inst\_override\_by\_type**( **uvm\_object\_wrapper***\* original\_type*,  **uvm\_object\_wrapper**\* *override\_type*,  const std::string& *full\_inst\_path* );  void **set\_inst\_override\_by\_name**( const std::string& *original\_type\_name*,  const std::string& *override\_type\_name*,  const std::string& *full\_inst\_path* );  void **set\_type\_override\_by\_type**( **uvm\_object\_wrapper**\* *original\_type*,  **uvm\_object\_wrapper**\* *override\_type*,  bool *replace* = true );  void **set\_type\_override\_by\_name**( const std::string& *original\_type\_name*,  const std::string& *override\_type\_name*,  bool *replace* = true );  *// Group: Creation*  **uvm\_object**\* **create\_object\_by\_type**( **uvm\_object\_wrapper**\* *requested\_type*,  const std::string& *parent\_inst\_path* = "",  const std::string& *name* = "" );  **uvm\_object**\* **create\_object\_by\_name**( const std::string& *requested\_type\_name*,  const std::string& *parent\_inst\_path* = "",  const std::string& *name* = "" );  **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 );    **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*  void **debug\_create\_by\_type**( **uvm\_object\_wrapper**\* *requested\_type*,  const std::string& *parent\_inst\_path* = "",  const std::string& *name* = "" );  void **debug\_create\_by\_name**( const std::string& *requested\_type\_name*,  const std::string& *parent\_inst\_path* = "",  const std::string& *name* = "" );    **uvm\_object\_wrapper**\* **find\_override\_by\_type**( **uvm\_object\_wrapper**\* *requested\_type*,  const std::string& *full\_inst\_path* );  **uvm\_object\_wrapper**\* **find\_override\_by\_name**( const std::string& *requested\_type\_name*,  const std::string& *full\_inst\_path* );  void **print**( int all\_types = 1 );  }; *// class uvm\_factory*  } *// namespace uvm* |

### Registering types

#### do\_register° (register*†*)

|  |
| --- |
| Void **do\_register**( **uvm\_object\_wrapper**\* *obj* ); |

The member function **do\_register**° shall be used to register an object or a component with the factory. Usually, an application will 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.

NOTE−The UVM standard defines the member function **register***†* for factory registration. As ‘register’ is a reserved keyword in C++, this member function has been renamed to **do\_register**° in UVM-SystemC.

### Type and instance overrides

#### set\_inst\_override\_by\_type

|  |
| --- |
| 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*. 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 will match all contexts.

When the factory processes instance overrides, the instance queue shall be processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides. This way, the general override will not override the specific override.

#### set\_inst\_override\_by\_name

|  |
| --- |
| 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*. 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 will produce the type represented by *override\_type\_name*, which must 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 will match all contexts.

When the factory processes instance overrides, the instance queue shall be processed in order of the override call. Thus, more specific overrides should be set in place first, followed by more general overrides. This way, the general override will not override the specific override.

#### set\_type\_override\_by\_type

|  |
| --- |
| void **set\_type\_override\_by\_type**( **uvm\_object\_wrapper**\* *original\_type*,  **uvm\_object\_wrapper**\* *override\_type*,  bool *replace* = true ); |

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, 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 replace is true, a previous override on *original\_type* is replaced, otherwise a previous override, if any, remains intact.

#### set\_type\_override\_by\_name

|  |
| --- |
| 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\_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, provided no instance override applies. 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 will produce the type represented by *override\_type\_name*, which must be preregistered with the factory.

When replace is true, a previous override on *original\_type\_name* is replaced, otherwise a previous override, if any, remains intact.

### Creation

#### create\_object\_by\_type

|  |
| --- |
| **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, 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.

#### create\_object\_by\_name

|  |
| --- |
| **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, which is specified by argument *requested\_type\_name*. The requested type must have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested\_type\_name*, an error 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 8.1.1). 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**.

#### create\_component\_by\_type

|  |
| --- |
| **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, 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*.

#### create\_component\_by\_name

|  |
| --- |
| **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, which is specified by argument *requested\_type\_name*. The requested type must have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested\_type\_name*, an error 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 8.1.1). 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**.

### Debug

#### debug\_create\_by\_type

|  |
| --- |
| 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. 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**.

#### debug\_create\_by\_name

|  |
| --- |
| 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. 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**.

#### find\_override\_by\_type

|  |
| --- |
| 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. The argument *full\_inst\_path* is typically derived from the parent’s instance path and the leaf name of the object to be created.

#### find\_override\_by\_name

|  |
| --- |
| **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. The argument *full\_inst\_path* is typically derived from the parent’s instance path and the leaf name of the object to be created.

#### print

|  |
| --- |
| 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.

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.

# 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 section 9)

## uvm\_component

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

* Hierarchy
* 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

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_component** : public **sc\_core**::**sc\_module**,  public **uvm\_report\_object**  {  public:  *// Group: Construction*  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* |

### Construction interface

When creating a new UVM component, an application must 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 must 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.

#### 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.

### Hierarchy interface

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

#### 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.

#### 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 5.2.3.2).

#### 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**.

#### 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*.

#### 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.

#### get\_next\_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.

#### get\_num\_children

|  |
| --- |
| int **get\_num\_children**() const; |

The member function **get\_num\_children** shall return the number of the component’s children.

#### 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;

#### 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.

#### 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 0. The test and any other top level components have a depth of 1, and so on.

### 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 must complete before the next phase begins.

The phases can be grouped in three main categories:

* Pre-run phases
* Run-time phases
* Post-run phases

#### 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.

#### 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**.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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 will 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.

#### 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.

#### 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.

#### get\_domain

|  |
| --- |
| **uvm**\_domain\* **get\_domain**() const; |

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

#### 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.

#### 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.

### 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−Process control extensions are only supported when using the Accellera Systems Initiative SystemC 2.3.0 release of the proof-of-concept library.

#### 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**.

#### 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.

### Configuration interface

The configuration interface accommodates additional printing and debug facilities for user-defined configurations using the configuration database **uvm\_config\_db**.

#### 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

#### 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.

#### 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.

### Objection interface

These member functions provide object level access into the **uvm\_objection** mechanism.

#### 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*.

#### 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*.

#### 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*.

### 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.

#### 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 7.4.4.4).

#### 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 7.4.4.2).

#### 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 7.4.3.3).

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 will produce the *override\_type*.

#### 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 7.4.3.1).

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 will produce the *override\_type*.

#### 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 7.4.3.4).

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 will produce the type represented by *override\_type\_name*. The argument *override\_type\_name* must refer to a preregistered type in the factory.

#### 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 7.4.3.2).

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 will produce the type represented by *override\_type\_name*. The *override\_type\_name* must refer to a preregistered type in the factory.

#### 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.

### 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 will be sent to its associated file descriptor.

#### 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.

#### 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. An verbosity associated with a particular severity-id pair takes precedence over an verbosity associated with id, which takes precedence over a verbosity associated with a severity.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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 will be ignored.

#### 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.

### 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.

## 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**.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ = **uvm\_sequence\_item**, typename RSP = REQ>  class **uvm\_driver** : public **uvm\_component**  {  public:  **uvm\_seq\_item\_pull\_port**<REQ, RSP> seq\_item\_port;  **uvm\_analysis\_port**<RSP> rsp\_port;  explicit **uvm\_driver**( **uvm\_component\_name** *name* );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_driver*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.These object types must be a derivative of class **uvm\_sequence\_item**.

### Ports

#### 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.

#### rsp\_port

|  |
| --- |
| **uvm\_analysis\_port**<RSP> rsp\_port; |

The port **rsp\_port** shall provide a way of sending responses back to the connected sequencer.

### Member functions

#### 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.

#### 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.

## 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 will automatically inherit features that may be added to **uvm\_monitor** in the future.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_monitor** : public **uvm\_component**  {  public:  explicit **uvm\_monitor**( **uvm\_component\_name** *name* );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_monitor*  } *// namespace uvm* |

### Member functions

#### 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.

#### 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.

## uvm\_agent

The class **uvm\_agent** is the base class for the creation of agents. Deriving from **uvm\_agent** will allow an application to distinguish agents from other component types also using its inheritance. Such agents will 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_agent** : public **uvm\_component**  {  public:  explicit **uvm\_agent**( **uvm\_component\_name** *name* );  virtual const std::string **get\_type\_name**() const;  **uvm\_active\_passive\_enum get\_is\_active**() const;  }; *// class uvm\_agent*  } *// namespace uvm* |

### Member functions

#### 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.

#### 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.

#### 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 16.3.4). An application may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_env** : public **uvm\_component**  {  public:  explicit **uvm\_env**( **uvm\_component\_name** *name* );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_env*  } *// namespace uvm* |

### Member functions

#### Constructor

|  |
| --- |
| explicit **uvm\_env**( **uvm\_component\_name** *name* ); |

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

#### 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.

## uvm\_test

The class **uvm\_test** is the base class for the test environment.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_test** : public **uvm\_component**  {  public:  explicit **uvm\_test**( **uvm\_component\_name** name );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_test*  } *// namespace uvm* |

### Member functions

#### 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.

#### 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.

## uvm\_scoreboard

The class **uvm\_scoreboard** is the base class for the creation of a scoreboard. Deriving from **uvm\_scoreboard** will allow an application to distinguish scoreboards from other component types inheriting directly from **uvm\_component**. Such scoreboards will automatically inherit and benefit from features that may be added to **uvm\_scoreboard** in the future.

### 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* |

### Member functions

#### 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.

#### 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.

## 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 must 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename T = int>  class **uvm\_subscriber** : public **uvm\_component**  {  public:  **uvm\_analysis\_export**<T> analysis\_export;  explicit **uvm\_subscriber**( **uvm\_component\_name** name );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_subscriber*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis export.

### Export

#### analysis\_export

|  |
| --- |
| **uvm\_analysis\_export**<T> analysis\_export; |

The export **analysis\_export** shall provide access to the member function **write** method, which derived subscribers shall implement.

### Member functions

#### 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.

#### 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.

# 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**

NOTE−The UVM-SystemC sequencer classes only partially implement the standardized UVM sequencer capabilities. However, these definitions are sufficient to build a functional sequencer.

## uvm\_sequencer\_base

The class **uvm\_sequencer\_base** is the root base class for all sequencer classes.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_sequencer\_base** : public **uvm\_component**  {  public:  explicit **uvm\_sequencer\_base**( **uvm\_component\_name** *name* );  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* |

### 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.

### Member functions

#### 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.

#### 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*.

#### 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, they will accumulate in the sequencer, eventually causing response overflow unless member function **uvm\_sequence\_base**::**set\_response\_queue\_error\_report\_disabled** is called.

#### 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.

#### 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 must 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.

#### 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 will return the next time that the driver calls **item\_done** or **put**. If a specific *transaction\_id* is specified, then the call will only return when the driver indicates that it has completed that specific item.

#### 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.

#### 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.

#### 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 will 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.

#### 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 will 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.

#### 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*.

#### 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*.

#### stop\_sequences

|  |
| --- |
| virtual void **stop\_sequences**(); |

The member function **stop\_sequences** shall inform the 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.

#### 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.

#### 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.

#### 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.

#### 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**. That member function will specify the next sequence to grant.

The default arbitration mechanism shall be set to **SEQ\_ARB\_FIFO**.

#### get\_arbitration

|  |
| --- |
| **SEQ\_ARB\_TYPE** **get\_arbitration**() const; |

The member function **get\_arbitration** shall return the current arbitration mode set for the sequencer (see 9.1.3.17).

#### 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.

#### 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. (See 9.2.2).

This function shall only be called after a **wait\_for\_grant** call.

NOTE−Rerandomize capabilities are not yet implemented for UVM-SystemC.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ = **uvm\_sequence\_item**, typename RSP = REQ>  class **uvm\_sequencer\_param\_base** : public **uvm\_sequencer\_base**  {  public:  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* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm\_sequence\_item**.

### 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.

### Requests

#### 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−Rerandomize capabilities are not yet implemented.

#### 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.

## 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**.

### 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:  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**( **tlm**::**tlm\_tag**<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**( **tlm**::**tlm\_tag**<REQ>\* *req* = NULL );  virtual void **get**( REQ& *req* );  virtual REQ **peek**( **tlm**::**tlm\_tag**<REQ>\* *req* = NULL );  virtual void **put**( const RSP& *rsp* );  virtual void **stop\_sequences**();  }; *// class uvm\_sequencer*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm\_sequence\_item**.

### 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.

### Exports

#### 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 15.13).

### Sequencer interface

#### get\_next\_item

|  |
| --- |
| virtual REQ **get\_next\_item**( **tlm**::**tlm\_tag**<REQ>\* *req* = NULL ); |

The member function **get\_next\_item** shall retrieve the next available item from a sequence (see also 15.13.3.1).

#### 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 15.13.3.2).

#### 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 15.13.3.3).

#### get

|  |
| --- |
| virtual REQ **get**( **tlm**::**tlm\_tag**<REQ>\* *req* = NULL );  virtual void **get**( REQ& *req* ); |

The member function **get** shall retrieve the next available item from a sequence (see also 15.13.3.4).

#### peek

|  |
| --- |
| virtual REQ **peek**( **tlm**::**tlm\_tag**<REQ>\* *req* = NULL ); |

The member function **peek** shall return the current request item if one is in the FIFO (see also 15.13.3.5).

#### 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 15.13.3.6).

#### 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.

### Macros

#### 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*.

# 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 will arbitrate among the parallel sequences. By default, requests are granted in a first-in-first-out (FIFO) order (see 9.1.3.17).

## uvm\_transaction

The class **uvm\_transaction** is the root base class for all UVM transactions. As such, the class **uvm\_sequence\_item** will be derived from this class. The main purpose of this class is to provide timestamp properties, notification events, and transaction recording.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_transaction** : public **uvm\_object**  {  public:  **uvm\_transaction**();  explicit **uvm\_transaction**( const std::string& *name* );  void **set\_transaction\_id**( int *id* );  int **get\_transaction\_id**() const;  }; *// class uvm\_transaction*  } *// namespace uvm* |

### 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.

### 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**.

### Member functions

#### 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.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_sequence\_item** : public **uvm\_transaction**  {  public:  **uvm\_sequence\_item**();  explicit **uvm\_sequence\_item**( const std::string& *name* );  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* |

### 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.

### Member functions

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### get\_depth

|  |
| --- |
| int **get\_depth**() const; |

The member function **get\_depth** shall return the depth of sequence from its parent. A parent sequence will have a depth of 1, its child will have a depth of 2, and its grandchild will have a depth of 3.

#### 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**.

#### 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).

#### 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).

#### 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.

## uvm\_sequence\_base

The class **uvm\_sequence\_base** defines the primary interface member functions to create, control and execute the sequences.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_sequence\_base** : public **uvm\_sequence\_item**  {  public:  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: 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**();  *// Data members*  **uvm\_phase**\* starting\_phase;  }; *// class uvm\_sequence\_base*  } *// namespace uvm* |

### 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.

### Sequence state

#### 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 16.3.5). This member function can be used to wait on the sequence reaching or changing from one or more states.

#### 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.

### Sequence execution

#### start

|  |
| --- |
| virtual void **start**( **uvm\_sequencer\_base**\* *sequencer*,  **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 must 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** will be called before and after calling the member function **body** of the sequence.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

### Sequence control

#### 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 will be used by the sequencer the next time that it arbitrates between sequences.

#### get\_priority

|  |
| --- |
| int **get\_priority**() const; |

The member function **get\_priority** shall return the current priority of the sequence.

#### 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 will not be chosen.

If all requesting sequences are not relevant, then the sequencer shall call **wait\_for\_relevant** on all sequences and re-arbitrate upon its return.

#### 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 must also implement the member function **wait\_for\_relevant**.

#### 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 will be requested on the current default sequencer. A lock request will 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.

#### 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 will be requested on the current default sequencer. A grab request is put in front of the arbitration queue. It will 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.

#### 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 will be done on the current default sequencer.

#### 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 will be done on the current default sequencer.

#### 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.

#### 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.

#### 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 STOPPED and the callback functions **post\_body** and **post\_start** are not being executed.

#### 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**.

### Sequence item execution

#### 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 will 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.

#### 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 will be initialized here by using the sequencer specified by argument *sequencer*. If argument *sequencer* is not specified or set to NULL, the default sequencer will be used (see also 10.2.3.4). 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.

#### 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**.

#### 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 will be used by the arbiter. If the argument *lock\_request* is set to true, then the sequencer will 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.

#### 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 will be randomized before being sent to the driver.

NOTE−In an application, the member function **send\_request** shall only be called after a call to **wait\_for\_grant**.

#### 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 will return the next time that the driver calls **item\_done** or **put**. If a specific *transaction\_id* is specified, then the call will 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 will hang, having missed the earlier notification.

### Response interface

#### 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**.

#### 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.

#### 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.

#### 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 will 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.

#### 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.

#### 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.

#### 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.

#### clear\_response\_queue

|  |
| --- |
| virtual void **clear\_response\_queue**(); |

The member function **clear\_response\_queue** shall empty the response queue for the sequence.

### Data members

#### starting\_phase

|  |
| --- |
| **uvm\_phase**\* *starting\_phase*; |

The data member *starting\_phase* shall specify the phase in which this sequence was started. The *starting\_phase* shall be set when the sequence is started as the default sequence (see 9.1.3.3).

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ = **uvm\_sequence\_item**, typename RSP = REQ>  class **uvm\_sequence** : public **uvm\_sequence\_base**  {  public:  explicit **uvm\_sequence**( const std::string& *name* );  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* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm\_sequence\_item**.

### 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.

### Member functions

#### 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 will be randomized before being sent to the driver.

NOTE−In an application, the member function **send\_request** shall only be called after a call to **wait\_for\_grant**.

#### 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 method will 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.

#### 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 will block until a response is received.

If no *transaction\_id* is passed as an argument, this member function will return the next response sent to this sequence. If a *transaction\_id* is specified, the member function will block until a response with that transaction ID is received in the response queue.

# 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**

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  template <class T>  class **uvm\_config\_db**  {  public:  **uvm\_config\_db**();  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* |

### Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the configuration database.

### Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm\_config\_db** are static, so they must be called using the **operator::**.

### Member functions

#### set

|  |
| --- |
| static void **set**( **uvm\_component**\* *cntxt*,  const std::string& *instname*,  const std::string& *fieldname*,  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.

#### get

|  |
| --- |
| static bool **get**( **uvm\_component**\* *cntxt*,  const std::string& *instname*,  const std::string& *fieldname*,  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.

#### 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.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  template < typename T = **uvm\_object\*** >  class **uvm\_resource\_db**  {  public:  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* |

### Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

### Constraints on usage

To remain compatible with UVM-SystemVerilog, all of the member functions in class **uvm\_resource\_db** are static, so they must be called using the **operator::**. An application shall not instantiate this class, but shall call the static member functions directly.

### Member functions

#### 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.

#### 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.

#### 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.

#### 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

#### 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 will not be entered into the name map. But is does have an argument *scope* for lookup purposes. The argument *accessor* is used for auditing.

#### 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.

#### 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.

#### 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.

#### 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. Care must be taken with this function. If a **get\_by\_name** match is found for name and scope then val will be written to that matching resource and thus may impact other scopes which also match the resource.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_resource\_db\_options**  {  public:  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* |

### Member functions

#### 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.

#### turn\_off\_tracing

|  |
| --- |
| static void **turn\_off\_tracing**(); |

The member function **turn\_off\_tracing** shall disable tracing for the resource database.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_resource\_options**  {  public:  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* |

### Member functions

#### 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.

#### 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.

#### is\_auditing

|  |
| --- |
| static bool **is\_auditing**(); |

The member function **is\_auditing** shall return true if auditing is enabled; otherwise it shall return false.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  class **uvm\_resource\_base** : public **uvm\_object**  {  public:  **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: Precedence*  unsigned int **precedence**;  static int unsigned **default\_precedence**;  }; *// class uvm\_resource\_base*  } *// namespace uvm* |

### 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.

### Resource database interface

#### 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.

### Read-only interface

#### 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 will cause an error.

#### 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.

### Notification

#### 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.

### Scope interface

#### 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 will be converted to a regular expression before it is stored.

#### 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.

#### 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.

### Priority

#### 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.

### Utility functions

#### 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.

### Audit trail

#### 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.

#### 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.

#### print\_accessors

|  |
| --- |
| virtual void **print\_accessors**() const; |

The member function **print\_accessors** shall print the access records for this resource.

#### 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.

## 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.

### 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: Set Priority*  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* |

### get

|  |
| --- |
| static **uvm\_resource\_pool**\* get(); |

The member function **get** shall return the singleton handle to the resource pool.

### 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.

### Set interface

#### 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 must 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**.

#### 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.

#### 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 will override the name.

#### 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 will override the type.

### Lookup

#### 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 ); |

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*.

#### 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 will be the one that is returned.

#### 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 will be first in the list and the lowest precedence will be last. Resources that have the same precedence and the same name will be ordered by most recently set first.

#### 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 true, the member function shall report potential errors.

#### 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.

#### get\_by\_type

|  |
| --- |
| **uvm\_resource\_base**\* **get\_by\_type**( const std::string& *scope*,  **uvm\_resource\_base**\* *type\_handle* ) const; |

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.

#### 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.

#### 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*.

#### 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.

### Set priority

#### 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.

#### 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.

#### 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*.

### Debug

#### 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.

#### 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 true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

#### 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 true, the audit trail is printed for each resource along with the name, value, and scope regular expression.

## uvm\_resource

The class **uvm\_resource** shall provide the interface to read and write to the resource database.

### 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* |

### Template parameter T

The template parameter T specifies the object type of the objects being stored in or retrieved from the resource database.

### Type interface

#### 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.

#### 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.

### Set/Get interface

#### set

|  |
| --- |
| void **set**(); |

The member function **set** shall put the resource into the global resource pool.

#### 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.

#### 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 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.

#### 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.

### Read/Write interface

#### 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.

#### 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**.

### Priority

#### 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.

#### 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.

## 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.

### 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* |

### Type definitions (typedefs)

#### rsrc\_q\_t

The typedef **rsrc\_q\_t** shall define a queue of handles of type **uvm\_resource\_base**.

NOTE−UVM-SystemC does not define **uvm\_queue**, since std::queue can be used.

#### override\_t

The typedef **override\_t** shall define an enumeration to override enumepriority of 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.

#### 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.

# 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**°: A phase implementation for phases which are launched as spawned processes. In UVM-SystemVerilog, this class was called **uvm\_task\_phase***†*.
* **uvm\_objection**: Mechanism to synchronize phases based on passing execution status information between running processes.

## uvm\_phase

The class **uvm\_phase** shall provide the base class for the UVM phasing mechanism.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_phase** : public **uvm\_object**  {  public:  *// Group: Construction*  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: Synchronization*  **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 );  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* |

### Construction

#### 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.

#### 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 16.3.6).

### State

#### get\_state

|  |
| --- |
| **uvm\_phase\_state** **get\_state**() const; |

The member function **get\_state** shall return the current state of this 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.

#### find\_by\_name

|  |
| --- |
| **uvm\_phase**\* **find\_by\_name**( const std::string& *name*, bool *stay\_in\_scope* = true ); |

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 true, it searches only within this phase’s schedule or domain.

#### find

|  |
| --- |
| **uvm\_phase**\* **find**( const **uvm\_phase**\* *phase*, bool *stay\_in\_scope* = true ); |

The member function **find** shall locate the phase node with the specified phase implementation and return its handle. If argument *stay\_in\_scope* is true, it searches only within this phase’s schedule or domain.

#### 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.

#### 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.

#### 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.

### Callbacks

#### 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.

#### 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.

### Schedule

#### 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.

#### get\_parent

|  |
| --- |
| **uvm\_phase**\* **get\_parent**() const; |

The member function **get\_parent** shall return the parent schedule node, if any, for hierarchical graph traversal.

#### 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.

#### 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.

#### 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.

#### get\_domain

|  |
| --- |
| **uvm\_domain**\* **get\_domain**() const; |

The member function **get\_domain** shall return the enclosing domain.

#### 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.

#### 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.

### Synchronization

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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 will wait until the phase state is any of **UVM\_EQ** or none of **UVM\_NE** the provided states.

### Jumping

#### 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.

#### 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.

## uvm\_domain

The class **uvm\_domain** shall provide a phasing schedule node representing an independent branch of the schedule.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_domain** : public **uvm\_phase**  {  public:  explicit **uvm\_domain**( const std::string& *name* );  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* |

### 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.

### Member functions

#### 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.

#### 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.

#### 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.

#### get\_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.

#### get\_uvm\_domain

|  |
| --- |
| static **uvm\_domain**\* **get\_uvm\_domain**(); |

The member function **get\_uvm\_domain** shall return the handle to the singleton *uvm* domain.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  class **uvm\_bottomup\_phase** : public **uvm\_phase**  {  public:  explicit **uvm\_bottomup\_phase**( const std::string& *name* );  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* |

### 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.

### Member functions

#### 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.

#### 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*.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  class **uvm\_topdown\_phase** : public **uvm\_phase**  {  public:  explicit **uvm\_topdown\_phase**( const std::string& *name* );  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* |

### 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.

### Member functions

#### 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.

#### 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*.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_process\_phase**° : public **uvm\_phase**  {  public:  explicit **uvm\_process\_phase**°( const std::string& *name* );  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* |

### Member functions

#### 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.

#### 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*.

## 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.

### 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* |

### Constructor

|  |
| --- |
| **uvm\_objection**();  **uvm\_objection**( const std::string& *name* ); |

The constructor shall create a new objection instance with name *name*, if specified.

### Objection control

#### 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.

#### 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.

#### 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.

#### 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 will have updated their source counts--objections that they originated--and total counts--the total number of objections by them and all their descendants.

If the total objection count reaches zero, 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 **uvm\_component**::**all\_dropped** method (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 will 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.

#### 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.

### Callback hooks

#### 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 8.1.7.1).

#### 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 8.1.7.2).

#### 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 8.1.7.3).

### Objections status

#### 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).

#### 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.

#### 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*.

#### 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.

#### 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**.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_callback** : public **uvm\_object**  {  public:  **uvm\_callback**( const std::string& *name* = "uvm\_callback" );  bool **callback\_mode**( int *on* = -1 );  bool **is\_enabled**();  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_callback*  } *// namespace uvm* |

#### 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.

### Member functions

#### 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.

#### is\_enabled

|  |
| --- |
| bool **is\_enabled**(); |

The member function **is\_enabled** shall return 1 if the callback is enabled, otherwise it shall return 0.

#### 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.

## uvm\_callback\_iter

The class **uvm\_callback\_iter** is an iterator class for iterating over callback queues of a specific callback type.

### Class definition

|  |
| --- |
| namespace **uvm** {  template < typename T = **uvm\_object**, typename CB = **uvm\_callback**>  class **uvm\_callback\_iter**  {  public:  **uvm\_callback\_iter**( T\* *obj* );  CB\* **first**();  CB\* **last**();  CB\* **next**();  CB\* **prev**();  CB\* **get\_cb**();  }; *// class uvm\_callback*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the base object type with which the callback objects CB will be registered. This object must be a derivative of class **uvm\_object**.

### Template parameter CB

The template parameter T specifies the base callback type that will be managed by this callback class. The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm\_callback**.

### Constructor

|  |
| --- |
| **uvm\_callback\_iter**( T\* *obj* ); |

The constructor shall create a new callback iterator object. It is required that the object context be provided.

### Member functions

#### 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.

#### 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.

#### 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.

#### 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.

#### get\_cb

|  |
| --- |
| CB\* **get\_cb**(); |

The member function **get\_cb** shall return the last callback accessed via the call **first** or **next**.

## 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 14.4.2).

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.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T = **uvm\_object**, typename CB = **uvm\_callback**>  class **uvm\_callbacks** : public **uvm\_typed\_callbacks<**T>  {  public:  **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* |

### Template parameter T

The template parameter T specifies the base object type with which the callback objects CB will be registered. This object must be a derivative of class **uvm\_object**.

### Template parameter CB

The template parameter CB specifies the base callback type that will be managed by this callback class.The template parameter CB is optional. If not specified, the parameter is assigned the type **uvm\_callback**.

### Constructor

|  |
| --- |
| **uvm\_callbacks**(); |

The constructor shall create a new object of type **uvm\_callbacks**<T, CB>.

### Add/delete interface

#### 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 will be executed after previously added callbacks, else the callback will be executed ahead of previously added callbacks. The argument *cb* is the callback handle; it must be non-NULL, and if the callback has already been added to the object instance then a warning shall be issued.

#### 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 must already exist and must 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** (see 5.3.3.2) for more details on searching by name.

#### 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 must be non-NULL, and if the callback has already been removed from the object instance then a warning is issued.

#### 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** for more details on searching by name (see 5.3.3.2).

### 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 12.8).

#### 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 will be 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.

#### 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 will be 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.

#### 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 will be 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** will 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.

#### 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 will be 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** will 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.

### Debug

#### 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.

# 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\_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.

## 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 16.3.1).
* *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 will be sent to its associated file descriptor.

### 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* |

### 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.

### 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 **set\_report\_verbosity\_level**), 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

### Verbosity configuration

#### 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 will be filtered out. The severity and tag arguments check if the verbosity level has been modified for specific severity/tag combinations.

#### 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 will be ignored.

#### 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.

#### 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 an verbosity associated with id, which take precedence over an verbosity associated with a severity.

### Action configuration

#### 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*.

#### 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, shalltake precedence over the association set by this member function.

#### 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.

#### 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.

### File configuration

#### 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*.

#### 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.

#### 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.

#### 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.

#### 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.

### Override configuration

#### 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,** shalltake precedence over an upgrade or downgrade set by this member function.

#### 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*.

### Report handler configuration

#### 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.

#### 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.

#### 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 8.1.9).

## uvm\_report\_handler

The class **uvm\_report\_handler** is the class to which most methods 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 **uvm\_report\_object** 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 **uvm\_report\_object**::**set\_report\_handler**.

The relationship between an object of type **uvm\_report\_handler** and an object of type **uvm\_report\_server** is many to one.

### 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* |

### Constructor

|  |
| --- |
| **uvm\_report\_handler**(); |

The constructor shall create and initialize a new handler object.

### Member functions

#### 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.

#### 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.

#### 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 an file handle associated with the *severity*, return that. Else, return the default file handle.

#### 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 methods, **uvm\_report\_error**, **uvm\_report\_info**, **uvm\_report\_warning**, **uvm\_report\_fatal**, of class **uvm\_report\_object**.

#### 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”.

## 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**. None of its member functions are intended to be called by normal testbench code, although in some circumstances the virtual member functions **process\_report** and/or **compose\_uvm\_info** may be overloaded in a subclass.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_report\_server** : public **uvm\_object**  {  public:  **uvm\_report\_server**();  static void **set\_server**( **uvm\_report\_server**\* *server* );  static **uvm\_report\_server**\* **get\_server**();  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**();  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**();  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 **process\_report**( **uvm\_severity** *severity*,  const std::string& *name*,  const std::string& *id*,  const std::string& *message*,  **uvm\_action** *action*,  **UVM\_FILE** *file*,  const std::string& *filename*,  int *line*,  const std::string& *composed\_message*,  int *verbosity\_level*,  **uvm\_report\_object**\* *client* );  virtual std::string **compose\_message**( **uvm\_severity** *severity*,  const std::string& *name*,  const std::string& *id*,  const std::string& *message*,  const std::string& *filename*,  int *line* ) const;  virtual void **report\_summarize**( **UVM\_FILE** *file* = 0 ) const;  void **dump\_server\_state**() const;  }; *// class uvm\_report\_server*  } *// namespace uvm* |

### Constructor

|  |
| --- |
| **uvm\_report\_server**(); |

The constructor shall create a **uvm\_report\_server** object, if not already created. Else, it does nothing.

### Member functions

#### 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.

#### get\_server

|  |
| --- |
| static **uvm\_report\_server**\* **get\_server**(); |

The member function **get\_server** shall get the global report server. This member function will always return a valid handle to a report server.

#### 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.

#### get\_max\_quit\_count

|  |
| --- |
| int **get\_max\_quit\_count**() const; |

The member function **get\_max\_quit\_count** shall get the 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.

#### set\_quit\_count

|  |
| --- |
| void **set\_quit\_count**( int *quit\_count* ); |

The member function **set\_quit\_count** shall set the quit count, i.e., the number of COUNT actions, to the value *quit\_count*.

#### get\_quit\_count

|  |
| --- |
| int **get\_quit\_count**() const; |

The member function **get\_quit\_count** shall get the quit count, i.e., the number of COUNT actions.

#### 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.

#### 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.

#### 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.

#### 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*.

#### get\_severity\_count

|  |
| --- |
| int **get\_severity\_count**( **uvm\_severity** *severity* ) const; |

The member function **get\_severity\_count** shall get the counter value for the given *severity*.

#### 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*.*

#### reset\_severity\_counts

|  |
| --- |
| void **reset\_severity\_counts**(); |

The member function **reset\_severity\_counts** shall reset all severity counters to 0.

#### 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*.

#### get\_id\_count

|  |
| --- |
| int **get\_id\_count**( const std::string& *id* ) const; |

The member function **get\_id\_count** shall get the counter for reports with the given *id*.

#### 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.

#### process\_report

|  |
| --- |
| virtual void **process\_report**( **uvm\_severity** *severity*,  const std::string& *name*,  const std::string& *id*,  const std::string& *message*,  **uvm\_action** *action*,  **UVM\_FILE** *file*,  const std::string& *filename*,  int *line*,  const std::string& *composed\_message*,  int *verbosity\_level*,  **uvm\_report\_object**\* *client* ); |

The member function **process\_report** shall call the member function **compose\_message** to construct the actual message to be output. It then takes the appropriate action according to the value of action and file. This member function can be overloaded by an application to customize the way the reporting system processes reports and the actions enabled for them.

#### compose\_message

|  |
| --- |
| virtual std::string **compose\_message**( **uvm\_severity** *severity*,  const std::string& *name*,  const std::string& *id*,  const std::string& *message*,  const std::string& *filename*,  int *line* ) const; |

The member function **compose\_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.

#### report\_summarize

|  |
| --- |
| virtual void **report\_summarize**( **UVM\_FILE** *file* = 0 ) const; |

The member function **report\_summarize** shall output statistical information 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.

#### dump\_server\_state

|  |
| --- |
| void **dump\_server\_state**() const; |

The member function **dump\_server\_state** shall print server state information.

## 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 must implement the catch method in which the action to be taken on catching the report is specified. The catch method can return CAUGHT, in which case further processing of the report is immediately stopped, or return THROW in which case the (possibly modified) report is passed on to other registered catchers. The catchers are processed in the order in which they are registered.

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

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**.

### 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* |

### Constructor

|  |
| --- |
| **uvm\_report\_catcher**( const std::string& *name* = "uvm\_report\_catcher" ); |

The constructor shall create a new report catcherobject. The argument *name* is optional, but should generally be provided to aid in debugging.

### Current message state

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

#### get\_fname

|  |
| --- |
| std::string **get\_fname**() const; |

The member function **get\_fname** shall return the file name of the message.

#### get\_line

|  |
| --- |
| int **get\_line**() const; |

The member function **get\_line** shall return the line number of the message.

### Change message state

#### 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.

#### 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.

#### 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.

#### 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.

#### 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.

### Debug

#### 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.

#### 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**).

### Callback interface

#### 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 (see 13.4.3) can be used to access information about the current message being processed.

### Reporting

#### 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 will bypass any message catching callbacks.

#### 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 will bypass any message catching callbacks.

#### 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 will bypass any message catching callbacks.

#### 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 will bypass any message catching callbacks.

#### 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 will update the report\_server stats, possibly multiple times if the message is not CAUGHT.

#### 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**.

# Macros

UVM-SystemC defines macros for the following functions:

* Component and object registration
* Reporting
* Sequence execution
* Callbacks

## 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.

### 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* |

### 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.

### 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.

## 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.

### 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* |

### 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\_\_.

### 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\_\_.

### 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\_\_.

### 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\_\_.

## 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 10.3.6.2) and **finish\_item** (see 10.3.6.3). To start a sequence, it is recommended to use the member function **start** (see 10.3.4.1).

### 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* |

### 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 implemented as part of the UVM\_DO macro.

### 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*.

### 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.*

### 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.*

### 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.

### 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*.

### 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 *SEQUENCER*.

## Callback macros

The callback macros shall register and execute callbacks which are derived from class **uvm\_callbacks**.

### Macro definitions

|  |
| --- |
| namespace uvm {  #define **UVM\_REGISTER\_CB**( *T*, *CB* ) *implementation-defined*  #define **UVM\_DO\_CALLBACKS**( *T*, *CB*, *METHOD* ) *implementation-defined*  } *// namespace uvm* |

### 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.).

### 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 must 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.

# TLM interfaces

The TLM interfaces 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.

NOTE−UVM-SystemC does not yet define the TLM-2.0 blocking and non-blocking transport interfaces, direct memory interface (DMI), nor a debug transport interface. Use the SystemC TLM-2.0 interfaces instead.

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**
  + **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**
  + **uvm\_nonblocking\_get\_peek\_port**
* Analysis port and export classes:
  + **uvm\_analysis\_port**
  + **uvm\_analysis\_export**
  + **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**
  + **uvm\_seq\_item\_pull\_imp**

NOTE 1−There are no dedicated TLM-1 FIFO and FIFO interface classes defined in UVM-SystemC. Instead, the use the SystemC FIFO base classes **tlm**::**tlm\_fifo**<T> or **tlm**::**tlm\_analysis\_fifo**, or FIFO interfaces **tlm**::**tlm\_fifo\_debug\_if**, **tlm**::**tlm\_fifo\_put\_if**, and **tlm**::**tlm\_fifo\_get\_if** is recommended.

## 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**.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_blocking\_put\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_blocking\_put\_if**<T> >  {  public:  **uvm\_blocking\_put\_port**();  **uvm\_blocking\_put\_port**( const std::string& *name* );  virtual const std::string **get\_type\_name**() const;  virtual void **put**( const T& *val* );  }; *// class uvm\_blocking\_put\_port*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

### Constructor

|  |
| --- |
| **uvm\_blocking\_put\_port**();  **uvm\_blocking\_put\_port**( conststd::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.

### Member functions

#### 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**”.

#### 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.

## 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**.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_blocking\_get\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_blocking\_get\_if**<T> >  {  public:  **uvm\_blocking\_get\_port**();  **uvm\_blocking\_get\_port**( const std::string& *name* );  virtual const std::string **get\_type\_name**() const;  virtual void **get**( T& *val* );    }; *// class uvm\_blocking\_get\_port*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

### Constructor

|  |
| --- |
| **uvm\_blocking\_get\_port**();  **uvm\_blocking\_get\_port**( conststd::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.

### Member functions

#### 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**”.

#### 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.

## 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**.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_blocking\_peek\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_blocking\_peek\_if**<T> >  {  public:  **uvm\_blocking\_peek\_port**();  **uvm\_blocking\_peek\_port**( const std::string& *name* );  virtual const std::string **get\_type\_name**() const;  virtual void **peek**( T& *val* ) const;    }; *// class uvm\_blocking\_peek\_port*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

### Constructor

|  |
| --- |
| **uvm\_blocking\_peek\_port**();  **uvm\_blocking\_peek\_port**( conststd::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.

### Member functions

#### 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**”.

#### 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**.

## 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**.

### 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:  **uvm\_blocking\_get\_peek\_port**();  **uvm\_blocking\_get\_peek\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be received by the port.

### 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.

### Member functions

#### 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**”.

#### 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.

#### 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 (see member function **connect**).

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**.

## uvm\_nonblocking\_put\_port

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_nonblocking\_put\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_nonblocking\_put\_if**<T> >  {  public:  **uvm\_nonblocking\_put\_port**();  **uvm\_nonblocking\_put\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

### 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.

### Member functions

#### 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**”.

#### 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 method 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.

#### 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.

## uvm\_nonblocking\_get\_port

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_nonblocking\_get\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_nonblocking\_get\_if**<T> >  {  public:  **uvm\_nonblocking\_get\_port**();  **uvm\_nonblocking\_get\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

### 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.

### Member functions

#### 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**”.

#### 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 method 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.

#### 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.

## uvm\_nonblocking\_peek\_port

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_nonblocking\_peek\_port** : public **uvm\_port\_base**< **tlm**::**tlm\_nonblocking\_peek\_if**<T> >  {  public:  **uvm\_nonblocking\_peek\_port**();  **uvm\_nonblocking\_peek\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

### Constructor

|  |
| --- |
| **uvm\_nonblocking\_peek\_port**();  **uvm\_nonblocking\_peek\_port**( conststd::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.

### Member functions

#### 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**”.

#### 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 method 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.

#### 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.

## uvm\_nonblocking\_get\_peek\_port

### 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:  **uvm\_nonblocking\_get\_peek\_port**();  **uvm\_nonblocking\_get\_peek\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the port.

### 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.

### Member functions

#### 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**”.

#### 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 method 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.

#### 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.

#### 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 method 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.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_analysis\_port** : public **tlm**::**tlm\_analysis\_port**<T>  {  public:  **uvm\_analysis\_port**();  **uvm\_analysis\_port**( const std::string& *name* );  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* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

### 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 must have been connected to this port by the end of elaboration.

### Member functions

#### 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**”.

#### 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.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T>  class **uvm\_analysis\_export** : public **sc\_core**::**sc\_export**< **tlm**::**tlm\_analysis\_if**<T> >  {  public:  **uvm\_analysis\_export**();  **uvm\_analysis\_export**( const std::string& *name* );  virtual const std::string **get\_type\_name**() const;  virtual void **connect**( **tlm**::**tlm\_analysis\_if**<T>& *\_if* );  }; *// class uvm\_analysis\_export*  } *// namespace uvm* |

### Template parameter T

The template parameter T specifies the type of transaction to be communicated by the analysis port.

### 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 must have been connected to this port by the end of elaboration.

### Member functions

#### 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**”.

#### 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.

## 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.

### Class definition

|  |
| --- |
| namespace uvm {  template <typename T = int, typename IMP = int>  class **uvm\_analysis\_imp** : public **tlm::tlm\_analysis\_port<T>**  {  public:  **uvm\_analysis\_imp**();  **uvm\_analysis\_imp**( const std::string& *name* );  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* |

### 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**.

### Constructor

|  |
| --- |
| **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.

### Member functions

#### 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**”.

#### 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 1−The member function **connect** implements the same functionality as the SystemC member function **bind**.

#### 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.

## 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 a 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.

### 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* |

### 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 will use FIFOs of type **tlm**::**tlm\_fifo**.

### Ports and exports

#### 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.

#### 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.

#### 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.

#### 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.

#### 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**.

#### 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**.

#### 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**.

#### 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**.

#### 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**.

#### 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**.

### Constructor

|  |
| --- |
| **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.

## 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ, typename RSP = REQ>  class **uvm\_sqr\_if\_base** : public virtual **sc\_core**::**sc\_interface**  {  public:  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:  **uvm\_sqr\_if\_base**();  }; *// class uvm\_sqr\_if\_base*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively. These object types must be a derivative of class **uvm\_sequence\_item**.

### Member functions

#### 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 will block until an item is available. The following steps occur on this call:

1. 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.
2. The chosen sequence will return from member function **wait\_for\_grant** (see 10.3.6.4).
3. The chosen sequence’s member function **uvm\_sequence\_base**::**pre\_do** is called (see 10.3.4.4).
4. The chosen sequence item is randomized.
5. The chosen sequence’s member function **uvm\_sequence\_base**::**post\_do** is called (see 10.3.4.7).
6. Return with a reference to the item.

Once member function **get\_next\_item** is called, the member function **item\_done** must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

#### 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:

1. 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.
2. The chosen sequence will return from member function **uvm\_sequence\_base**::**wait\_for\_grant** (see 10.3.6.4).
3. The chosen sequence’s member function **uvm\_sequence\_base**::**pre\_do** is called (see 10.3.4.4).
4. The chosen sequence item is randomized.
5. The chosen sequence **uvm\_sequence\_base**::**post\_do** is called (see 10.3.4.7).
6. Return with a reference to the item.

Once the member function **try\_next\_item** is called, the member function **item\_done** must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

#### 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 will return.

The current item is removed from the sequencer FIFO.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must 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** will retrieve the current item that was obtained by member function **get\_next\_item**. After the member function **item\_done** is called, member function **peek** will cause the sequencer to arbitrate for a new item.

#### 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:

1. 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.
2. The chosen sequence will return from member function **uvm\_sequence\_base**::**wait\_for\_grant** (see 10.3.6.4).
3. The chosen sequence’s member function **uvm\_sequence\_base**::**pre\_do** is called (see 10.3.4.4).
4. The chosen sequence item is randomized.
5. The chosen sequence’s member function **uvm\_sequence\_base**::**post\_do** is called (see 10.3.4.7).
6. Indicate **item\_done** to the sequencer
7. 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**().

#### 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 will block until the sequencer has a new request. The following steps will occur if the sequencer FIFO is empty:

1. 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.
2. The chosen sequence will return from member function **uvm\_sequence\_base**::**wait\_for\_grant** (see 10.3.6.4).
3. The chosen sequence’s member function **uvm\_sequence\_base**::**pre\_do** is called (see 10.3.4.4).
4. The chosen sequence item is randomized.
5. The chosen sequence’s member function **uvm\_sequence\_base**::**post\_do** is called (see 10.3.4.7).

Once a request item has been retrieved and is in the sequencer FIFO, subsequent calls to member function **peek** will return the same item. The item will stay in the FIFO until either the member function **get** or **item\_done** is called.

#### 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 must 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 will not block. The response will be put into the sequence response queue or it will be sent to the sequence response handler.

## uvm\_seq\_item\_pull\_port

The class **uvm\_seq\_item\_pull\_port** shall define the port for use in sequencer-driver communication.

### 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:  **uvm\_seq\_item\_pull\_port**( const char\* name );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_seq\_item\_pull\_port*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.

### 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.

### Member functions

#### 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**”.

## uvm\_seq\_item\_pull\_export

The class **uvm\_seq\_item\_pull\_port** shall define the export for use in sequencer-driver communication.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ, typename RSP = REQ>  class **uvm\_seq\_item\_pull\_export** : public **uvm\_port\_base**< **uvm\_sqr\_if\_base**<REQ, RSP> >  {  public:  **uvm\_seq\_item\_pull\_export**( const char\* name );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_seq\_item\_pull\_export*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.

### 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.

### Member functions

#### 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**”.

## uvm\_seq\_item\_pull\_imp

The class **uvm\_seq\_item\_pull\_imp** shall implement the interface used in sequencer-driver communication.

### Class definition

|  |
| --- |
| namespace **uvm** {  template <typename REQ, typename RSP = REQ>  class **uvm\_seq\_item\_pull\_imp** : public **uvm\_port\_base**< **uvm\_sqr\_if\_base**<REQ, RSP> >  {  public:  **uvm\_seq\_item\_pull\_imp**( const char\* name );  virtual const std::string **get\_type\_name**() const;  }; *// class uvm\_seq\_item\_pull\_imp*  } *// namespace uvm* |

### Template parameters

The template parameters REQ and RSP specify the request and response object types, respectively.

### Constructor

|  |
| --- |
| **uvm\_seq\_item\_pull\_imp**( 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.

### Member functions

#### 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**”.

# Register layer

The UVM register layer defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a design-underverification.

A register model is typically composed of a hierarchy of blocks that usually map to the design hierarchy. Blocks contain registers, register files and memories.

The UVM register layer classes are not usable as-is. They only provide generic and introspection capabilities. They must 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 library.

The following register classes are defined:

* **uvm\_reg**: Base class for register
* **uvm\_reg\_block**: Base class for register blocks
* **uvm\_reg\_file**: Base class for register file
* **uvm\_reg\_fields**: Base class for register field
* **uvm\_reg\_indirect\_data**: Class for indirect data access
* **uvm\_reg\_fifo**: Class for register FIFO
* **uvm\_vreg**: Base class for virtual register
* **uvm\_vreg\_field**: Base class for virtual register field

Addition memory and allocation manager classes are defined:

* **uvm\_mem**: Base class for memory
* **uvm\_mem\_mam**: Class for memory allocation manager
* **uvm\_mem\_mam\_cfg**: Class to configure the memory managed by the memory allocation manager
* **uvm\_mem\_mam\_policy**: Class to specify the starting offset of the memory managed by the memory allocation manager
* **uvm\_mem\_region**: Class for allocated memory region descriptor

## Types

### uvm\_reg\_data\_t

2-state data value with **UVM\_REG\_DATA\_WIDTH** bits

### uvm\_reg\_data\_logic\_t

4-state data value with **UVM\_REG\_DATA\_WIDTH** bits

### uvm\_reg\_addr\_t

2-state address value with **UVM\_REG\_ADDR\_WIDTH** bits (maximum 64 bits, sc\_uint)

### uvm\_reg\_addr\_logic\_t

4-state address value with **UVM\_REG\_ADDR\_WIDTH** bits

### uvm\_reg\_byte\_en\_t

2-state byte\_enable value with **UVM\_REG\_BYTENABLE\_WIDTH** bits (why mapped to sc\_bv??)

### uvm\_reg\_cvr\_t

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. (Currently mapped to int)

* The following bits in the set are assigned as follows
* 0-7 UVM pre-defined coverage models
* 8-15 Coverage models defined by EDA vendors, implemented in a register model generator.
* 16-23 User-defined coverage models 24.. Reserved

### uvm\_hdl\_path\_slice

Slice of an HDL path. Struct that specifies 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.

## Enumerations

### uvm\_status\_e

Return 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.

### uvm\_path\_e

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 uvm\_reg\_predictor class.
* **UVM\_DEFAULT\_PATH** Operation specified by the context

### uvm\_check\_e

Read-only or read-and-check

* **UVM\_NO\_CHECK** Read only
* **UVM\_CHECK** Read and check

### uvm\_endianness\_e

Specifies 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

### uvm\_elem\_kind\_e

Type of element being read or written

* **UVM\_REG** Register
* **UVM\_FIELD** Field
* **UVM\_MEM** Memory location

### uvm\_access\_e

Type of operation begin performed

* **UVM\_READ** Read operation
* **UVM\_WRITE** Write operation

### uvm\_hier\_e

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

### uvm\_predict\_e

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

### uvm\_coverage\_model\_e

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

### uvm\_reg\_mem\_tests\_e

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.

## uvm\_hdl\_path\_concat

Concatenation of HDL variables

An vector of type **uvm\_hdl\_path\_slice** TODO check documentation specifying a concatenation of HDL variables that implement a register in the HDL.

Slices must be specified in most-to-least significant order. Slices must not overlap. Gaps may exist in the concatenation if portions of the registers are not implemented.

For example, the following register:

|  |
| --- |
| 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 Bits: 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0  +-+---+-------------+---+-------+  |A|xxx| B |xxx| C |  +-+---+-------------+---+-------+ |

If the register is implemented using a single HDL variable, the vector should specify a single slice with its offset and size specified as -1. For example:

|  |
| --- |
| concat.set({"r1", -1, -1}); |

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_hdl\_path\_concat**  {  public:  std::vector<**uvm\_hdl\_path\_slice**> slices;    void **set** std::vector<**uvm\_hdl\_path\_slice**> t );  void **add\_slice**( **uvm\_hdl\_path\_slice** slice);  void **add\_path**( const std::string& path,  unsigned int offset = -1,  unsigned int size = -1 );  };  } |

### Member functions

#### set

|  |
| --- |
| void **set** std::vector<**uvm\_hdl\_path\_slice**> t ); |

Initialize the concatenation using an array literal.

#### add\_slice

|  |
| --- |
| void **add\_slice**( **uvm\_hdl\_path\_slice** slice); |

Append the specified slice literal to the path concatenation.

#### add\_path

|  |
| --- |
| void **add\_path**( const std::string& path,  unsigned int offset = -1,  unsigned int size = -1 ); TODO unsingned int of -1??? |

Append the specified *path* to the path concatenation, for the specified number of bits at the specified *offset*.

### Data members

#### slices

|  |
| --- |
| std::vector<**uvm\_hdl\_path\_slice**> slices; |

Vector of individual slices, stored in most-to-least significant order.

## uvm\_reg\_block

The class **uvm\_reg\_block** shall define the block abstraction base class. A block represents a design hierarchy. It can contain registers, register files, memories and sub-blocks. A block has one or more address maps, each corresponding to a physical interface on the block.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_reg** : public **uvm\_object**  {  public:  //--------------------------------------------------------------------------  // Group: Initialization  //--------------------------------------------------------------------------  **uvm\_reg**( const std::string& name, // no default possible  unsigned int,  int has\_coverage );  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 string get\_name() const; TODO why not in code, part of UVM Code  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;  // TODO arguments swapped in UVM-SystemC compared to UVM-SV to enable defaults  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;  // TODO not in LRM? - Implementation defined??  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 uvm\_status\_e backdoor\_read\_func( 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 );  /////////////////////////////////////////////////////  // Implementation-defined member functions below,  // not part of UVM Class reference / LRM  /////////////////////////////////////////////////////  private:  UVM\_REGISTER\_CB(uvm\_reg, uvm\_reg\_cbs)  virtual void set\_parent( uvm\_reg\_block\* blk\_parent,  uvm\_reg\_file\* regfile\_parent );  uvm\_reg\_block\* get\_block() const;  virtual void add\_field( uvm\_reg\_field\* field );  virtual void add\_map( uvm\_reg\_map\* map );  void m\_lock\_model();  // Implementation defined - Introspection  virtual uvm\_reg\_map\* get\_local\_map( const uvm\_reg\_map\* map,  const std::string& caller = "" ) const;  virtual uvm\_reg\_map\* get\_default\_map( const std::string& caller = "" ) const;  std::string m\_get\_fields\_access( uvm\_reg\_map\* map ) const;  /////////////  // Implementation defined - Access  void m\_set\_busy( bool busy );  void m\_read( uvm\_status\_e& status,  uvm\_reg\_data\_t& value,  uvm\_path\_e path,  uvm\_reg\_map\* map,  uvm\_sequence\_base\* parent = NULL,  int prior = -1,  uvm\_object\* extension = NULL,  const std::string& fname = "",  int lineno = 0 );  void m\_atomic\_check\_lock( bool on );  virtual bool m\_check\_access( uvm\_reg\_item\* rw,  uvm\_reg\_map\_info\*& map\_info,  const std::string& caller );  bool m\_get\_is\_locked\_by\_field() const;  virtual bool do\_check( uvm\_reg\_data\_t expected,  uvm\_reg\_data\_t actual,  uvm\_reg\_map\* map);  virtual void do\_write( uvm\_reg\_item\* rw );  virtual void do\_read( uvm\_reg\_item\* rw );  virtual void do\_predict( uvm\_reg\_item\* rw,  uvm\_predict\_e kind = UVM\_PREDICT\_DIRECT,  uvm\_reg\_byte\_en\_t be = -1 );  // Implementation defined - Backdoor  void m\_sample( uvm\_reg\_data\_t data,  uvm\_reg\_data\_t byte\_en,  bool is\_read,  uvm\_reg\_map\* map );  // Implementation defined - UVM object  virtual void do\_print( const uvm\_printer& printer ) const;  virtual std::string convert2string() const;  virtual uvm\_object\* clone();  virtual void do\_copy( const uvm\_object& rhs );  virtual bool do\_compare( const uvm\_object& rhs,  const uvm\_comparer\* comparer ) const;  virtual void do\_pack( uvm\_packer& packer ) const;  virtual void do\_unpack( uvm\_packer& packer );  // data members  protected:  bool m\_locked;  uvm\_reg\_block\* m\_parent;  uvm\_reg\_file\* m\_regfile\_parent;  unsigned int m\_n\_bits;  unsigned int m\_n\_used\_bits;  typedef std::map<uvm\_reg\_map\*, bool> m\_maps\_t;  typedef m\_maps\_t::const\_iterator m\_maps\_citt;  m\_maps\_t m\_maps;  typedef std::vector<uvm\_reg\_field\*> m\_fields\_t; // Fields in LSB to MSB order  typedef m\_fields\_t::iterator m\_fields\_itt;  m\_fields\_t m\_fields;  private:  int m\_has\_cover;  int m\_cover\_on;  mutable std::string m\_fname;  mutable int m\_lineno;  sc\_core::sc\_host\_mutex m\_atomic; // semaphore  sc\_core::sc\_process\_handle m\_process;  bool m\_process\_valid;  bool m\_read\_in\_progress;  bool m\_write\_in\_progress;  bool m\_is\_busy;  bool m\_is\_locked\_by\_field;  mutable uvm\_reg\_backdoor\* m\_backdoor;  static unsigned int m\_max\_size;  //uvm\_object\_string\_pool< uvm\_queue<uvm\_hdl\_path\_concat\*> >\* m\_hdl\_paths\_pool;  std::map<std::string, std::vector<uvm\_hdl\_path\_concat> > m\_hdl\_paths\_pool;  typedef std::map<std::string, std::vector<uvm\_hdl\_path\_concat> >::const\_iterator m\_hdl\_paths\_pool\_itT;  protected:  bool m\_update\_in\_progress;  }; // class uvm\_reg  } // namespace uvm |

## uvm\_reg

The class **uvm\_reg** shall define the abstraction base class for register. 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.

### Class definition

|  |
| --- |
| namespace **uvm** {  class **uvm\_reg** : public **uvm\_object**  {  public:  // Group: Initialization  **uvm\_reg**( const std::string& *name*,  unsigned *int*,  int *has\_coverage* );  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 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;  // TODO arguments swapped in UVM-SystemC compared to UVM-SV to enable defaults  virtual int **get\_addresses**( std::valarray<**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;  // TODO not in LRM? - Implementation defined??  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**( 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 **uvm\_status\_e** **backdoor\_read\_func**( **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 );  uvm\_reg\_cvr\_t **build\_coverage**( **uvm\_reg\_cvr\_t** *models* );  virtual void **add\_coverage**( **uvm\_reg\_cvr\_t** *models* );  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;  virtual void **sample**( **uvm\_reg\_data\_t** *data*,  **uvm\_reg\_data\_t** *byte\_en*,  bool *is\_read*,  **uvm\_reg\_map**\* *map* );  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* |

TODO add register classes

# Global defines, typedefs and enumerations

This section lists the global defines, types and enumerations used in UVM-SystemC.

## Global defines

### 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.

### 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.

### 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 5.3.2.3).

## Type definitions (typedefs)

### 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 6.2.3.1), **uvm\_packer**::**pack\_field** (see 6.1.3.1) and **uvm\_packer**::**unpack\_field** (see 6.1.4.3).

### uvm\_integral\_t

The typedef **uvm\_bitstream\_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 6.2.3.2), **uvm\_packer::pack\_field\_int** (see 6.1.3.2) and **uvm\_packer::unpack\_field\_int** (see 6.1.4.2).

### UVM\_FILE

The typedef **uvm\_file** shall define the file descriptor which supports output streams.

### uvm\_report\_cb

The typedef **uvm\_report\_cb** is the alias for **uvm\_callbacks**<**uvm\_report\_object**, **uvm\_report\_catcher**>.

### uvm\_config\_int

The typedef **uvm\_config\_int** is the alias for **uvm\_config\_db**<**uvm\_bitstream\_t**>.

### uvm\_config\_string

The typedef **uvm\_config\_string** is the alias for **uvm\_config\_db**<**std::string**>.

### uvm\_config\_object

The typedef **uvm\_config\_object** is the alias for **uvm\_config\_db**<**uvm\_object\***>.

### uvm\_config\_wrapper

The typedef **uvm\_config\_wrapper** is the alias for **uvm\_config\_db**<**uvm\_object\_wrapper\***>.

## Enumeration

### 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.

### 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 will be terminated immediately.

### 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.

### 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.

### 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.

### 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 will 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**.

1. UVM-SystemVerilog features not included in   
   UVM-SystemC

(Informative)

The following is a list of major UVM-SystemVerilog features not available in UVM-SystemC. However, future UVM-SystemC implementations may address these topics. Note that this is not an exhaustive list.

## No field macros

UVM in SystemVerilog provides field macros to set up automation of fields inside a **uvm\_object**. The automation of the fields means that the fields automatically get an implementation for all of the mandatory member functions of **uvm\_object**. It is recommended not to use these field automation macros, because their implementation has impact on simulation performance and gives intransparent results for e.g. debugging. Therefore UVM-SystemC will not implement these field automation macros. As a consequence, an application needs to implement each of the mandatory member functions of **uvm\_object**: **do\_print**, **do\_pack**, **do\_unpack**, **do\_copy**, and **do\_compare**.

## No automated configuration

UVM-SystemC does not define automated configuration through the member function **apply\_config\_settings**. All configuration needs to be explicitly retrieved using **get\_config\_int**, **get\_config\_string**, or **get\_config\_object**.

## No transaction recording

UVM-SystemC does not define a transaction recording mechanism aligned with that of UVM-SystemVerilog. An application may use the existing transaction recording mechanism available in the SystemC Verification library (SCV) where appropriate.

## No register abstraction layer

UVM-SystemC does not define any register layer classes to create registers and memories.

## No constraint randomization and coverage classes

SystemVerilog offers the API for constraint randomization and coverage classes. Classes for constraint randomization of parameters and usage of coverage classes are not yet available in UVM-SystemC. Future extensions may integrate the SystemC Verification library (SCV) or CRAVE library to introduce randomization and a constraint solver.

## No assertions

Although not part of the UVM, but native functionality in SystemVerilog are assertions. These elements are not part of the SystemC language and therefore not supported in the UVM-SystemC implementation.

1. Renamed functions UVM-SystemC versus   
   UVM-SystemVerilog

(Informative)

Classes or member functions marked with symbol **°** are renamed in UVM-SystemC compared to the UVM 1.1 standard implemented in SystemVerilog, due to the incompatibility in case of reserved keywords in C/C++ or an inappropriate name in the context of SystemC base class or member function definitions. Table C.1 below shows the renamed classes and member functions and also the reference to the original UVM 1.1 name is given.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class name in UVM-SystemC** | **Class name  in UVM-SystemVerilog** | **Member function in UVM-SystemC** | **Method  in UVM-SystemVerilog** | **Section** |
| **uvm\_process\_phase** | **uvm\_task\_phase** |  |  | 3.11 |
| **uvm\_factory** |  | **do\_register** | **register** | 7.4.2.1 |
| **uvm\_phase** |  | **exec\_process** | **exec\_task** | 12.1.4.2 |
| **uvm\_callbacks** |  | **do\_delete** | **delete** | 12.9.5.3 |
| **uvm\_report\_catcher** |  | **do\_catch** | **catch** | 13.4.6.1 |

1. Terminology

(Informative)

## C.1 Definitions

**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.

**interface**: A class derived, directly or indirectly, from class **sc\_core**::**sc\_interface**. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper.

**interface proper**: An abstract class derived, directly or indirectly, from class **sc\_core**::**sc\_interface** but not derived from class **sc\_core**::**sc\_object**. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically it contains no function definitions and no data members.

**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.

**master**: This term has no precise technical definition in this standard, but it is used to mean a module or port that can take control of a memory-mapped bus in order to initiate bus traffic, or a component that can execute an autonomous software thread and thus initiate other system activity. Generally, a bus master would be an initiator.

**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.

**parent**:: The inverse relationship to child. Component A is the parent of component B if component B is a child of component A.

**parent sequence**: A sequence which contains one or more child sequences.

**port**: A TLM interface that defines the set of methods used for communication. Used in UVM to connect to an export.

**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.

**recipient**: The component that implements a callback or function that receives and processes a transaction. See also: sender.

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

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

**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 transactions to be executed and/or controls the execution of other sequences.

**sequencer**: An advanced stimulus generator which executes sequences that define the transactions 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.

**verification environment**: See environment.

**virtual sequence**: A conceptual term for a sequence that controls the execution of sequences on other sequencers.

## C.2 Acronyms and Abbreviations

AMS analog mixed signal

API application programming interface

CDV coverage-driven verification

CBCL common base class library

CLI command line interface

DUT device under test

DUV device under verification

EDA electronic design automation

FIFO first-in, first-out

HDL hardware description language

HVL high-level verification language

IP intellectual property

OSCI Open SystemC Initiative

SC SystemC

SCV SystemC Verification library

SV SystemVerilog

TLM transaction level modeling

UVC UVM Verification Component

UVM Universal Verification Methodology

VIP verification intellectual property

# Index

~

~uvm\_component\_name, destructor, 39

A

abstract, data member

class uvm\_packer, 46

action configuration

class uvm\_report\_object, 184

add, member function

class uvm\_callbacks, 175

class uvm\_phase, 158

add\_by\_name, member function

class uvm\_callbacks, 176

adjust\_name, member function

class uvm\_printer, 52

agent. definition, 240

all\_dropped, member function

class uvm\_component, 91

class uvm\_objection, 170

analysis\_export, export

class tuvm\_subscriber, 103

application, definition, 240

B

big\_endian, data member

class uvm\_packer, 46

blocking, definition, 240

body, member function

class uvm\_sequence\_base, 122

build\_phase, member function

class uvm\_component, 84

C

callback hooks

class uvm\_objection, 169

callback macros, 206

callback, definition, 240

callback\_mode, member function

class uvm\_callback, 172

callbacks

class uvm\_phase, 157

can\_get, member function

class uvm\_nonblocking\_get\_peek\_port, 219

class uvm\_nonblocking\_get\_port, 216

can\_peek, member function

class uvm\_nonblocking\_get\_peek\_port, 220

class uvm\_nonblocking\_peek\_port, 218

can\_put, member function

class uvm\_nonblocking\_put\_port, 215

check\_phase, member function

class uvm\_component, 88

child, definition, 240

clear, member function

class uvm\_objection, 167

clear\_response\_queue, member function

class uvm\_sequence\_base, 128

clone, member function

class uvm\_object, 28

compare, member function

class uvm\_object, 30

compare\_field, member function

class uvm\_comparer, 56

compare\_field\_int, member function

class uvm\_comparer, 56

compare\_field\_real, member function

class uvm\_comparer, 57

compare\_object, member function

class uvm\_comparer, 57

compare\_string, member function

class uvm\_comparer, 57

compare\_type, member function

class uvm\_comparer, 59

comparing

class uvm\_object, 30

component and object registration, 202

component, definition, 240

compose\_message, member function

class uvm\_report\_server, 194

configuration, definition, 240

configuration\_phase, member function

class uvm\_component, 86

connect, member function

class uvm\_analysis\_export, 222

class uvm\_analysis\_imp, 224

class uvm\_analysis\_port, 221

class uvm\_port\_base, 37

connect\_phase, member function

class uvm\_component, 84

construction

class uvm\_phase, 156

construction interface

class uvm\_component, 80

constructor

class uvm\_analysis\_export, 222

class uvm\_analysis\_imp, 223

class uvm\_analysis\_port, 220

class uvm\_blocking\_get\_peek\_port, 213

class uvm\_blocking\_get\_port, 210

class uvm\_blocking\_peek\_port, 212

class uvm\_blocking\_put\_port, 209

class uvm\_nonblocking\_get\_port, 216

class uvm\_nonblocking\_peek\_port, 217, 219

class uvm\_nonblocking\_put\_port, 215

class uvm\_report\_catcher, 197

class uvm\_report\_handler, 188

class uvm\_report\_server, 191

class uvm\_seq\_item\_pull\_export, 232

class uvm\_seq\_item\_pull\_imp, 233

class uvm\_seq\_item\_pull\_port, 231

class uvm\_tlm\_req\_rsp\_channel, 227

constructors

class uvm\_report\_object, 181

consumer, definition, 240

convert2string, member function

class uvm\_object, 28

copy, member function

class uvm\_object, 29

copying

class uvm\_object, 29

create, member function

class uvm\_component\_registry, 67

class uvm\_object, 27

class uvm\_object\_registry, 65

create\_component, member function

class uvm\_component, 92

class uvm\_component\_registry, 67

class uvm\_object\_wrapper, 63

create\_component\_by\_name, member function

class uvm\_factory, 73

create\_component\_by\_type, member function

class uvm\_factory, 73

create\_item, member function

class uvm\_sequence\_base, 125

create\_object, member function

class uvm\_component, 92

class uvm\_object\_registry, 64

class uvm\_object\_wrapper, 63

create\_object\_by\_name, member function

class uvm\_factory, 72

create\_object\_by\_type, member function

class uvm\_factory, 72

creation

class uvm\_factory, 72

class uvm\_object, 27

current\_grabber, member function

class uvm\_sequencer\_base, 108

D

debug

class uvm\_factory, 74

class uvm\_resource\_pool, 148

debug\_create\_by\_name, member function

class uvm\_factory, 74

debug\_create\_by\_type, member function

class uvm\_factory, 74

define\_domain, member function

class uvm\_component, 89

delete\_by\_name, member function

class uvm\_callbacks, 176

die, member function

class uvm\_root, 34

display, member function

class uvm\_callbacks, 177

display\_objections, member function

class uvm\_objection, 171

do\_catch, member function

class uvm\_report\_catcher, 200

do\_compare, member function

class uvm\_object, 30

do\_copy, member function

class uvm\_object, 29

do\_delete, member function

class uvm\_callbacks, 176

do\_kill, member function

class uvm\_sequence\_base, 125

do\_pack, member function

class uvm\_object, 31

do\_print, member function

class uvm\_object, 28

class uvm\_resource\_base, 142

do\_record, member function

class uvm\_object, 29

do\_register, member function

class uvm\_factory, 70

do\_unpack, member function

class uvm\_object, 32

driver, definition, 240

drop\_objection, member function

class uvm\_objection, 168

class uvm\_phase, 159

dropped, member function

class uvm\_component, 91

class uvm\_objection, 169

dump, member function

class uvm\_resource\_pool, 149

dump\_server\_state, member function

class uvm\_report\_server, 195

E

emit, member function

class uvm\_line\_printer, 54

class uvm\_printer, 51

class uvm\_table\_printer, 53

class uvm\_tree\_printer, 54

enable\_print\_topology, member function

class uvm\_root, 35

end\_of\_elaboration\_phase, member function

class uvm\_component, 84

environment, definition, 240

exec\_func, member function

class uvm\_phase, 157

exec\_process, member function

class uvm\_phase, 157

exec\_task. *See* exec\_process

execute, member function

class uvm\_bottomup\_phase, 163

class uvm\_process\_phase, 165

class uvm\_topdown\_phase, 164

execute\_item, member function

class uvm\_sequencer\_base, 106

exists, member function

uvm\_config\_db, 133

export, definition, 240

extract\_phase, member function

class uvm\_component, 88

F

factory

class uvm\_component, 92

factory method, definition, 240

fifo, definition, 240

file configuration

class uvm\_report\_object, 185

final\_phase, member function

class uvm\_component, 88

find, member function

class uvm\_phase, 157

class uvm\_root, 35

find\_all, member function

class uvm\_root, 35

find\_by\_name, member function

class uvm\_phase, 157

find\_override\_by\_name, member function

class uvm\_factory, 74

find\_override\_by\_type, member function

class uvm\_factory, 74

find\_unused\_resources, member function

class uvm\_resource\_pool, 148

finish\_item, member function

class uvm\_sequence\_base, 126

first, member function

class uvm\_callback\_iter, 173

foreign methodology, definition, 240

format\_action, member function

class uvm\_report\_handler, 190

format\_footer, member function

class uvm\_printer, 51

format\_header, member function

class uvm\_printer, 51

format\_row, member function

class uvm\_printer, 51

G

generator, definition, 240

get, member function

class uvm\_blocking\_get\_peek\_port, 213

class uvm\_blocking\_get\_port, 211

class uvm\_component\_registry, 67

class uvm\_object\_registry, 64

class uvm\_resource\_pool, 145

class uvm\_sequencer, 112

class uvm\_sqr\_if\_base, 229

uvm\_config\_db, 132

get\_action, member function

class uvm\_report\_catcher, 198

class uvm\_report\_handler, 189

get\_by\_name, member function

class uvm\_resource, 151

class uvm\_resource\_db, 135

class uvm\_resource\_pool, 146

get\_by\_type, member function

class uvm\_resource, 151

class uvm\_resource\_db, 135

class uvm\_resource\_pool, 147

get\_cb, member function

class uvm\_callback\_iter, 174

get\_child, member function

class uvm\_component, 81

get\_children, member function

class uvm\_component, 81

get\_client, member function

class uvm\_report\_catcher, 197

get\_common\_domain, member function

class uvm\_domain, 161

get\_current\_item, member function

class uvm\_sequencer\_param\_base, 110

uvm\_sequence, 129

get\_depth, member function

class uvm\_component, 82

class uvm\_sequence\_item, 118

get\_domain, member function

class uvm\_component, 89

class uvm\_phase, 159

get\_domain\_name, member function

class uvm\_phase, 159

get\_domains, member function

class uvm\_domain, 161

get\_drain\_time, member function

class uvm\_objection, 170

get\_field\_attribute, member function

class uvm\_comparer, 59

get\_file\_handle, member function

class uvm\_report\_handler, 189

get\_finish\_on\_completion, member function

class uvm\_root, 34

get\_first, member function

class uvm\_callbacks, 176

get\_first\_child, member function

class uvm\_component, 81

get\_fname, member function

class uvm\_report\_catcher, 198

get\_full\_name, member function

class uvm\_component, 81

class uvm\_object, 26

class uvm\_phase, 158

class uvm\_port\_base, 37

get\_highest\_precedence, member function

class uvm\_resource, 152

class uvm\_resource\_pool, 146

get\_id, member function

class uvm\_report\_catcher, 198

get\_id\_count, member function

class uvm\_report\_server, 194

get\_imp, member function

class uvm\_phase, 159

get\_inst\_count, member function

class uvm\_object, 27

get\_inst\_id, member function

class uvm\_object, 26

get\_is\_active, member function

class uvm\_agent, 100

get\_jump\_target, member function

class uvm\_phase, 160

get\_last, member function

class uvm\_callbacks, 177

get\_line, member function

class uvm\_report\_catcher, 198

get\_max\_messages, member function

class uvm\_comparer, 58

get\_max\_quit\_count, member function

class uvm\_report\_server, 192

get\_message, member function

class uvm\_report\_catcher, 198

get\_miscompare\_string, member function

class uvm\_comparer, 59

get\_name, member function

class uvm\_object, 26

class uvm\_port\_base, 37

get\_next, member function

class uvm\_callbacks, 177

get\_next\_child, member function

class uvm\_component, 81

get\_next\_item, member function

class uvm\_sequencer, 112

class uvm\_sqr\_if\_base, 228

get\_num\_children, member function

class uvm\_component, 81

get\_object\_type, member function

class uvm\_object, 27

get\_objection, member function

class uvm\_phase, 159

get\_objection\_count, member function

class uvm\_objection, 170

get\_objection\_total, member function

class uvm\_objection, 170

get\_objectors, member function

class uvm\_objection, 170

get\_packet\_size, member function

class uvm\_packer, 45

get\_parent, member function

class uvm\_component, 80

class uvm\_phase, 158

class uvm\_port\_base, 37

get\_parent\_sequence, member function

class uvm\_sequence\_item, 117

get\_peek\_request\_export, export

class uvm\_tlm\_req\_rsp\_channel, 226

get\_peek\_response\_export, export

class uvm\_tlm\_req\_rsp\_channel, 227

get\_phase\_type, member function

class uvm\_phase, 156

get\_policy, member function

class uvm\_comparer, 58

get\_prev, member function

class uvm\_callbacks, 177

get\_priority, member function

class uvm\_sequence\_base, 123

get\_quit\_count, member function

class uvm\_report\_server, 192

get\_report\_action, member function

class uvm\_report\_object, 184

get\_report\_catcher, member function

class uvm\_report\_catcher, 199

get\_report\_file\_handle, member function

class uvm\_report\_object, 185

get\_report\_handler, member function

class uvm\_report\_object, 187

get\_report\_verbosity\_level, member function

class uvm\_report\_object, 183

get\_request\_export, export

class uvm\_tlm\_req\_rsp\_channel, 226

get\_response, member function

uvm\_sequence, 130

get\_response\_export, export

class uvm\_tlm\_req\_rsp\_channel, 226

get\_response\_queue\_depth, member function

class uvm\_sequence\_base, 128

get\_response\_queue\_error\_report\_disabled, member function

class uvm\_sequence\_base, 127

get\_result, member function

class uvm\_comparer, 60

get\_root\_sequence, member function

class uvm\_sequence\_item, 118

get\_root\_sequence\_name, member function

class uvm\_sequence\_item, 118

get\_run\_count, member function

class uvm\_phase, 156

get\_schedule, member function

class uvm\_phase, 158

get\_schedule\_name, member function

class uvm\_phase, 158

get\_scope, member function

class uvm\_resource\_base, 142

get\_sequence\_path, member function

class uvm\_sequence\_item, 118

get\_sequence\_state, member function

class uvm\_sequence\_base, 121

get\_sequencer, member function

class uvm\_sequence\_item, 117

get\_server, member function

class uvm\_report\_server, 192

get\_severity, member function

class uvm\_comparer, 59

class uvm\_report\_catcher, 198

get\_severity\_count, member function

class uvm\_report\_server, 193

get\_state, member function

class uvm\_phase, 156

get\_transaction\_id, member function

class uvm\_transaction, 115

get\_type, member function

class uvm\_object, 27

class uvm\_resource, 150

get\_type\_handle, member function

class uvm\_resource, 150

class uvm\_resource\_base, 141

get\_type\_name, member function

class uvm\_agent, 99

class uvm\_analysis\_export, 222

class uvm\_analysis\_imp, 223

class uvm\_analysis\_port, 221

class uvm\_blocking\_get\_peek\_port, 213

class uvm\_blocking\_get\_port, 211

class uvm\_blocking\_peek\_port, 212

class uvm\_blocking\_put\_port, 209

class uvm\_callback, 172

class uvm\_component\_registry, 67

class uvm\_driver, 98

class uvm\_env, 100

class uvm\_monitor, 99

class uvm\_nonblocking\_get\_peek\_port, 219

class uvm\_nonblocking\_get\_port, 216

class uvm\_nonblocking\_peek\_port, 217

class uvm\_nonblocking\_put\_port, 215

class uvm\_object, 27

class uvm\_object\_registry, 64

class uvm\_object\_wrapper, 63

class uvm\_port\_base, 37

class uvm\_scoreboard, 102

class uvm\_seq\_item\_pull\_export, 232

class uvm\_seq\_item\_pull\_imp, 233

class uvm\_seq\_item\_pull\_port, 231

class uvm\_subscriber, 103

class uvm\_test, 101

get\_use\_response\_handler, member function

class uvm\_sequence\_base, 127

get\_use\_sequence\_info, member function

class uvm\_sequence\_item, 117

get\_uvm\_domain, member function

class uvm\_domain, 162

get\_uvm\_phases, member function

class uvm\_domain, 162

get\_uvm\_schedule, member function

class uvm\_domain, 161

get\_verbosity, member function

class uvm\_comparer, 58

class uvm\_report\_catcher, 198

get\_verbosity\_level, member function

class uvm\_report\_handler, 189

grab, member function

class uvm\_sequence\_base, 124

class uvm\_sequencer\_base, 107

H

has\_child, member function

class uvm\_component, 81

has\_do\_available, member function

class uvm\_sequencer\_base, 108

has\_lock, member function

class uvm\_sequence\_base, 125

class uvm\_sequencer\_base, 107

hierarchical reporting interface

class uvm\_component, 94

hierarchy interface

class uvm\_component, 80

host methodology, definition, 241

I

identification

class uvm\_object, 26

implementation, definition, 241

incr\_id\_count, member function

class uvm\_report\_server, 194

incr\_quit\_count, member function

class uvm\_report\_server, 192

incr\_severity\_count, member function

class uvm\_report\_server, 193

init\_access\_record, member function

class uvm\_resource\_base, 143

interface proper, definition, 241

interface, definition, 241

is, member function

class uvm\_phase, 157

is\_after, member function

class uvm\_phase, 157

is\_auditing, member function

class uvm\_resource\_options, 139

is\_before, member function

class uvm\_phase, 157

is\_blocked, member function

class uvm\_sequence\_base, 124

class uvm\_sequencer\_base, 107

is\_child, member function

class uvm\_sequencer\_base, 105

is\_enabled, member function

class uvm\_callback, 172

is\_grabbed, member function

class uvm\_sequencer\_base, 108

is\_item, member function

class uvm\_sequence\_item, 118

is\_null, member function

class uvm\_packer, 44

is\_quit\_count\_reached, member function

class uvm\_report\_server, 193

is\_read\_only, member function

class uvm\_resource\_base, 141

is\_relevant, member function

class uvm\_sequence\_base, 123

is\_tracing, member function

class uvm\_resource\_db\_options, 138

issue, member function

class uvm\_report\_catcher, 201

item\_done, member function

class uvm\_sequencer, 112

class uvm\_sqr\_if\_base, 229

J

jump, member function

class uvm\_phase, 160

jumping

class uvm\_phase, 160

K

kernel, definition, 241

kill, member function

class uvm\_sequence\_base, 125

knobs, data member

class uvm\_printer, 52

L

last, member function

class uvm\_callback\_iter, 173

lock, member function

class uvm\_sequence\_base, 124

class uvm\_sequencer\_base, 107

lookup

class uvm\_resource\_pool, 146

lookup, member function

class uvm\_component, 82

lookup\_name, member function

class uvm\_resource\_pool, 146

lookup\_regex, member function

class uvm\_resource\_pool, 147

lookup\_regex\_names, member function

class uvm\_resource\_pool, 147

lookup\_scope, member function

class uvm\_resource\_pool, 147

lookup\_type, member function

class uvm\_resource\_pool, 147

M

macros

class uvm\_component, 96

class uvm\_object, 32

main\_phase, member function

class uvm\_component, 86

master, definition, 241

master\_export, export

class uvm\_tlm\_req\_rsp\_channel, 227

match\_scope, member function

class uvm\_resource\_base, 142

member function, definition, 241

method, definition, 241

mid\_do, member function

class uvm\_sequence\_base, 122

monitor, definition, 241

N

next, member function

class uvm\_callback\_iter, 173

non blocking, definition, 241

notification

class uvm\_resource\_base, 141

O

objection control

class uvm\_objection, 167

objection interface

class uvm\_component, 91

objection status

class uvm\_objection, 170

operator const char\*(), operator

class uvm\_component\_name, 39

override configuration

class uvm\_report\_object, 186

override\_t, enum

class uvm\_resource\_types, 152

P

pack, member function

class uvm\_object, 30

pack\_bytes, member function

class uvm\_object, 30

pack\_field, member function

class uvm\_packer, 43

pack\_field\_int, member function

class uvm\_packer, 43

pack\_ints, member function

class uvm\_object, 31

pack\_object, member function

class uvm\_packer, 43

pack\_real, member function

class uvm\_packer, 43

pack\_string, member function

class uvm\_packer, 43

pack\_time, member function

class uvm\_packer, 43

packing

class uvm\_object, 30

parent sequence, definition, 241

parent, definition, 241

peek, member function

class uvm\_blocking\_get\_peek\_port, 214

class uvm\_blocking\_peek\_port, 212

class uvm\_sequencer, 113

class uvm\_sqr\_if\_base, 230

phase\_ended, member function

class uvm\_component, 89

phase\_ready\_to\_end, member function

class uvm\_component, 88

phase\_started, member function

class uvm\_component, 88

phasing interface

class uvm\_component, 82

physical, data member

class uvm\_packer, 45

port, definition, 241

post\_body, member function

class uvm\_sequence\_base, 123

post\_configuration\_phase, member function

class uvm\_component, 86

post\_do, member function

class uvm\_sequence\_base, 122

post\_main\_phase, member function

class uvm\_component, 87

post\_reset\_phase, member function

class uvm\_component, 85

post\_shutdown\_phase, member function

class uvm\_component, 87

post\_start, member function

class uvm\_sequence\_base, 123

post-run phases

class uvm\_component, 83

pre\_abort, member function

class uvm\_component, 96

pre\_body, member function

class uvm\_sequence\_base, 122

pre\_configuration\_phase, member function

class uvm\_component, 85

pre\_do, member function

class uvm\_sequence\_base, 122

pre\_main\_phase, member function

class uvm\_component, 86

pre\_reset\_phase, member function

class uvm\_component, 85

pre\_shutdown\_phase, member function

class uvm\_component, 87

pre\_start, member function

class uvm\_sequence\_base, 122

pre-run phases

class uvm\_component, 82

prev, member function

class uvm\_callback\_iter, 173

primary methodology, definition, 241

print, member function

class uvm\_factory, 75

class uvm\_object, 28

print\_accessors, member function

class uvm\_resource\_base, 143

print\_array\_footer, member function

class uvm\_printer, 52

print\_array\_header, member function

class uvm\_printer, 52

print\_array\_range, member function

class uvm\_printer, 52

print\_catcher, member function

class uvm\_report\_catcher, 199

print\_config, member function

class uvm\_component, 90

print\_config\_matches, member function

class uvm\_component, 91

print\_config\_with\_audit, member function

class uvm\_component, 90

print\_double, member function

class uvm\_printer, 50

print\_field, member function

class uvm\_printer, 49

print\_field\_int, member function

class uvm\_printer, 49

print\_generic, member function

class uvm\_printer, 51

print\_msg, member function

class uvm\_comparer, 58

print\_object, member function

class uvm\_printer, 50

print\_object\_header, member function

class uvm\_printer, 50

print\_override\_info, member function

class uvm\_component, 93

print\_real, member function

class uvm\_printer, 49

print\_resources, member function

class uvm\_resource\_pool, 148

print\_string, member function

class uvm\_printer, 50

print\_time, member function

class uvm\_printer, 51

print\_topology, member function

class uvm\_root, 35

printing

class uvm\_object, 28

priority

class uvm\_resource, 151

class uvm\_resource\_base, 142

priority\_t, enum

class uvm\_resource\_types, 152

process control interface

class uvm\_component, 89

process, definition, 241

process\_report, member function

class uvm\_report\_server, 194

put, member function

class uvm\_blocking\_put\_port, 210

class uvm\_sequencer, 113

class uvm\_sqr\_if\_base, 230

put\_request\_export, export

class uvm\_tlm\_req\_rsp\_channel, 226

put\_response\_export, export

class uvm\_tlm\_req\_rsp\_channel, 226

R

raise\_objection, member function

class uvm\_objection, 168

class uvm\_phase, 159

raised, member function

class uvm\_component, 91

class uvm\_objection, 169

read only interface

class uvm\_resource\_base, 141

read, member function

class uvm\_resource, 151

read/write interface

class uvm\_resource, 151

read\_by\_name, member function

class uvm\_resource\_db, 136

read\_by\_type, member function

class uvm\_resource\_db, 136

recipient, definition, 241

record, member function

class uvm\_object, 29

record\_read\_access, member function

class uvm\_resource\_base, 142

record\_write\_access, member function

class uvm\_resource\_base, 142

recording

class uvm\_object, 29

register. *See* do\_register

registering types

class uvm\_factory, 70

report handler configuration

class uvm\_report\_object, 187

report, member function

class uvm\_report\_handler, 189

report\_phase, member function

class uvm\_component, 88

report\_summarize, member function

class uvm\_report\_object, 195

reporting

class uvm\_report\_object, 181

reporting macros, 203

request, definition, 242

request\_ap, port

class uvm\_tlm\_req\_rsp\_channel, 225

reset\_phase, member function

class uvm\_component, 85

reset\_quit\_count, member function

class uvm\_report\_server, 193

reset\_report\_handler, member function

class uvm\_report\_object, 187

reset\_severity\_counts, member function

class uvm\_report\_server, 193

resource database interface

class uvm\_resource\_base, 141

response interface

class uvm\_sequence\_base, 127

response, definition, 242

response\_ap, port

class uvm\_tlm\_req\_rsp\_channel, 225

response\_handler, member function

class uvm\_sequence\_base, 127

resume, member function

class uvm\_component, 90

root sequence, definition, 242

rsp\_port, port

class uvm\_driver, 98

rsrc\_q\_t, typedef

class uvm\_resource\_types, 152

run\_phase, member function

class uvm\_component, 84

run\_test, global function, 23

run\_test, member function

class uvm\_root, 34

run-time phases

class uvm\_component, 83

S

schedule

class uvm\_phase, 158

scope interface

class uvm\_resource\_base, 141

scoreboard, definition, 242

send\_request, member function

class uvm\_sequencer\_base, 109

class uvm\_sequencer\_param\_base, 110

uvm\_sequence, 129

sender, definition, 242

seq\_item\_export, data member

class uvm\_sequencer, 112

seq\_item\_port, port

class uvm\_driver, 97

sequence control

class uvm\_sequence\_base, 123

sequence execution

class uvm\_sequence\_base, 121

sequence execution macros, 204

sequence item execution

class uvm\_sequence\_base, 125

sequence, definition, 242

sequencer, definition, 242

set

class uvm\_resource\_pool, 145

set priority

class uvm\_resource\_pool, 148

set, member function

class uvm\_resource, 150

class uvm\_resource\_db, 136

class uvm\_resource\_pool, 145

uvm\_config\_db, 132

set/get interface

class uvm\_resource, 150

set\_action, member function

class uvm\_report\_catcher, 199

set\_anonymous, member function

class uvm\_resource\_db, 136

set\_arbitration, member function

class uvm\_sequencer\_base, 108, 109

set\_default, member function

class uvm\_resource\_db, 135

set\_depth, member function

class uvm\_sequence\_item, 117

set\_domain, member function

class uvm\_component, 89

set\_drain\_time, member function

class uvm\_objection, 169

set\_field\_attribute, member function

class uvm\_comparer, 59

set\_finish\_on\_completion, member function

class uvm\_root, 34

set\_id, member function

class uvm\_report\_catcher, 199

set\_id\_count, member function

class uvm\_report\_server, 193

set\_id\_info, member function

class uvm\_sequence\_item, 117

set\_inst\_override, member function

class uvm\_component, 93

class uvm\_component\_registry, 68

class uvm\_object\_registry, 65

set\_inst\_override\_by\_name, member function

class uvm\_factory, 71

set\_inst\_override\_by\_type, member function

class uvm\_component, 92

class uvm\_factory, 70

set\_max\_messages, member function

class uvm\_comparer, 58

set\_max\_quit\_count, member function

class uvm\_report\_server, 192

set\_message, member function

class uvm\_report\_catcher, 199

set\_miscompare\_string, member function

class uvm\_comparer, 59

set\_name, member function

class uvm\_object, 26

set\_name\_override, member function

class uvm\_resource\_pool, 145

set\_override, member function

class uvm\_resource, 150

class uvm\_resource\_pool, 145

set\_parent\_sequence, member function

class uvm\_sequence\_item, 117

set\_phase\_imp, member function

class uvm\_component, 89

set\_policy, member function

class uvm\_comparer, 58

set\_priority, member function

class uvm\_resource, 151

class uvm\_resource\_base, 142

class uvm\_resource\_pool, 148

class uvm\_sequence\_base, 123

set\_priority\_name, member function

class uvm\_resource\_pool, 148

set\_priority\_type, member function

class uvm\_resource\_pool, 148

set\_quit\_count, member function

class uvm\_report\_server, 192

set\_read\_only, member function

class uvm\_resource\_base, 141

set\_report\_default\_file, member function

class uvm\_report\_object, 185

set\_report\_default\_file\_hier, member function

class uvm\_component, 95

set\_report\_handler, member function

class uvm\_report\_object, 187

set\_report\_id\_action, member function

class uvm\_report\_object, 185

set\_report\_id\_action\_hier, member function

class uvm\_component, 95

set\_report\_id\_file, member function

class uvm\_report\_object, 185

set\_report\_id\_file\_hier, member function

class uvm\_component, 95

set\_report\_id\_verbosity, member function

class uvm\_report\_object, 184

set\_report\_id\_verbosity\_hier, member function

class uvm\_component, 94

set\_report\_severity\_action, member function

class uvm\_report\_object, 184

set\_report\_severity\_action\_hier, member function

class uvm\_component, 94

set\_report\_severity\_file, member function

class uvm\_report\_object, 186

set\_report\_severity\_file\_hier, member function

class uvm\_component, 95

set\_report\_severity\_id\_action, member function

class uvm\_report\_object, 185

set\_report\_severity\_id\_action\_hier, member function

class uvm\_component, 95

set\_report\_severity\_id\_file, member function

class uvm\_report\_object, 186

set\_report\_severity\_id\_file\_hier, member function

class uvm\_component, 96

set\_report\_severity\_id\_override, member function

class uvm\_report\_object, 186

set\_report\_severity\_id\_verbosity, member function

class uvm\_report\_object, 184

set\_report\_severity\_id\_verbosity\_hier, member function

class uvm\_component, 94

set\_report\_severity\_override, member function

class uvm\_report\_object, 186

set\_report\_verbosity\_level, member function

class uvm\_report\_object, 184

set\_report\_verbosity\_level\_hier, member function

class uvm\_component, 96

set\_request, member function

class uvm\_sequence\_base, 126

set\_response\_queue\_depth, member function

class uvm\_sequence\_base, 128

set\_response\_queue\_error\_report\_disabled, member function

class uvm\_sequence\_base, 127

set\_scope, member function

class uvm\_resource\_base, 141

set\_sequencer, member function

class uvm\_sequence\_item, 117

set\_server, member function

class uvm\_report\_server, 191

set\_severity, member function

class uvm\_comparer, 59

class uvm\_report\_catcher, 199

set\_severity\_count, member function

class uvm\_report\_server, 193

set\_timeout, member function

class uvm\_root, 34

set\_transaction\_id, member function

class uvm\_transaction, 115

set\_type\_override, member function

class uvm\_component, 93

class uvm\_component\_registry, 67

class uvm\_object\_registry, 65

class uvm\_resource\_pool, 146

set\_type\_override\_by\_name, member function

class uvm\_factory, 72

set\_type\_override\_by\_type, member function

class uvm\_component, 92

class uvm\_factory, 71

set\_use\_sequence\_info, member function

class uvm\_sequence\_item, 116

set\_verbosity, member function

class uvm\_comparer, 58

class uvm\_report\_catcher, 199

shutdown\_phase, member function

class uvm\_component, 87

slave\_export, export

class uvm\_tlm\_req\_rsp\_channel, 227

sort\_by\_precedence, member function

class uvm\_resource\_pool, 146

spawned process, definition, 242

spell\_check, member function

class uvm\_resource\_pool, 145

sprint, member function

class uvm\_object, 28

start, member function

class uvm\_sequence\_base, 121

start\_item, member function

class uvm\_sequence\_base, 125

start\_of\_simulation\_phase, member function

class uvm\_component, 84

start\_phase\_sequence, member function

class uvm\_sequencer\_base, 106

starting\_phase, member function

class uvm\_sequence\_base, 128

state

class uvm\_phase, 156

stop\_sequences, member function

class uvm\_sequencer, 113

class uvm\_sequencer\_base, 108

summarize\_report\_catcher, member function

class uvm\_report\_catcher, 201

suspend, member function

class uvm\_component, 90

sync, member function

class uvm\_phase, 159

synchronization

class uvm\_phase, 159

T

template parameter CB

class uvm\_callback\_iter, 173

class uvm\_callbacks, 175

template parameter IF

class uvm\_ port\_base, 37

template parameter T

class uvm\_analysis\_export, 222

class uvm\_analysis\_imp, 223

class uvm\_analysis\_port, 220

class uvm\_blocking\_get\_peek\_port, 213

class uvm\_blocking\_get\_port, 210

class uvm\_blocking\_peek\_port, 212

class uvm\_blocking\_put\_port, 209

class uvm\_callback\_iter, 173

class uvm\_callbacks, 175

class uvm\_component\_registry, 66

class uvm\_config\_db, 132

class uvm\_nonblocking\_get\_peek\_port, 219

class uvm\_nonblocking\_get\_port, 216

class uvm\_nonblocking\_peek\_port, 217

class uvm\_nonblocking\_put\_port, 214

class uvm\_object\_registry, 64

class uvm\_resource, 150

class uvm\_resource\_db, 135

template parameters

class uvm\_driver, 97

class uvm\_seq\_item\_pull\_export, 232

class uvm\_seq\_item\_pull\_imp, 233

class uvm\_seq\_item\_pull\_port, 231

class uvm\_sequence, 129

class uvm\_sequencer, 112

class uvm\_sequencer\_param\_base, 110

class uvm\_sqr\_if\_base, 228

class uvm\_tlm\_req\_rsp\_channel, 225

test, definition, 242

testbench, definition, 242

trace\_mode, member function

class uvm\_objection, 167

transaction, definition, 242

transactor. *See* component

traverse, member function

class uvm\_bottomup\_phase, 163

class uvm\_process\_phase, 165

class uvm\_topdown\_phase, 164

try\_get, member function

class uvm\_nonblocking\_get\_peek\_port, 219

class uvm\_nonblocking\_get\_port, 216

try\_next\_item, member function

class uvm\_sequencer, 112

class uvm\_sqr\_if\_base, 229

try\_peek, member function

class uvm\_nonblocking\_get\_peek\_port, 219

class uvm\_nonblocking\_peek\_port, 218

try\_put, member function

class uvm\_nonblocking\_put\_port, 215

turn\_off\_auditing, member function

class uvm\_resource\_options, 139

turn\_off\_tracing, member function

class uvm\_resource\_db\_options, 138

turn\_on\_auditing, member function

class uvm\_resource\_options, 139

turn\_on\_tracing, member function

class uvm\_resource\_db\_options, 138

type and instance overrides types

class uvm\_factory, 70

type interface

class uvm\_resource, 150

U

ungrab, member function

class uvm\_sequence\_base, 124

class uvm\_sequencer\_base, 107

unlock, member function

class uvm\_sequence\_base, 124

class uvm\_sequencer\_base, 107

unpack, member function

class uvm\_object, 31

unpack\_bytes, member function

class uvm\_object, 31

unpack\_field, member function

class uvm\_packer, 44

unpack\_field\_int, member function

class uvm\_packer, 44

unpack\_ints, member function

class uvm\_object, 32

unpack\_object, member function

class uvm\_packer, 45

unpack\_real, member function

class uvm\_packer, 44

unpack\_string, member function

class uvm\_packer, 44

unpack\_time, member function

class uvm\_packer, 44

unpacking

class uvm\_object, 31

unsync, member function

class uvm\_phase, 160

use\_metadata, data member

class uvm\_packer, 46

use\_response\_handler, member function

class uvm\_sequence\_base, 127

user\_priority\_arbitration, member function

class uvm\_sequencer\_base, 105

utility functions

class uvm\_resource\_base, 142

uvm\_action, enum, 235

uvm\_active\_passive\_enum, enum, 236

uvm\_agent

class, 99

class definition, 99

constructor, 99

uvm\_analysis\_export

class, 221

class definition, 221

uvm\_analysis\_imp

class, 223

class definition, 223

uvm\_analysis\_port

class, 220

class definition, 220

uvm\_bitstream\_t, typedef, 234

uvm\_blocking\_get\_peek\_port

class, 212

uvm\_blocking\_get\_port

class, 210

class definition, 210

uvm\_blocking\_peek\_port

class, 211

class definition, 211, 212

uvm\_blocking\_put\_port

class, 209

class definition, 209

uvm\_bottomup\_phase

class, 162

class definition, 162

constructor, 163

overview, 19

uvm\_callback

class, 171

class definition, 171

constructor, 171

overview, 20

uvm\_callback\_iter

class, 172

class definition, 172

constructor, 173

overview, 20

uvm\_callbacks

class, 174

class definition, 174

constructor, 175

overview, 20

uvm\_comparer

class, 54

class definition, 54

overview, 18

uvm\_component

class, 76

class definition, 76

constructor, 80

uvm\_component\_name

class, 38

class definition, 38

constructor, 39

overview, 17

UVM\_COMPONENT\_PARAM\_UTILS, macro, 203

uvm\_component\_registry

class, 65

class definition, 66

overview, 18

UVM\_COMPONENT\_UTILS, macro, 203

uvm\_config\_db

class, 131

class definition, 131

constraints on usage, 132

overview, 19

uvm\_config\_int, typedef, 235

uvm\_config\_object, typedef, 235

uvm\_config\_string, typedef, 235

uvm\_config\_wrapper, typedef, 235

UVM\_CREATE, macro, 206

UVM\_CREATE\_ON, macro, 206

UVM\_DECLARE\_P\_SEQUENCER

macro, 113

UVM\_DECLARE\_P\_SEQUENCER, macro, 206

uvm\_default\_comparer, default policy object, 61

uvm\_default\_line\_printer, default policy object, 60

uvm\_default\_packer, default policy object, 60

uvm\_default\_printer, default policy object, 60

uvm\_default\_recorder, default policy object, 61

uvm\_default\_table\_printer, default policy object, 60

UVM\_DEFAULT\_TIMEOUT, global define, 234

uvm\_default\_tree\_printer, default policy object, 60

UVM\_DO, macro, 205

UVM\_DO\_CALLBACKS, macro, 206

UVM\_DO\_ON, macro, 205

UVM\_DO\_ON\_PRI, macro, 205

UVM\_DO\_PRI, macro, 205

uvm\_domain

class, 160

class definition, 161

constructor, 161

overview, 19

uvm\_driver

class, 97

class definition, 97

constructor, 98

uvm\_env

class, 100

class definition, 100

constructor, 100

UVM\_ERROR, macro, 204

uvm\_factory

class, 68

class definition, 68

overview, 18

UVM\_FATAL, macro, 204

UVM\_INFO, macro, 203

uvm\_integral\_t, typedef, 234

uvm\_line\_printer

constructor, 54

uvm\_line\_printer

class, 54

class definition, 54

UVM\_MAX\_STREAMBITS, global define, 234

uvm\_monitor

class, 98

class definition, 98

constructor, 98

uvm\_nonblocking\_get\_peek\_port

class, 218

class definition, 218

uvm\_nonblocking\_get\_port

class, 215

class definition, 215

uvm\_nonblocking\_peek\_port

class, 217

class definition, 217

uvm\_nonblocking\_put\_port

class, 214

class definition, 214

uvm\_object

class, 24

class definition, 24

constructor, 26

overview, 17

UVM\_OBJECT\_PARAM\_UTILS, macro, 202

uvm\_object\_registry

class, 63

class definition, 63

overview, 18

UVM\_OBJECT\_UTILS, macro, 202

uvm\_object\_wrapper

class, 62

class definition, 62

overview, 18

uvm\_objection

class, 165

class definition, 166

constructor, 167

overview, 20

uvm\_packer

class, 40

class definition, 40

overview, 18

UVM\_PACKER\_MAX\_BYTES, global define, 234

uvm\_phase

class, 154

class definition, 154

constructor, 156

overview, 19

uvm\_phase\_type, enum, 236

uvm\_port\_base

class, 36

class definition, 36

constructor, 37

overview, 17

uvm\_printer

class, 46

class definition, 46

overview, 17

uvm\_process\_phase

class, 164

class definition, 164

overview, 20

UVM\_REGISTER\_CB, macro, 206

uvm\_report\_catcher

class, 195

class definition, 195

overview, 20

uvm\_report\_cb, typedef, 234

uvm\_report\_enabled, member function

class uvm\_report\_object, 182

uvm\_report\_error, member function

class uvm\_report\_catcher, 200

class uvm\_report\_object, 183

uvm\_report\_fatal, member function

class uvm\_report\_catcher, 200

class uvm\_report\_object, 183

uvm\_report\_handler

class, 187

class definition, 188

overview, 20

uvm\_report\_info, member function

class uvm\_report\_catcher, 201

class uvm\_report\_object, 182

uvm\_report\_object

class, 178

class definition, 179

overview, 20

uvm\_report\_server

class, 190

class definition, 190

overview, 20

uvm\_report\_warning, member function

class uvm\_report\_catcher, 200

class uvm\_report\_object, 182

uvm\_resource

class, 149

class definition, 149

overview, 19

uvm\_resource\_base

class, 139

class definition, 139

constructor, 140

overview, 19

uvm\_resource\_db

class, 133

class definition, 133

overview, 19

uvm\_resource\_db\_options

class, 137

class definition, 137

uvm\_resource\_options

class, 138

class definition, 138

overview, 19

uvm\_resource\_pool

class, 143

class definition, 143

overview, 19

uvm\_resource\_types

class, 152

class definition, 152

uvm\_root

class, 33

class definition, 33

overview, 17

uvm\_scoreboard

class, 101

class definition, 101

constructor, 102

uvm\_seq\_item\_pull\_export

class, 231

class definition, 232

uvm\_seq\_item\_pull\_imp

class, 232

class definition, 232

uvm\_seq\_item\_pull\_port

class, 231

class definition, 231

uvm\_sequence

class, 128

class definition, 128

constructor, 129

overview, 19

uvm\_sequence\_base

class, 118

class definition, 119

constructor, 121

overview, 19

uvm\_sequence\_item

class, 115

class definition, 115

constructor, 116

overview, 19

uvm\_sequence\_state\_enum, enum, 236

uvm\_sequencer

class, 111

class definition, 111

constructor, 112

macros, 113

overview, 19

uvm\_sequencer\_base

class, 104

class definition, 104

constructor, 105

overview, 18

uvm\_sequencer\_param\_base

class, 109

class definition, 109

constructor, 110

overview, 19

uvm\_set\_config\_int, global function, 23

uvm\_set\_config\_string, global function, 23

uvm\_severity, enum, 235

uvm\_sqr\_if\_base

class, 227

class definition, 228

overview, 19

uvm\_subscriber

class, 102

class definition, 102

constructor, 103

uvm\_table\_printer

constructor, 53

uvm\_table\_printer

class, 52

class definition, 53

uvm\_task\_phase. *See* uvm\_process\_phase

uvm\_test

class, 100

class definition, 101

constructor, 101

uvm\_tlm\_req\_rsp\_channel

class, 224

class definition, 224

uvm\_top, data member

class uvm\_root, 35

uvm\_topdown\_phase

class, 163

class definition, 163

constructor, 164

overview, 19

uvm\_transaction

class, 114

class definition, 114

constructor, 115

overview, 19

uvm\_tree\_printer

constructor, 53

uvm\_tree\_printer

class, 53

class definition, 53

uvm\_verbosity, enum, 235

uvm\_void

class, 24

class definition, 24

overview, 17

UVM\_WARNING, macro, 204

V

verbosity configuration

class uvm\_report\_object, 183

verification environment. *See* environment

virtual sequence, definition, 242

W

wait\_for, member function

class uvm\_objection, 170

wait\_for\_grant, member function

class uvm\_sequence\_base, 126

class uvm\_sequencer\_base, 106

wait\_for\_item\_done, member function

class uvm\_sequence\_base, 126

class uvm\_sequencer\_base, 106

wait\_for\_relevant, member function

class uvm\_sequence\_base, 124

wait\_for\_sequence\_state, member function

class uvm\_sequence\_base, 121

wait\_for\_sequences, member function

class uvm\_sequencer\_base, 109

wait\_for\_state, member function

class uvm\_phase, 160

wait\_modified, member function

uvm\_config\_db, 133

wait\_modified, member function

class uvm\_resource\_base, 141

write, member function

class uvm\_resource, 151

write\_by\_name, member function

class uvm\_resource\_db, 136

write\_by\_type, member function

class uvm\_resource\_db, 137