VMM Register Abstraction Layer User Guide

G-2012.09
September 2012
VMM RAL Source Code Version 1.12

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2012 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

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 to comply with them.

Disclaimer

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

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1 Overview of the Register Abstraction Layer

Unsupported Features 1-3

2 Usage Model

Mirroring 2-4
Memories Are Not Mirrored 2-6

3 Installing and Using RAL

Minimum Requirements 3-1
Installation Instructions 3-2
Compiling and Running OpenVera Code With Vera 3-2
Compiling and Running OpenVera Code With VCS 3-3
Compiling and Running SystemVerilog Code 3-3
Using a VMM Open Source Distribution 3-4

4 Register and Memory Specification

Systems, Blocks, Registers, and Fields 4-2
Reusability and Composition 4-3
Naming 4-5
Hierarchical Descriptions and Composition 4-9
Arrays and Register Files 4-11

Virtual Fields and Virtual Registers 4-13
Multiple Physical Interfaces 4-16

5 Code Generation

Generating a RAL Model 5-1
Options 5-2
Understanding the Generated Model 5-4
Fields. 5-4
Registers 5-4
Arrays 5-6
Register Files 5-6
Virtual Registers 5-8
Memories. 5-9
Blocks 5-10
Systems. 5-13
Inserting User Defined Code in the Generated Model 5-15
UML Diagram 5-18

6 Physical-layer Transactors

Concurrently Executing Generic Transactions 6-5
Resetting Transactors and Disabling Threads 6-7

7 Verification Environment

Top-level Module 7-2
SystemVerilog Interfaces 7-3
OpenVera Interfaces 7-3
Environment Class 7-4
RAL Model. 7-4
Physical-level Transactors 7-5
Reset Task 7-7
Implicitly Phased Environment 7-7

8 Executing Pre-defined Tests

SystemVerilog 8-2
Using a VMM Open Source Installation 8-3
OpenVera with VCS 8-3
OpenVera with Vera 8-4
Predefined Tests 8-4

9 DUT Configuration

10 Back-door Access

Back-door Read/Write vs. Peek/Poke 10-2
Generated Backdoors 10-4
Arrays 10-6
VHDL. 10-10
Target Structures. 10-13
Reserved RALF Keywords in Backdoor Path 10-21
Support for Separate Compile. 10-22
RAL Backdoor Callbacks. 10-24
User-defined Backdoors 10-29
Implementing a Register Backdoor in OpenVera with Verilog DUT 10-30
Implementing a Register Backdoor in SystemVerilog 10-34
Implementing a Memory Backdoor in OpenVera with DWMM 10-38
Implementing a Memory Backdoor in OpenVera with Verilog DUT 10-39
Implementing a Memory Backdoor in SystemVerilog 10-43

11 User-defined Field Access

Field Usage vs. Field Behavior 11-3

12 Memories with ECC

ECC Backdoor Access 12-2
ECC Physical Access 12-3

13 Non-linear, Non-mapped Access

User-defined Register Access 13-2
User-defined Memory Access 13-4

14 Functional Coverage Model

Predefined Functional Coverage Models 14-3
Register Bits 14-4
Address Map 14-5
Field Values. 14-6
RALF cover attribute 14-9
User-defined Functional Coverage Model 14-11

15 Randomizing Field Values

Adding Constraints 15-3
Relaxing Constraints 15-5

16 Maximum Data Size

Maximum Data Size in SystemVerilog 16-2
Maximum Data Size in OpenVera 16-3
Maximum Data Size using Vera 16-3
Maximum Data Size Using VCS 16-4

17 C Interface

Entry Point 17-2
Application Entry Point 17-3
Service Entry Point 17-5
Execution Timeline 17-6
Writing Firmware Code 17-7
Accessing Registers 17-8
Accessing Fields 17-10
Accessing Memories 17-11

18 Using the RALF C++ API

RALF C++ API Code Example 18-2
Compiling 18-3
Compilation Command 18-4
Compiling in RHEL32 18-4
Compiling in Solaris. 18-5
Specifying the Library Path 18-5

19 Generating RALF from IP-XACT

Definition of the IP-XACT Schema 19-2
RALF File Description Mechanism 19-2
Supported IP-XACT Schema 19-4
Generic RALF Features and IP-XACT Mapping 19-4
Specific RALF Features and IP-XACT Mapping 19-5
Limitations of IP-XACT to RALF Feature Mapping 19-7

20 RAL C Interface

RAL C API Versions 20-2
Entry Point 20-2
Overview of Entry Points 20-3
Application Entry Point 20-4
Service Entry Point 20-6
Execution Timeline 20-6
Writing Firmware Code 20-8
Firmware Code Overview 20-8
Accessing Registers 20-10
Accessing Fields 20-11
Accessing Memories 20-12

21 Automatic Mirror Updating Techniques

Direct Update 21-2
Passive Update 21-6

22 Sub-Register Access

Support for Sub-Register Accesses 22-3
Backward Compatibility 22-4

A RALF Syntax

Grammar Notation A-2
Reserved Words A-2
Useful Tcl Commands A-3
Tcl Syntax and FAQ A-4
RALF Construct Summary A-6
field A-6
register A-12
regfile. A-19
memory A-23
virtual register A-27
block A-29
system A-37

B RAL Classes

RAL Classes Based on vmm_object B-2
RAL Class Summary B-4
vmm_mam B-5
vmm_mam::new() B-6
vmm_mam::log B-8
vmm_mam::default_alloc B-10
vmm_mam::reconfigure() B-12
vmm_mam::reserve_region() B-14
vmm_mam::request_region() B-16
vmm_mam::release_region() B-18
vmm_mam::release_all_regions() B-20
vmm_mam::psdisplay() B-22
vmm_mam::for_each() B-23
vmm_mam::get_memory() B-25

vmm_mam_allocator B-27
vmm_mam_allocator::len B-28
vmm_mam_allocator::min_offset B-30
vmm_mam_allocator::max_offset B-32
vmm_mam_allocator::in_use[$] B-34
vmm_mam_allocator::start_offset B-36
vmm_mam_cfg B-38
vmm_mam_cfg::n_bytes B-39
vmm_mam_cfg::start_offset B-41
vmm_mam_cfg::end_offset B-43
vmm_mam_cfg::mode B-45
vmm_mam_cfg::locality B-47
vmm_mam_region B-49
vmm_mam_region::get_start_offset() B-50
vmm_mam_region::get_end_offset() B-52
vmm_mam_region::get_len() B-53
vmm_mam_region::get_n_bytes() B-54
vmm_mam_region::get_memory() B-55
vmm_mam_region::get_virtual_registers() B-56
vmm_mam_region::psdisplay() B-57
vmm_mam_region::read() B-58
vmm_mam_region::write() B-60
vmm_mam_region::burst_read() B-62
vmm_mam_region::burst_write() B-65
vmm_mam_region::peek() B-68
vmm_mam_region::poke() B-70
vmm_ral B-72
vmm_ral::path_e B-73
vmm_ral::access_e B-75
vmm_ral::check_e B-79
vmm_ral::endianness_e B-80
vmm_ral::reset_e B-82
vmm_ral::coverage_e B-84
vmm_ral_access B-86
vmm_ral_access::set_model() B-87
vmm_ral_access::get_model() B-89
vmm_ral_access::add_xactor() B-90

vmm_ral_access::read() B-92
vmm_ral_access::write() B-94
vmm_ral_access::burst_read() B-96
vmm_ral_access::burst_write() B-98
vmm_ral_access::default_path B-100
vmm_ral_access::set_by_name() B-102
vmm_ral_access::get_by_name() B-104
vmm_ral_access::read_by_name() B-106
vmm_ral_access::write_by_name() B-108
vmm_ral_access::read_mem_by_name() B-110
vmm_ral_access::write_mem_by_name() B-112
vmm_ral_block_or_sys B-114
vmm_ral_block_or_sys::log B-116
vmm_ral_block_or_sys::get_name() B-117
vmm_ral_block_or_sys::get_type() B-118
vmm_ral_block_or_sys::get_fullname() B-120
vmm_ral_block_or_sys::get_domains() B-121
vmm_ral_block_or_sys::get_parent_domain() B-123
vmm_ral_block_or_sys::get_external_domain() B-124
vmm_ral_block_or_sys::get_n_tops() B-125
vmm_ral_block_or_sys::get_parent() B-126
vmm_ral_block_or_sys::get_base_addr() B-127
vmm_ral_block_or_sys::get_top() B-128
vmm_ral_block_or_sys::C_addr_of() B-129
vmm_ral_block_or_sys::set_offset() B-130
vmm_ral_block_or_sys::get_n_bytes() B-132
vmm_ral_block_or_sys::get_endian() B-134
vmm_ral_block_or_sys::default_access B-136
vmm_ral_block_or_sys::get_default_access() B-138
vmm_ral_block_or_sys::display() B-140
vmm_ral_block_or_sys::psdisplay() B-141
vmm_ral_block_or_sys::psdisplay_domain() B-143
vmm_ral_block_or_sys::get_fields() B-145
vmm_ral_block_or_sys::get_field_by_name() B-147
vmm_ral_block_or_sys::get_registers() B-149
vmm_ral_block_or_sys::get_reg_by_name() B-150

vmm_ral_block_or_sys::get_reg_by_offset() B-152
vmm_ral_block_or_sys::get_virtual_fields() B-154
vmm_ral_block_or_sys::get_virtual_field_by_name() B-155
vmm_ral_block_or_sys::get_virtual_registers() B-156
vmm_ral_block_or_sys::get_vreg_by_name() B-157
vmm_ral_block_or_sys::get_memories() B-158
vmm_ral_block_or_sys::get_mem_by_name() B-159
vmm_ral_block_or_sys::get_constraints() B-160
vmm_ral_block_or_sys::get_n_tops(); B-162
vmm_ral_block_or_sys::get_top(); B-163
vmm_ral_block_or_sys::has_cover() B-164
vmm_ral_block_or_sys::set_cover() B-165
vmm_ral_block_or_sys::is_cover_on() B-167
vmm_ral_block_or_sys::reset() B-169
vmm_ral_block_or_sys::needs_update() B-170
vmm_ral_block_or_sys::update() B-172
vmm_ral_block_or_sys::mirror() B-173
vmm_ral_block_or_sys::readmemh() B-175
vmm_ral_block_or_sys::writememh() B-176
vmm_ral_block_or_sys::set_attribute() B-177
vmm_ral_block_or_sys::get_attribute() B-178
vmm_ral_block_or_sys::get_all_attributes() B-179
vmm_ral_block_or_sys::ral_power_down() B-180
vmm_ral_block_or_sys::ral_power_up() B-181
vmm_ral_block B-182
vmm_ral_block::new() B-183
vmm_ral_block::sample_field_values() B-185
vmm_ral_env B-186
vmm_ral_env::new() B-187
vmm_ral_env::ral B-188
vmm_ral_env::hw_reset B-190
vmm_ral_env::sw_reset B-192
vmm_ral_env::reset_dut B-193
vmm_ral_field B-195
vmm_ral_field::log B-196
vmm_ral_field::get_name() B-197
vmm_ral_field::get_fullname() B-198

vmm_ral_field::get_register() B-199
vmm_ral_field::get_lsb_pos_in_register() B-200
vmm_ral_field::get_n_bits() B-201
vmm_ral_field::get_access() B-202
vmm_ral_field::set_access() B-204
vmm_ral_field::display() B-206
vmm_ral_field::psdisplay() B-207
vmm_ral_field::set() B-208
vmm_ral_field::predict() B-210
vmm_ral_field::get() B-212
vmm_ral_field::reset() B-213
vmm_ral_field::set_reset() B-214
vmm_ral_field::needs_update() B-216
vmm_ral_field::read() B-217
vmm_ral_field::write() B-219
vmm_ral_field::peek() B-223
vmm_ral_field::poke() B-225
vmm_ral_field::mirror() B-227
vmm_ral_field::append_callback() B-229
vmm_ral_field::prepend_callback() B-231
vmm_ral_field::unregister_callback() B-233
vmm_ral_field_callbacks B-235
vmm_ral_field_callbacks::pre_write() B-236
vmm_ral_field_callbacks::post_write() B-238
vmm_ral_field_callbacks::pre_read() B-240
vmm_ral_field_callbacks::post_read() B-242
vmm_ral_mem B-244
vmm_ral_mem::log B-245
vmm_ral_mem::mam B-246
vmm_ral_mem::get_name() B-247
vmm_ral_mem::get_fullname() B-248
vmm_ral_mem::get_domains() B-249
vmm_ral_mem::get_block() B-250
vmm_ral_mem::get_offset_in_block() B-251
vmm_ral_mem::get_address_in_system() B-252
vmm_ral_mem::get_size() B-253
vmm_ral_mem::get_n_bits() B-254

vmm_ral_mem::get_n_bytes() B-255
vmm_ral_mem::get_access() B-256
vmm_ral_mem::get_rights() B-258
vmm_ral_mem::get_virtual_fields() B-259
vmm_ral_mem::get_virtual_field_by_name() B-260
vmm_ral_mem::get_virtual_registers() B-261
vmm_ral_mem::get_vreg_by_name() B-262
vmm_ral_mem::display() B-263
vmm_ral_mem::psdisplay() B-264
vmm_ral_mem::psdisplay_domain() B-265
vmm_ral_mem::set_frontdoor() B-266
vmm_ral_mem::get_frontdoor() B-267
vmm_ral_mem::set_backdoor() B-268
vmm_ral_mem::get_backdoor() B-270
vmm_ral_mem::init() B-271
vmm_ral_mem::init_e B-272
vmm_ral_mem::read() B-274
vmm_ral_mem::write() B-276
vmm_ral_mem::burst_read() B-278
vmm_ral_mem::burst_write() B-281
vmm_ral_mem::peek() B-284
vmm_ral_mem::poke() B-286
vmm_ral_mem::readmemh() B-288
vmm_ral_mem::writememh() B-289
vmm_ral_mem::append_callback() B-290
vmm_ral_mem::prepend_callback() B-292
vmm_ral_mem::unregister_callback() B-294
vmm_ral_mem::set_attribute() B-296
vmm_ral_mem::get_attribute() B-297
vmm_ral_mem::get_all_attributes() B-298
vmm_ral_mem::ral_power_down() B-299
vmm_ral_mem::ral_power_up() B-300
vmm_ral_mem_backdoor B-301
vmm_ral_mem_backdoor::read() B-302
vmm_ral_mem_backdoor::write() B-304
vmm_ral_mem_backdoor::pre_write() B-306

vmm_ral_mem_backdoor::post_write() B-308
vmm_ral_mem_backdoor::pre_read() B-309
vmm_ral_mem_backdoor::post_read() B-310
vmm_ral_mem_burst B-312
vmm_ral_mem_burst::n_beats B-313
vmm_ral_mem_burst::start_offset B-315
vmm_ral_mem_burst::incr_offset B-316
vmm_ral_mem_burst::max_offset B-317
vmm_ral_mem_burst::user_data B-318
vmm_ral_mem_callbacks B-319
vmm_ral_mem_callbacks::pre_write() B-320
vmm_ral_mem_callbacks::post_write() B-322
vmm_ral_mem_callbacks::pre_read() B-324
vmm_ral_mem_callbacks::post_read() B-326
vmm_ral_mem_callbacks::pre_burst() B-328
vmm_ral_mem_callbacks::post_burst() B-329
vmm_ral_mem_backdoor::prepend_callback() B-330
vmm_ral_mem_backdoor::unregister_callback() B-332
vmm_ral_mem_backdoor_callbacks B-334
vmm_ral_mem_backdoor_callbacks::pre_write() B-335
vmm_ral_mem_backdoor_callbacks::post_write() B-336
vmm_ral_mem_backdoor_callbacks::pre_read() B-337
vmm_ral_mem_backdoor_callbacks::post_read() B-338
vmm_ral_mem_backdoor_callbacks::encode() B-339
vmm_ral_mem_backdoor_callbacks::decode() B-340
vmm_ral_mem_frontdoor B-341
vmm_ral_mem_frontdoor::read() B-342
vmm_ral_mem_frontdoor::write() B-344
vmm_ral_mem_frontdoor::burst_read() B-346
vmm_ral_mem_frontdoor::burst_write() B-348
vmm_ral_reg B-350
vmm_ral_reg::log B-351
vmm_ral_reg::get_name() B-352
vmm_ral_reg::get_fullname() B-353
vmm_ral_reg::get_n_domains() B-354
vmm_ral_reg::get_domains() B-355
vmm_ral_reg::get_rights() B-356

vmm_ral_reg::get_block() B-357
vmm_ral_reg::get_offset_in_block() B-358
vmm_ral_reg::get_address_in_system() B-359
vmm_ral_reg::get_n_bytes() B-360
vmm_ral_reg::get_constraints() B-361
vmm_ral_reg::display() B-363
vmm_ral_reg::psdisplay() B-364
vmm_ral_reg::psdisplay_domain() B-365
vmm_ral_reg::get_fields() B-366
vmm_ral_reg::get_field_by_name() B-367
vmm_ral_reg::set_frontdoor() B-368
vmm_ral_reg::get_frontdoor() B-369
vmm_ral_reg::set_backdoor() B-370
vmm_ral_reg::get_backdoor() B-371
vmm_ral_reg::set() B-372
vmm_ral_reg::predict() B-374
vmm_ral_reg::get() B-376
vmm_ral_reg::reset() B-377
vmm_ral_reg::sample_field_values() B-378
vmm_ral_reg::needs_update() B-379
vmm_ral_reg::update() B-380
vmm_ral_reg::mirror() B-382
vmm_ral_reg::read() B-384
vmm_ral_reg::write() B-386
vmm_ral_reg::peek() B-388
vmm_ral_reg::poke() B-390
vmm_ral_reg::append_callback() B-392
vmm_ral_reg::prepend_callback() B-394
vmm_ral_reg::unregister_callback() B-396
vmm_ral_reg::get_reset() B-398
vmm_ral_reg::set_attribute() B-399
vmm_ral_reg::get_attribute() B-400
vmm_ral_reg::get_all_attributes() B-401
vmm_ral_reg_backdoor B-402
vmm_ral_reg_backdoor::read() B-403
vmm_ral_reg_backdoor::write() B-405

vmm_ral_reg_backdoor::pre_write() B-407
vmm_ral_reg_backdoor::post_write() B-409
vmm_ral_reg_backdoor::pre_read() B-410
vmm_ral_reg_backdoor::post_read() B-411
vmm_ral_reg_callbacks B-413
vmm_ral_reg_callbacks::pre_write() B-414
vmm_ral_reg_callbacks::post_write() B-416
vmm_ral_reg_callbacks::pre_read() B-418
vmm_ral_reg_callbacks::post_read() B-421
vmm_ral_reg_backdoor::append_callback() B-423
vmm_ral_reg_backdoor::prepend_callback() B-425
vmm_ral_reg_backdoor::unregister_callback() B-427
vmm_ral_reg_backdoor_callbacks B-429
vmm_ral_reg_backdoor_callbacks::pre_write() B-430
vmm_ral_reg_backdoor_callbacks::post_write() B-431
vmm_ral_reg_backdoor_callbacks::pre_read() B-432
vmm_ral_reg_backdoor_callbacks::post_read() B-433
vmm_ral_reg_backdoor_callbacks::encode() B-434
vmm_ral_reg_backdoor_callbacks::decode() B-435
vmm_ral_reg_frontdoor B-436
vmm_ral_reg_frontdoor::read() B-437
vmm_ral_reg_frontdoor::write() B-439
vmm_ral_sys B-441
vmm_ral_sys::new() B-442
vmm_ral_sys::get_blocks() B-444
vmm_ral_sys::get_all_blocks() B-446
vmm_ral_sys::get_block_by_name() B-448
vmm_ral_sys::get_subsys() B-449
vmm_ral_sys::get_all_subsys() B-451
vmm_ral_sys::get_subsys_by_name() B-453
vmm_ral_vfield B-454
vmm_ral_vfield::log B-455
vmm_ral_vfield::get_name() B-456
vmm_ral_vfield::get_fullname() B-457
vmm_ral_vfield::get_register() B-458
vmm_ral_vfield::get_lsb_pos_in_register() B-459
vmm_ral_vfield::get_n_bits() B-460

vmm_ral_vfield::get_access() B-461
vmm_ral_vfield::display() B-463
vmm_ral_vfield::psdisplay() B-464
vmm_ral_vfield::read() B-465
vmm_ral_vfield::write() B-467
vmm_ral_vfield::peek() B-469
vmm_ral_vfield::poke() B-471
vmm_ral_vfield::append_callback() B-473
vmm_ral_vfield::prepend_callback() B-474
vmm_ral_vfield::unregister_callback() B-475
vmm_ral_vfield_callbacks B-476
vmm_ral_vfield_callbacks::pre_write() B-477
vmm_ral_vfield_callbacks::post_write() B-479
vmm_ral_vfield_callbacks::pre_read() B-481
vmm_ral_vfield_callbacks::post_read() B-483
vmm_ral_vreg B-485
vmm_ral_vreg::log B-486
vmm_ral_vreg::get_name() B-487
vmm_ral_vreg::get_fullname() B-488
vmm_ral_vreg::get_block() B-489
vmm_ral_vreg::implement() B-490
vmm_ral_vreg::allocate() B-492
vmm_ral_vreg::get_region() B-493
vmm_ral_vreg::release_region() B-494
vmm_ral_vreg::get_memory() B-495
vmm_ral_vreg::get_n_domains() B-496
vmm_ral_vreg::get_domains() B-497
vmm_ral_vreg::get_access() B-498
vmm_ral_vreg::get_rights() B-499
vmm_ral_vreg::get_offset_in_memory() B-500
vmm_ral_vreg::get_address_in_system() B-501
vmm_ral_vreg::get_size() B-502
vmm_ral_vreg::get_n_bytes() B-503
vmm_ral_vreg::get_n_memlocs() B-504
vmm_ral_vreg::get_incr() B-505
vmm_ral_vreg::display() B-506
vmm_ral_vreg::psdisplay() B-507

vmm_ral_vreg::psdisplay_domain() B-508
vmm_ral_vreg::get_fields() B-509
vmm_ral_vreg::get_field_by_name() B-510
vmm_ral_vreg::read() B-511
vmm_ral_vreg::write() B-513
vmm_ral_vreg::peek() B-515
vmm_ral_vreg::poke() B-517
vmm_ral_vreg::append_callback() B-518
vmm_ral_vreg::prepend_callback() B-519
vmm_ral_vreg::unregister_callback() B-520
vmm_ral_vreg_callbacks B-521
vmm_ral_vreg_callbacks::pre_write() B-522
vmm_ral_vreg_callbacks::post_write() B-524
vmm_ral_vreg_callbacks::pre_read() B-526
vmm_ral_vreg_callbacks::post_read() B-528
vmm_ral_tests B-530
vmm_ral_tests::bit_bash() B-531
vmm_ral_tests::hw_reset() B-532
vmm_ral_tests::mem_access() B-533
vmm_ral_tests::mem_walk() B-534
vmm_ral_tests::reg_access() B-535
vmm_ral_tests::shared_access() B-536
vmm_rw B-537
vmm_rw::kind_e B-538
vmm_rw::status_e B-540
vmm_rw_access B-542
vmm_rw_access::kind B-543
vmm_rw_access::addr B-544
vmm_rw_access::data B-545
vmm_rw_access::n_bits B-546
vmm_rw_access::status B-548
vmm_rw_access::new() B-549
vmm_rw_access::psdisplay() B-550
vmm_rw_burst B-551
vmm_rw_burst::n_beats B-552
vmm_rw_burst::incr_addr B-553
vmm_rw_burst::max_addr B-555

vmm_rw_burst::data B-557
vmm_rw_burst::user_data B-559
vmm_rw_xactor B-560
vmm_rw_xactor::exec_chan B-561
vmm_rw_xactor::new() B-563
vmm_rw_xactor::execute_burst() B-564
vmm_rw_xactor::execute_single() B-566
vmm_rw_xactor::notifications_e B-568
vmm_rw_xactor::pre_single() B-570
vmm_rw_xactor::pre_burst() B-571
vmm_rw_xactor::post_single() B-572
vmm_rw_xactor::post_burst() B-573
vmm_rw_xactor_callbacks B-574
vmm_rw_xactor_callbacks::pre_single() B-575
vmm_rw_xactor_callbacks::pre_burst() B-577
vmm_rw_xactor_callbacks::post_single() B-579
vmm_rw_xactor_callbacks::post_burst() B-581

C RAL C API Macros and Functions

Generic API C-2
VMM_RAL_DATA_BUS_WIDTH C-2
VMM_RAL_ADDR_GRANULARITY C-3
vmm_ral_status C-4
DUT-Specific Generated API C-5
ral_addr_of_<sys>_in_<sys>() C-8
ral_addr_of_<blk>_in_<sys>() C-8
ral_addr_of_<reg>_in_<blk>() C-9
ral_addr_of_<mem>_in_<blk>() C-10
ral_read_<reg>_in_<blk>() C-11
ral_write_<reg>_in_<blk>() C-12
ral_read_<fld>_in_<reg>() C-13
ral_write_<fld>_in_<reg>() C-13
ral_read_<fld>_in_<blk>() C-14
ral_write_<fld>_in_<blk>() C-14

D RALF C++ API Class Reference

Class Summary D-17
ralf::vmm_ralf_exception Class Reference D-19
ralf::vmm_ralf_exception::vmm_ralf_exception(...) D-20
ralf::vmm_ralf_exception::getMessage() D-21
ralf::vmm_ralf_error Class Reference D-22
ralf::vmm_ralf_error::vmm_ralf_error(...). D-23
ralf::vmm_ralf_invalid_oprn Class Reference D-24
ralf::vmm_ralf_invalid_oprn::vmm_ralf_invalid_oprn(...) D-25
ralf::vmm_ralf_hdl_path::const_hdl_path_iter Class Reference D-26
ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator++() . . . D-28
ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator*() D-29
ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isBegin() D-30
ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isEnd() D-31
ralf::vmm_ralf_hdl_path Class Reference D-32
ralf::vmm_ralf_hdl_path::createHdlPathIter() D-33
ralf::vmm_ralf_hdl_path::isVhdlPath() D-34
ralf::vmm_ralf_constraint Class Reference D-35
ralf::vmm_ralf_constraint::getName() D-36
ralf::vmm_ralf_constraint::getBody() D-37
ralf::vmm_ralf_fld_typ Class Reference D-38
ralf::vmm_ralf_fld_typ::getName() D-39
ralf::vmm_ralf_fld_typ::getNBits() D-40
ralf::vmm_ralf_fld_typ::getAccess() D-41
ralf::vmm_ralf_fld_typ::getHardReset() D-42
ralf::vmm_ralf_fld_typ::getSoftReset() D-43
ralf::vmm_ralf_fld_typ::getConstraint() D-44
ralf::vmm_ralf_fld_instance Class Reference D-45
ralf::vmm_ralf_fld_instance::getName() D-46
ralf::vmm_ralf_fld_instance::getType() D-47
ralf::vmm_ralf_fld_instance::getHdlPath() D-48
ralf::vmm_ralf_fld_instance::getLSBPosition() D-49
ralf::vmm_ralf_blk_component_instance Class Reference D-50
ralf::vmm_ralf_blk_component_instance::getName() D-51
ralf::vmm_ralf_blk_component_instance::getHdlPath(). D-52
ralf::vmm_ralf_blk_component_instance::getOffset() D-53

ralf::vmm_ralf_blk_component_instance::getAddrSpaceSize(). D-54
ralf::vmm_ralf_reg_typ::const_fld_iter Class Reference D-55
ralf::vmm_ralf_reg_typ::const_fld_iter::operator++() D-56
ralf::vmm_ralf_reg_typ::const_fld_iter::operator*() D-57
ralf::vmm_ralf_reg_typ::const_fld_iter::isBegin() D-58
ralf::vmm_ralf_reg_typ::const_fld_iter::isEnd() D-59
ralf::vmm_ralf_reg_typ Class Reference D-60
ralf::vmm_ralf_reg_typ::getName() D-61
ralf::vmm_ralf_reg_typ::getScopeName() D-62
ralf::vmm_ralf_reg_typ::getNBytes() D-63
ralf::vmm_ralf_reg_typ::createFldIter() D-64
ralf::vmm_ralf_reg_typ::isLeft2Right() D-65
ralf::vmm_ralf_reg_typ::isShared() D-66
ralf::vmm_ralf_reg_typ::getConstraint() D-67
ralf::vmm_ralf_reg_typ::getSharedHdlPath() D-68
ralf::vmm_ralf_reg_instance Class Reference D-69
ralf::vmm_ralf_reg_instance::getType() D-70
ralf::vmm_ralf_reg_instance::isArray() D-71
ralf::vmm_ralf_reg_instance::getArraySize() D-72
ralf::vmm_ralf_reg_instance::getArrayIncr() D-73
ralf::vmm_ralf_reg_instance::isUnMapped() D-74
ralf::vmm_ralf_reg_instance::getOffset() D-75
ralf::vmm_ralf_reg_instance::getSharedAccess() D-76
ralf::vmm_ralf_mem_typ Class Reference D-77
ralf::vmm_ralf_mem_typ::getName() D-78
ralf::vmm_ralf_mem_typ::getScopeName() D-79
ralf::vmm_ralf_mem_typ::getSize() D-80
ralf::vmm_ralf_mem_typ::getNBits() D-81
ralf::vmm_ralf_mem_typ::getAccess() D-82
ralf::vmm_ralf_mem_typ::isShared() D-83
ralf::vmm_ralf_mem_typ::getSharedHdlPath() D-84
ralf::vmm_ralf_mem_instance Class Reference D-85
ralf::vmm_ralf_mem_instance::getType() D-86
ralf::vmm_ralf_mem_instance::isUnMapped() D-87
ralf::vmm_ralf_mem_instance::getOffset() D-88
ralf::vmm_ralf_mem_instance::getSharedAccess() D-89
ralf::vmm_ralf_rfile_typ::const_reg_iter Class Reference D-90

ralf::vmm_ralf_rfile_typ::const_reg_iter::operator++() D-91
ralf::vmm_ralf_rfile_typ::const_reg_iter::operator*() D-92
ralf::vmm_ralf_rfile_typ::const_reg_iter::isBegin() D-93
ralf::vmm_ralf_rfile_typ::const_reg_iter::isEnd() D-94
ralf::vmm_ralf_rfile_typ Class Reference D-95
ralf::vmm_ralf_rfile_typ::getName() D-96
ralf::vmm_ralf_rfile_typ::getScopeName() D-97
ralf::vmm_ralf_rfile_typ::createRegIter() D-98
ralf::vmm_ralf_rfile_typ::getConstraint() D-99
ralf::vmm_ralf_rfile_instance Class Reference D-100
ralf::vmm_ralf_rfile_instance::getType() D-101
ralf::vmm_ralf_rfile_instance::isArray() D-102
ralf::vmm_ralf_rfile_instance::getArraySize() D-103
ralf::vmm_ralf_rfile_instance::getArrayIncr() D-104
ralf::vmm_ralf_domain Class Reference D-105
ralf::vmm_ralf_domain::getNBytes() D-106
ralf::vmm_ralf_domain::getName() D-107
ralf::vmm_ralf_domain::getEndian() D-108
ralf::vmm_ralf_domain::getConstraint() D-109
ralf::vmm_ralf_blk_domain::const_component_iter Class Reference D-110
ralf:: vmm_ralf_blk_domain::const_component_iter:: operator++() D-111
ralf:: vmm_ralf_blk_domain::const_component_iter:: operator*() D-112
ralf::vmm_ralf_blk_domain::const_component_iter:: isBegin() D-113
ralf::vmm_ralf_blk_domain::const_component_iter:: isEnd() D-114
ralf::vmm_ralf_blk_domain Class Reference D-115
ralf::vmm_ralf_blk_domain::createComponentIter() D-116
ralf::vmm_ralf_blk_or_sys_typ Class Reference D-117
ralf::vmm_ralf_blk_or_sys_typ::getName() D-118
ralf::vmm_ralf_blk_or_sys_typ::getScopeName() D-119
ralf::vmm_ralf_blk_typ::const_domain_iter Class Reference D-120
ralf::vmm_ralf_blk_typ::const_domain_iter::operator++() D-121
ralf::vmm_ralf_blk_typ::const_domain_iter::operator*() D-122
ralf::vmm_ralf_blk_typ::const_domain_iter::isBegin() D-123

ralf::vmm_ralf_blk_typ::const_domain_iter::isEnd() D-124
ralf::vmm_ralf_blk_typ Class Reference D-125
ralf::vmm_ralf_blk_typ::createDomainIter() D-126
ralf::vmm_ralf_domain_instance Class Reference D-127
ralf::vmm_ralf_domain_instance::getName() D-128
ralf::vmm_ralf_domain_instance::getHdlPath() D-129
ralf:: vmm_ralf_domain_instance::getInstantiated DomainName() D-130
ralf::vmm_ralf_domain_instance::isArray() D-131
ralf::vmm_ralf_domain_instance::getArraySize() D-132
ralf::vmm_ralf_domain_instance::getArrayIncr() D-133
ralf::vmm_ralf_domain_instance::getOffset() D-134
ralf::vmm_ralf_domain_instance::getAddrSpaceSize() D-135
ralf::vmm_ralf_blk_domain_instance Class Reference D-136
ralf::vmm_ralf_blk_domain_instance::getType() D-137
ralf::vmm_ralf_sys_domain::const_component_iter Class Reference D-138
ralf:: vmm_ralf_sys_domain::const_component_iter:: operator++() D-139
ralf:: vmm_ralf_sys_domain::const_component_iter:: operator*() D-140
ralf:: vmm_ralf_sys_domain::const_component_iter:: isBegin() D-141
ralf::vmm_ralf_sys_domain::const_component_iter:: isEnd() D-142
ralf::vmm_ralf_sys_domain Class Reference D-143
ralf::vmm_ralf_sys_domain::createComponentIter() D-144
ralf::vmm_ralf_sys_typ::const_domain_iter Class Reference D-145
ralf::vmm_ralf_sys_typ::const_domain_iter::operator++() D-146
ralf::vmm_ralf_sys_typ::const_domain_iter::operator*() D-147
ralf::vmm_ralf_sys_typ::const_domain_iter::isBegin() D-148
ralf::vmm_ralf_sys_typ::const_domain_iter::isEnd() D-149
ralf::vmm_ralf_sys_typ Class Reference D-150
ralf::vmm_ralf_sys_typ::createDomainIter() D-151
ralf::vmm_ralf_sys_domain_instance Class Reference D-152
ralf::vmm_ralf_sys_domain_instance::getType() D-153
ralf::vmm_ralf Class Reference D-154
ralf::vmm_ralf::vmm_ralf(...) D-155

ralf::vmm_ralf::generateRTL() D-156
ralf::vmm_ralf::getTopLevelBlockOrSys() D-157

1

Overview of the Register Abstraction Layer

The VMM Register Abstraction Layer (RAL) is a SystemVerilog or OpenVera VMM application package used to automate the creation of a high-level, object-oriented abstraction layer for memory-mapped registers and memories in a design under verification. The abstraction mechanism allows verification environments and tests to be migrated from block to system levels without any modifications. It also allows fields to be moved between physical registers without requiring modifications in the verification environment or tests.

The VMM Register Abstraction Layer can be used in a SystemVerilog VMM-based methodology or in an OpenVera RVM-based methodology. Regardless of the language used, all of the classes implementing the RAL are prefixed with `vmm_`.

The RAL includes predefined testcases that you can use to verify the correct operation of registers and memories in a design under verification. It includes usage assertions to detect incorrect register

and memory accesses. A functional coverage model is included to accurately measure how thoroughly the registers and memories have been exercised.

The RAL supports front-door and back-door access to provide redundant paths to the register and memory implementation, and verify the correctness of the decoding and access paths. The RAL also supports designs with multiple physical interfaces, as well as registers, register files and memories shared across multiple interfaces.

Figure 1-1 Structure of a RAL-based Environment

Figure 1-1 shows the structure of a verification environment based on a RAL. Existing VMM environments can be easily retrofitted to include a RAL.

The sections in this User Guide describe how to use the RAL in the order of which each task should be accomplished. Reading and applying this document in a linear fashion will result in a working RAL-based verification environment with automatically verified registers and memories.

Unsupported Features

Appendix B, "RAL Classes" identifies those methods in the base classes that are not yet implemented.

2

Usage Model

A RAL model comprises fields grouped into registers. Registers and memories are grouped into blocks, and blocks correspond to individually designed and verified components with their own host processor interfaces, address decoding and memory-mapped registers and memories. Even if a memory is physically implemented externally to the block and it is accessed through the block, as part of the block's address space, then the memory is considered as part of the block RAL model.

Blocks may be grouped into systems. Systems and blocks can be grouped into larger systems. A system may contain multiple instances of the same block or subsystem.

The smallest RAL model that can be used is a block. A block may contain one register and no memories or thousands of registers and gigabytes of memory. A system may contain a single block and no subsystems, or several instances of the same block and different subsystems.

For each element in a RAL model—field, register, memory, block or system—there is a class instance that abstracts the read and write operations on that element. Figure 2-1 shows the structure of a design block with two registers, which have two and three fields respectively, and an internal and external memory. Figure 2-2 shows the structure of the corresponding RAL model.

Figure 2-1 Structure of a Design Block

Figure 2-2 Structure of RAL Model for a Design Block

When using RAL, fields, registers and memory locations are not accessed via read and write cycles at specific addresses through a bus-functional model. Rather, they are accessed through read and write methods in their corresponding abstraction class. It is the responsibility of the RAL to turn these abstracted accesses into read and write cycles at the appropriate addresses via the appropriate bus-functional model. A RAL user never needs to worry about the specific address or location of a field, register or memory location, only the name.

For example, the field ADDR in the CONFIG register shown in Figure 2-1 can be accessed through the RAL shown in Figure 2-2 using the `CODEC.CONFIG.ADDR.read()` method. Similarly, location 7 in the BFR memory can be accessed using the `CODEC.BFR.write(7,...)` method. Furthermore, since the location of fields within physical registers is somewhat arbitrary, fields are also accessible independently of their register location. For example, the same `ADDR` field can also be accessed using the `CODEC.ADDR.read()` method. Therefore, if a field is relocated in another register, you do not need to modify a verification environment or testcase.

“Understanding the Generated Model” on page 4 details how the structure of the RAL model is created from a specification and how the names of the specification are used to create the instance names of the various abstraction classes.

Appendix B, "RAL Classes" details the functionality available in each abstraction class.

Chapter 9, "DUT Configuration" details how a RAL model is used to configure a DUT. It also explains how to write block-level register and memory access operations that are portable from a block-level environment to a system-level environment.

Mirroring

The RAL abstraction model maintains a mirror of what it thinks the current value of registers is inside the DUT. The mirrored value is not guaranteed to be correct because the only information the RAL abstraction model has is the read and write access to those registers. If the DUT internally modifies the content of any field or register through its normal operations (for example, by setting a status bit or incrementing an accounting counter), the mirrored value becomes outdated.

The RAL abstraction model takes every opportunity to update its mirrored value. Upon every read operation, whether via a physical interface or back-door access, the mirror for the read register is updated. Upon every write operation, whether via a physical interface or back-door access, the new mirror value for the written register is predicted based on the access modes of the bits in the register. Resetting a RAL abstraction model sets the mirror to the reset value specified in the model.

A mirror is not a scoreboard. It can accurately predict the content of registers that are not updated by the design. However, it cannot determine if an updated value is correct or not. For example, a one-bit field of mode "`vmm_ral::A1`" can return a value of 0 in one read operation and a value of 1 in the subsequent read operation. The mirror notices the change in value and updates itself, but it cannot assume that the raising of that bit is invalid. However, if the same bit goes from 1 to 0 without an intervening write operation or a reset to clear, the mirror flags the updated value as invalid.

You can access mirrored values in zero-time by using the "vmm_ral_field::get()" or "vmm_ral_reg::get()" methods. You can also update the mirror by using the "vmm_ral_field::mirror()" , "vmm_ral_reg::mirror()" or "vmm_ral_block_or_sys::mirror()" methods. Updating the mirror for a field also updates the mirror for all the other fields in the same register. Updating the mirror for a block or system updates the mirror for all fields and registers it contains. Updating a mirror in a large block or system may take a lot of simulation time if physical read cycles are used, whereas updating using back-door access usually takes zero-time.

You can write to mirrored values in zero-time by using the "vmm_ral_field::set()" or "vmm_ral_reg::set()" methods. Once a mirror value has been written to, it no longer reflects the value in the corresponding field or register in the DUT. You can update the DUT to match the mirror values by using the "vmm_ral_reg::update()" or "vmm_ral_block_or_sys::update()" methods. If the new mirrored value matches the old mirrored value, the register is not updated. Updating a block or system with its mirror updates all fields and registers it contains with their corresponding mirror values. Updating a large block or system may take a lot of simulation time if physical write cycles are used, whereas updating using back-door access

usually takes zero-time. Synopsys recommends that you use this update-from-mirror process when configuring the DUT to minimize the number of write operations performed.

Memories Are Not Mirrored

Memories can be quite large. That is why they are usually modelled using a sparse array approach. Only the locations that have been written to are stored, and later read back. Any unused memory location is not modelled. Mirroring a memory would require that the same technique be used.

When verifying the correct operations of a memory, it is necessary to read and write all addresses. This negates the memory-saving characteristics of a sparse-array technique. Both the memory model of the DUT and the memory mirror, would end up being fully populated, and duplicating the same large amount of information.

Unlike bits in fields and registers, the behavior of bits in a memory is very simple: all bits of a memory can either be written to or not. A memory mirror would then be a ROM or RAM memory model—a model that is already being used in the DUT to model the memory being mirrored. The memory mirror can then be replaced by providing back-door access to the memory model.

Therefore, using the "vmm_ral_mem::peek()" or "vmm_ral_mem::poke()" methods provide the exact same functionality as a memory mirror. Additionally, unlike a mirror based on observed read and write operations, using back-door accesses instead of a mirror always returns or sets the actual value of a memory location in the DUT.

If you specify the necessary `hdl_node` attributes in the RALF specification, memories modelled inside a Verilog DUT using single-dimension unpacked arrays of packed bits, have their back-door access automatically generated. You must provide a user-defined back-door mechanism if the memory is modelled using an associative array, discrete registers or uses a third-party memory model (such as the DesignWare memory models). Refer to Chapter 10, "Back-door Access" for additional information.

3

Installing and Using RAL

Every shipment of VCS or Vera includes RAL. Therefore, no further installation is required. If you are using SystemVerilog with an older version of VCS that does not include a RAL installation, or you wish to use a more recent version of the SystemVerilog implementation of RAL, you can use a separate installation. First, you must obtain the VMM open source distribution available at http://vmmcentral.org and install it in some directory identified by the `$VMM_HOME` environment variable.

Minimum Requirements

You must have, at a minimum, the specified versions of the following tools:

- VCS or Pioneer 2005.06-SP12008.03
- Vera 2005.06

- Perl 5.6
- TCL 8.3

Installation Instructions

Perform the following steps to install the VMM Open Source distribution:

1. Create an installation directory.

   This directory must be accessible by all users.

2. Decompress and expand the VMM Open Source package.

   Install the package in the installation directory created in step 1.

3. Define the `VMM_HOME` environment variable.

   For the remainder of this document, the installation directory will be referred to as `${VMM_HOME}`.

4. Add `${VMM_HOME}/shared/bin` to your `PATH` environment variable.

Compiling and Running OpenVera Code With Vera

To make the predefined RAL classes visible in your OpenVera code, put the following directive at the top of the relevant source files:

```
#include "vmm_ral.vrh"
```

Vera will automatically locate the header file and load the appropriate object file found in the Vera distribution.

Compiling and Running OpenVera Code With VCS

To make the predefined RAL classes visible in your OpenVera code, you must specify the following file before any OpenVera source files when compiling using VCS:

```
vcs -ntb -ntb_opts rvm ...\
   ${VCS_HOME}/etc/rvm/vmm_ral.vrp ...
```

Compiling and Running SystemVerilog Code

To make the predefined RAL classes visible in your SystemVerilog code, include the following directive at the top of the relevant source files:

```
`include "vmm_ral.sv"
```

...and specify the following command-line option:

```
% vcs ... -ntb_opts rvm ...
```

This automatically includes the source code for the predefined classes in your simulation found in the VCS installation.

Alternatively, you can simply list the following file on the command line before any file that uses the RAL classes:

```
${VCS_HOME}/etc/rvm/vmm_ral.sv
```

Using a VMM Open Source Distribution

You must indicate to VCSyour simulator where to locate the source files by using the following command-line option and by omitting the `"-ntb_opts rvm"` option:

```
+incdir+${VMM_HOME}/sv
```

Alternatively, you can simply list the following file, along with the include directory, on the command line before any file that uses the RAL classes:

```
% vsc ... +incdir+${VMM_HOME}/sv \
          ${VMM_HOME}/sv/vmm_ral.sv ...
```

4

Register and Memory Specification

The Register Abstraction Layer File (RALF) is used to specify all the registers and memories in the design under verification. It is used to generate the object-oriented register and memory high-level abstraction layer. The first step in a project is to create a RALF description. Appendix A, "RALF Syntax" contains detailed syntax and documentation for the RALF description.

As you add and modify fields, registers, and memories, you can update the RALF description many times during a project. You can then regenerate the abstraction layer multiple times without requiring modifications to the existing environment or tests.

Systems, Blocks, Registers, and Fields

In RAL, a design is a block or a system of blocks. The smallest functional unit that can be verified is a block. Systems are designs composed of blocks. Systems can also be composed of smaller systems of blocks, called subsystems.

There must be at least one block in a RALF description. The top-level construct describing the design under verification can be a `block` or `system` construct. The top-level block is identified when the RAL code is generated, therefore, a single RALF description may contain descriptions of multiple blocks and systems. The following example shows the RALF description of a design block:

Example 4-1 RALF Description of a Design Block

```
block blk_name {
   ...
}
```

Systems are composed of subsystems or blocks. Blocks are composed of registers and memories. There can be no registers or memories directly in a system. If a design has system-wide registers or memories, they should be described in a block named, for example, `system_wide`. The following example shows the RALF description of a system:

Example 4-2 RALF Description of a System

```
system sys_name {
   ...
   block blk_name ...
   system subsys_name ...
}
```

Registers are composed of fields. Fields are concatenated to form a register, with optional unused bits between fields. A register must contain at least one field. The following example shows the RALF description of registers and memories in a block:

Example 4-3 RALF Description of Registers and Memories in a Block

```
block blk_name {
   ...
   register reg_name ...
   register reg_name ...
   ...
   memory mem_name ...
}
```

The field is the basic unit of the RAL. Fields are accessed atomically, independently of their location within a register or other fields. Therefore, fields can be moved within or across registers without having to modify the code that uses them. The following example shows the RALF description of fields in a register:

Example 4-4 RALF Description of Fields in a Register

```
register reg_name {
   ...
   field fld_name ...
   field fld_name ...
}
```

Reusability and Composition

RALF descriptions are intended to describe designs that can be arbitrarily combined and reused to create larger designs. There is no need for a RALF description of a block or subsystem to be aware of the context in which the block or subsystem is going to be used. In RALF descriptions, blocks and subsystems are described as stand-alone designs.

Although a RALF can describe an entire design inline, as in Example 4-5, a description can also instantiate blocks, registers and fields as required. The granularity of the description is arbitrary and you should plan for it to maximize reuse.

Example 4-5 Inlined RALF Description

```
system sys_name {
   ...
   block blk_name {
      ...
      register reg_name {
         ...
         field fld_name {
            ...
         }
      }
      ...
      memory mem_name {
         ...
      }
   }
}
```

RALF descriptions can include other RALF descriptions of smaller designs. Included descriptions can be reused and instantiated to compose the description of a larger design. The following example illustrates how this can be done:

Example 4-6 Hierarchical RALF Description

```
field fld_name {
   ...
}

register reg_name {
   ...
   field fld_name ;
}

memory mem_name {
   ...
```

```
}

block blk_name {
   ...
   register reg_name;
   memory mem_name;
}

system sys_name {
   ...
   block blk_name;
}
```

Naming

The names of fields, registers, memories, blocks, and systems are very important because these names are used to identify their corresponding abstraction class in the RAL abstraction model.

The following naming conventions apply to the names elements within a RALF description:

- Names must not be OV or SV reserved keywords

  These names are used as the name of abstraction classes in the generated OV or SV code. Therefore, they cannot be the same as reserved keywords in OV or SV.

- Field names should be unique within a block

Each block abstraction class contains a class property for each field contained in all of its registers, regardless of the specific register where it is located. If unique, the name of the field class property within the block abstraction class is the name of the field. In this case, fields can be moved within or across physical registers without affecting the verification environment or tests. Regardless of field name uniqueness, the block abstraction class contains another field class property referring to each field using the concatenation of the register and field name. See "Registers" for additional information.

Example 4-7 Field Class Properties in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   register xyz {
      field fld2;
   }
}

Yields:

class ral_block_blk_name;
   ...
   vmm_ral_field fld1, reg_name_fld1;
   vmm_ral_field reg_name_fld2;
   ...
   vmm_ral_field xyz_fld2
endclass
```

- Register names must be unique within a block and should be unique from field names.

  Each block abstraction class contains a class property for each register it contains. The name of the register class property within the block abstraction class is the name of the register and must, therefore, be unique and should be different from field names.

Example 4-8 Register Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
}

Yields:

class ral_block_blk_name;
   ral_reg_blk_name_reg_name reg_name;
   vmm_ral_field             fld1, reg_name_fld1;
   vmm_ral_field             fld2, reg_name_fld2;
endclass
```

- Memory names must be unique within a block and unique from register names and should be unique from field names.

  Each block abstraction class contains a class property for each memory it contains. The name of the memory class property within the block abstraction class is the name of the memory and must, therefore, be unique and different from register names. It should also be different from field names.

Example 4-9 Memory Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   memory mem_name;
}

Yields:

class ral_block_blk_name;
   ral_reg_blk_name_reg_name reg_name;
   vmm_ral_field             fld1, reg_name_fld1;
   vmm_ral_field             fld2, reg_name_fld2;
   ral_mem_blk_name_mem_name mem_name;
```

```
endclass
```

- Block and subsystem names must be unique within a system.

  Each system abstraction class contains a class property for each block and subsystem it contains. The name of the block and subsystem class property within the system abstraction class is the name of the block or subsystem. Therefore, block and subsystem names must be unique.

- Independently defined names of registers, memories, blocks, and systems must be, respectively, globally unique within a RALF description.

  Each independently defined RALF element corresponds to a generated abstraction class in the RALF model (see "Understanding the Generated Model"). The names of these elements are used to generate the name of the corresponding class. Class names must be globally unique in SystemVerilog and OpenVera. Therefore, the names of independently defined registers, memories, blocks, and systems must be globally unique, otherwise they will generate identical abstraction class names.

  This requirement does not apply to elements defined inline within another definition.

Note:

Instantiated fields, registers, memories, blocks and subsystems can be renamed. With all of these naming requirements, it would be very difficult to have reusable RALF descriptions. Descriptions would need to know of their contexts to ensure uniqueness. Nor would it be possible to describe a design that contains multiple instances of the same block. Fortunately, any element of a RALF description can be renamed when instantiated to ensure uniqueness.

Example 4-10 Renaming RALF Elements

```
block blk_name {
   ...
}

system sys_name {
   ...
   block blk_name=blk1;
   block blk_name=blk2;
}
```

Hierarchical Descriptions and Composition

A RAL description can have independently specified registers, memories, blocks, and subsystems. You can instantiate these elements in higher level elements to create complete design descriptions.

When you specify registers, memories, blocks and subsystems, you also independently and explicitly specify their physical width as a number of bytes. Therefore, a block can be composed of registers and memories of smaller or larger width. Similarly, systems can be composed of blocks of smaller or larger width.

If you instantiate an element in a wider element, the value of the narrower element is justified to the least-significant bit and the most significant bits are padded with zero or truncated.

If you instantiate an element in a narrower element, the value of the wider element is split into the minimum number of narrower values.

You can specify splitting as:

- **Big Endian -** The most-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x0012, 0x3456 and 0x7890.
- **Little Endian -** The least-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x7890, 0x3456 and 0x0012.
- **Big FIFO -** All split values are accessed at the same physical address in the narrower address space. The most-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x0012, 0x3456 and 0x7890.
- **Little FIFO -** All split values are accessed at the same physical address in the narrower address space. The least-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x7890, 0x3456 and 0x0012.

Arrays and Register Files

Many designs have identical registers or groups of registers located in consecutive memory locations. These registers could be described by explicitly specifying each register, ignoring the fact that they are identical.

Example 4-11 Explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name=reg_0;
   register reg_name=reg_1;
   ...
   register reg_name=reg_7;
}
```

The repetitive process could be simplified by using the TCL *for-loop* command. Using the for-loop only simplifies the syntactical requirements of the specification. It does not change the RAL model that will be ultimately generated.

Example 4-12 Iterated explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   for {set n 0} {$n < 8} {incr n} {
      register reg_name=reg_$n;
   }
}
```

The problem with explicitly enumerating consecutive registers is that they have unique names. It will not be possible to randomly index or iterate over their RAL model when writing SystemVerilog or OpenVera code that uses these consecutive registers.

Specifying consecutive registers using a register array will result in an array being available to be indexed or iterated on at runtime, not just at specification time. See "Arrays" for more details on the code generation process for arrays.

Example 4-13 Specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name[8];
   register regX[5] {
      ...
   }
}
```

A sequence of register arrays will locate them in consecutive memory locations. For example, the specification in Example 4-13 will result in the following address map: `reg_name[0]`, `reg_name[1]`, ... `reg_name[7]`, `regX[0]`, `regX[1]`, ... `regX[4]`. If sequences of register groups, or interleaved register arrays are required, then you should a register file array. The specification in Example 4-14 will yield the following address map: `reg[0].reg_name`, `reg[0].X`, `reg[1].reg_name`, ... `reg[4].reg_name`, `reg[4].X`.

Example 4-14 Specification of register file arrays

```
register reg_name {
   ...
}
block blk_name {
```

```
    ...
    regfile reg[5] {
       register reg_name;
       register X {
          ...
       }
    }
}
```

Virtual Fields and Virtual Registers

By default, fields and registers are assumed to be implemented in individual, dedicated hardware structures with a constant and permanent physical location such as a set of D flip-flops. In contrast, virtual fields and virtual registers are implemented in memory or RAM. Their physical location and layout is created by an agreement between the hardware and the software, not by their physical implementation.

Virtual fields and registers can be modelled using RAL by creating a logical overlay on a RAL memory model that can then be accessed as if they were real physical fields and registers. The RAL model of the memory itself remains available for directly accessing the raw memory without regard to any virtual structure it may contain.

Virtual fields define continuous bits in one or more memory locations and can span a memory location boundary. Virtual fields are contained in virtual registers. Virtual registers define continuous whole memory locations. They can span multiple memory locations but are always composed of entire memory locations, never fractions of memory locations.

Figure 4-1 Virtual Field and Virtual Register Structure

Virtual registers are always arrays because the usual reason they are virtual is that there are a large number of them and implementing them in a RAM instead of individual flip-flops is most efficient. Arrays of virtual registers are associated with a memory. The association of a virtual register array with a memory can be static (for example, specified in the RALF file) or dynamic (for example, specified at runtime through user code).

Static virtual registers are associated with a specific memory and are located at specific offsets within that memory. The association is specified in the RALF file and is created by the code generator. This association is permanent and cannot be broken at runtime.

Example 4-15 Static virtual register array

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
   virtual register CHANNEL[1024] DMABFRS@0 {
      field {...};
      ...
   }
}
```

Dynamic virtual registers are dynamically associated with a user-specified memory and are located at user-specified offsets within that memory at runtime. The dynamic allocation of virtual register arrays can also be performed randomly by a Memory Allocation

Manager instance. The structure of the virtual registers is specified in the RALF file, but the number of virtual registers in the array and its association with a memory is specified in the SystemVerilog or OpenVera code and must be correctly implemented by the user. Dynamic virtual registers arrays can be relocated or resized at runtime.

Example 4-16 Dynamic virtual register specification

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
   virtual register CHANNEL {
      field {...};
      ...
   }
}
```

Example 4-17 Implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.implement(1024,
                                ral_model.MAC.DMABFRS,
                                0);
```

Example 4-18 Randomly implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.allocate(1024,
                               ral_model.MAC.DMABFRS.mam);
```

Because virtual fields and virtual registers are implemented in memory, their content is not mirrored by the RAL model.

Multiple Physical Interfaces

Some designs may have more than one physical interface, each with accessible registers or memories. Some registers or memories may even be accessible via more than one physical interfaces and be shared.

A physical interface is called a domain. Only blocks and systems can have domains. Domains contain registers and memories. If a block or system has only one physical interface, there is no need to specify a domain for that interface.

For example, the block "bridge" shown in Example 4-19 specifies a block with two physical interfaces and a register accessible from both interfaces at offset 0 in their respective address spaces.

Example 4-19 Specification for a two-domain block

```
register xfer {
   bytes 4;
   field data {
      access rw;
   }
   shared (xfer_reg);
}

block bridge {
   domain apb {
      bytes 4;
      register xfer;
   }
   domain ahb {
      bytes 4;
      register xfer;
   }
}
```

Some physical interfaces may have different transactions used for configuration than the transactions used for normal operations. For example, PCI interfaces have `configuration write` transactions that are different from normal `write` transactions. Configuration transactions are typically used to set a base address and other decoding information required by normal transactions. Because configuration transactions are used separately from normal transactions, and normal transactions cannot occur until the DUT has been suitably configured using configuration transactions, configuration and normal transactions on the same physical interface must be modelled as separate physical interfaces.

Systems with multiple domains can instantiate blocks with a single domain. A domain must be entirely instantiated within a system domain, that is, a block-level or subsystem-level domain cannot be split between two system-level domains. Different block-level or subsystem-level domains can be instantiated in the same system-level domain but in different address offsets.

When instantiating a multiple-domain block or sub-system in a multiple-domain system, the same name and `hdl_path` must be used for all instances. This creates a single instance of the block or subsystem with its various domains instantiated in different domains.

Example 4-20 shows a specification of a multiple-domain instantiation. Notice how the same instance name "br" and HDL path are used in both cases. Example 4-21 shows the corresponding abstraction model of the system. Notice how domains do not create an additional abstraction scope.

Example 4-20 Instantiating a two-domain block in a two-domain system

```
system amba {
   domain apb {
      bytes 4;
```

```
      block bridge.apb=br (amba_bus.bridge);
   }
   domain ahb {
      bytes 4;
      block bridge.ahb=br (amba_bus.bridge);
   }
}
```

Example 4-21 Model of a two-domain block in a two-domain system

```
class ral_block_bridge extends vmm_ral_block;
   ral_reg_xfer xfer;
   ...
endclass

class ral_sys_amba extends vmm_ral_sys;
   ral_block_bridge br;
   ...
endclass
```

5

Code Generation

Once a description of the available registers and memories in a design is available, `ralgen` can automatically generate the RAL abstraction model for these registers and memories. Test cases, firmware, device drivers and DUT configuration code use this model to access the registers and memories through an object-oriented abstraction layer. The predefined tests also use this model to verify the functional correctness of the registers and memories.

Generating a RAL Model

To generate the RAL model, use the following command:

```
% ralgen [options] -t topname -l sv|ov {-I dir}
{filename.ralf}
```

Where:

`-t topname`

The name of the top-level block or system description in the RALF file that entirely describes the design under verification.

`-l sv|ov`

Specifies SystemVerilog (`sv`) or OpenVera (`ov`) as the implementation language for the generated code. Depending on which option you specify, the RAL model for the entire design is generated in either a file named `ral_`*`topname`*`.sv` or `ral_`*`topname`*`.vr` in the current working directory.

`-I dir`

An optional list of directories that `ralgen` searches for sourced Tcl files.

`filename``.ralf`

The name of the files containing the RALF description. Although, the `.ralf` extension is not required, Synopsys recommends you specify it. However, for multiple files, you specify one top-level RALF file, which should include (source) all the other files through the `include` Tcl option. For example, in the top RALF file, you will have, *`source bottom`*`.ralf`.

Options

The following options are available:

`-b`

Generate the back-door access code for those registers and memories where a complete `hdl_path` has been specified.

`-c a`

Generate the "Address Map" functional coverage model. The `-c` option may be specified multiple times.

`-c b`

Generate the "Register Bits" functional coverage model. The `-c` option may be specified multiple times.

`-c f`

Generate the "Field Values" functional coverage model. The `-c` option may be specified multiple times.

`-e`

Generate empty constraint blocks for every abstract class.

`-f <filename>`

Specifies all the ralgen options within a file.

`-no_bkdr_cb`

By using this option, the backdoor pre/post_read/write callback invocation code is not generated. This option can only be used with option `'-b'`.

Understanding the Generated Model

The generated abstraction model is a function of the RALF description used to generate it. Therefore, understanding how the generation process works will help you use the generated model based on the knowledge of the RALF description.

The generated abstraction model is described using a bottom-up approach, in the order in which the classes are generated and then compiled. If you prefer to read a top-down description, simply read the following sections (“Fields” , “Registers” , “Register Files” , “Virtual Registers” , “Memories” , “Blocks” , and “Systems”) in the reverse order.

Fields

No abstraction class is generated for a field definition. Instead, each field is modeled by an instance of the `vmm_ral_field` class (for details, see “vmm_ral_field”).

The instance of that class is stored in a property of the class modeling the register that instantiates it and the block that instantiates the register.

Registers

An abstraction class is generated for each register definition. For each:

- Independently defined register named `regnam`, there is a class named `ral_reg_regnam`

- Register named `regnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_reg_blknam_regnam`
- Register named `regnam` defined inline in the specification of a register file named `filnam` in a block named `blknam`, there is a class named `ral_reg_blknam_filnam_regnam`

In all cases, the register abstraction class is derived from the `vmm_ral_reg` class (for details, see "vmm_ral_reg").

All virtual methods defined in the `vmm_ral_reg` class are overloaded in the register model class. Each virtual method is overloaded to implement register-specific behavior of the register as defined in the RALF description. No new methods are added to the register abstraction class.

As shown in Example 5-1, the register abstraction class contains a class property for each field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields.

Example 5-1 Register Model Class for Register in Example A-6

```
class ral_reg_CTRL extends vmm_ral_reg;
   vmm_ral_field TXE;
   vmm_ral_field RXE;
   vmm_ral_field PAR;
   vmm_ral_field DTR;
   vmm_ral_field CTS;
   ...
endclass: ral_reg_CTRL
```

Instances of this class are found in the block abstraction class for the blocks instantiating this register.

Arrays

If a register contains any field array, the class property for the field array is declared as a fixed sized array in the corresponding register abstraction class.

Example 5-2 Array Specifications and Corresponding Model

```
register r {
      bytes 1;
field f[8] {
bits 1;
  }
}
```

Corresponding abstraction model:

```
class ral_reg_b_r extends vmm_ral_reg;
   rand vmm_ral_field f[8];
   ...
endclass: ral_reg_b_r
```

Register Files

An abstraction class is generated for each register file definition. For each register file named `filnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_regfile_blknam_filnam`. The register abstraction class is not derived from any base class because it is purely a container for the registers instantiated in the register file.

The register file container class contains a class property for each register it contains. See the "Blocks" on page 10 section for a description of those properties.

Example 5-3 Register File Specification and Corresponding Model

```
block dma_ctrl {
   regfile chan {
      register src {
         field addr { ... }
      }
      register dst {
         field addr { ... }
      }
      register count {
         field n_bytes { ... }
      }
      register ctrl {
         field TXE { ... }
         field BSY { ... }
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_dma_ctrl_chan;
   ral_reg_dma_ctrl_chan_src src;
   vmm_ral_field             src_addr;

   ral_reg_dma_ctrl_chan_dst dst;
   vmm_ral_field             dst_addr;

   ral_reg_dma_ctrl_chan_count count;
   vmm_ral_field               n_bytes, count_n_bytes;
   vmm_ral_field               TXE, ctrl_TXE;
   vmm_ral_field               BSY, ctrl_BSY;
   ...
endclass: ral_reg_dma_ctrl_chan
```

Instances (usually arrays of instances) of this class are found in the block abstraction class for the blocks instantiating this register file.

Virtual Registers

An abstraction class is generated for each virtual register array definition. For each independently defined virtual register array named `vregnam`, there is a class named `ral_vreg_vregnam`. For each virtual register array named `vregnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_vreg_blknam_vregnam`. In both cases, the virtual register array abstraction class is derived from the "vmm_ral_vreg" class (for details, see "vmm_ral_reg"). A single abstraction class is used for all virtual registers in the array.

All virtual methods defined in the "vmm_ral_vreg" class are overloaded in the virtual register array abstraction class. Each virtual method is overloaded to implement register-specific behavior of the virtual register array as defined in the RALF description. No new methods are added to the virtual register array abstraction class.

As shown in Example 5-4, the virtual register array abstraction class contains a class property for each virtual field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields, and unlike register arrays, a single instance of the virtual register array abstraction class is used to model the complete virtual register array.

Example 5-4 Virtual Register Abstraction Class

```
block blk1 {
   memory ram0 { ... }

   virtual register dma[256] ram0@0x0000 {
      field len { ... }
      field bfrptr { ... }
      field ok { ... }
   }
```

```
}
```

Corresponding abstraction model:

```
class ral_vreg_blk1_dma extends vmm_ral_vreg;
   vmm_ral_vfield len;
   vmm_ral_vfield bfrptr;
   vmm_ral_vfield ok;
   ...
endclass: ral_vreg_blk1_dma

class ral_block_blk1 extends vmm_ral_block;
   vmm_ral_mem       ram0;
   ral_vreg_blk1_dma dma;
   ...
endclass: ral_block_blk1
```

A single instance (not an array of instance) of this class is found in the block abstraction class for the blocks instantiating a virtual register array.

Memories

An abstraction class is generated for each memory definition. For each independently defined memory named `memnam`, there is a class named `ral_mem_memnam`. For each memory named `memnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_mem_blknam_memnam`.

In both cases, the memory abstraction class is derived from the `vmm_ral_mem` class (for details, see “vmm_ral_mem”).

All virtual methods defined in the `vmm_ral_mem` class are overloaded in the memory abstraction class. Each virtual method is overloaded to implement memory-specific behavior of the memory as defined in the RALF description. No new methods are added to the memory abstraction class.

As shown in Example 5-5, the memory abstraction class contains no additional class properties.

Example 5-5 Memory Abstraction Class for Memory in Example A-11

```
class ral_mem_ROM extends vmm_ral_mem;
   ...
endclass: ral_mem_ROM
```

Instances of this class are found in the block abstraction class for the blocks instantiating this memory.

Blocks

An abstraction class is generated for each block definition. For each independently defined block named `blknam`, there is a class named `ral_block_blknam`. For each block named `blknam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_block_sysnam_blknam`. In both cases, the block abstraction class is derived from the `vmm_ral_block` class (for details, see "vmm_ral_block").

All virtual methods defined in the `vmm_ral_block` class are overloaded in the block abstraction class. Each virtual method is overloaded to implement block-specific behavior of the block as defined in the RALF description. No new methods are added to the block abstraction class.

As shown in Example 5-6 and Example 5-7, the block abstraction class contains a class property for each register and register file it contains. The name of the register or register file property is the name of the register or file. The block abstraction class also contains one or two class properties for each field it contains. The name of each field property is the name of the field (if unique within the

register) and the name of the register concatenated with the name of the field, respectively. There are no properties for unused or reserved fields.

Important:

It is preferable that field names be unique across blocks. Therefore, each field has a property with the same name in the block abstraction class that instantiates them. If you move the field to another physical register, you can use this uniquely-named field property to reduce testbench maintenance. If you use the name that is prefixed with the register name, you must modify testbenches if the field is relocated to another physical register.

Example 5-6 Block Abstraction Class for Block in Example A-12

```
class ral_block_uart extends vmm_ral_block;
   ral_reg_CTRL  CTRL;
   vmm_ral_field TXE, CTRL_TXE;
   vmm_ral_field RXE, CTRL_RXE;
   vmm_ral_field PAR, CTRL_PAR;
   vmm_ral_field DTR, CTRL_DTR;
   vmm_ral_field CTS, CTRL_CTS;

   ral_mem_tx_bfr tx_bfr;
   ...
endclass: ral_block_uart
```

Example 5-7 Block Abstraction Class for Block in Example A-14

```
ral_block_bridge extends vmm_ral_block;
   ral_reg_flags     pci_flags;
   vmm_ral_field     pci_flags_cts;
   vmm_ral_field     pci_flags_dtr;

   ral_reg_data_xfer to_ahb;
   vmm_ral_field     to_ahb_data;

   ral_reg_data_xfer frm_ahb;
   vmm_ral_field     frm_ahb_data;

   ral_reg_flags     ahb_flags;
   vmm_ral_field     ahb_flags_cts;
```

```
   vmm_ral_field     ahb_flags_dtr;

   ral_reg_data_xfer to_pci;
   vmm_ral_field     to_pci_data;

   ral_reg_data_xfer frm_pci;
   vmm_ral_field     frm_pci_data;
   ...
endclass: ral_block_bridge
```

Instances of this class are found in the system model class for the systems instantiating this block.

Arrays

If a block contains a register array or register file array, the class property for the register array or register file array is declared as a fixed sized array to the corresponding register abstraction class or register file container class. Similarly, the field properties for the fields contained in the register array are declared as a fixed sized array of `vmm_ral_field` classes.

Example 5-8 Array Specifications and Corresponding Model

```
block b1 {
   register r1[32] {
      field f1 { ... }
   }
   regfile  rf[16] {
      register r1 {
         field f1 { ... }
      }
      register r2[4] {
         field f1 { ... };
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_b1_rf;
   ral_reg_b1_rf_r1 r1;
```

```
   vmm_ral_field    r1_f1;

   ral_reg_b1_rf_r2 r2[4];
   vmm_ral_field    f2_f1[4];
   ...
endclass: ral_regfile_b1_rf

class ral_block_b1 extends vmm_ral_block;
   ral_reg_b1_r1 r1[32];
   vmm_ral_field f1[32], r1_f1[32];

   ral_regfile_b1_rf rf[16]
   ...
endclass: ral_block_b1
```

Systems

An abstraction class is generated for each system definition. For each independently defined system named `sysnam`, there is a class named `ral_sys_sysnam`. For each subsystem named `subnam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_sys_sysnam_subnam`.

In both cases, the system abstraction class is derived from the `vmm_ral_sys` class (for details, see "vmm_ral_sys").

All virtual methods defined in the `vmm_ral_sys` class are overloaded in the system abstraction class. Each virtual method is overloaded to implement system-specific behavior of the system as defined in the RALF description. No new methods are added to the system abstraction class.

As shown in Example 5-9 and Example 5-10, the system abstraction class contains a class property for each block and subsystem it contains. The name of the block or subsystem property is the name

of the block or system. For blocks with multiple domains, the name of the blocks and subsystems are also available prefixed with the domain name.

Example 5-9 System Abstraction Class for Example A-15

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart0;
   ral_block_uart uart1;
   ...
endclass: ral_sys_SoC
```

Example 5-10 System Abstraction Class for Example A-16

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart0, ahb_uart0;
   ral_block_uart uart1, ahb_uart1,
   ral_block_bridge ahb_br;
   ral_block_bridge pci_br;
   ...
endclass: ral_sys_SoC
```

Arrays

If a system contains a block array or subsystem array, the class property for the block array or subsystem array is declared as a fixed sized array of the corresponding block abstraction class or system abstraction class.

Example 5-11 System Abstraction Class with Block Array

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart[2]
   ...
endclass: ral_sys_SoC
```

Inserting User Defined Code in the Generated Model

In specific scenarios, you might want to make some additions to the generated code of `ralgen`. For example, you might want to add an extra constraint or a couple of additional properties within a register description. This section highlights the support for user code through the new construct, `user_code`.

Using the construct `user_code`, you can insert your own user defined code to the generated RAL model classes as shown below.

```
user_code lang=<language> [(new)] {
      // Any kind/syntax of user code can be added here.
}
```

Here the `<language>` can be SystemVerilog or OpenVera.

The optional `(new)` argument, if specified, would indicate that the `user_code` be appended/inlined to/in the corresponding (RAL Model) class's constructor body. By default, user code will not be inlined in the constructor code. By default, it will be located outside the constructor body (but inside the class), thus, opening up options of adding new data members, functions or task definitions, etc. in the generated RAL model classes.

You can also use the `user_code` construct as an optional property in the definition or specification of a RALF register, virtual register, memory, regfile, block or system as shown in Example 5-12.

Note: You can specify more than one RALF `user_code` in the definition/specification of given a RALF register, virtual register, memory, regfile, block or system. And in this situation, their contents will be concatenated in the generated RAL model class in the same sequence as their corresponding occurrence in the RALF file.

Example 5-12 shows the `user_code` in use.

Example 5-12 RALF Description With User Code'

```
block b {
      bytes 1

      user_code lang=SV {
           // Any block level code can come here.
      }

      register r {
           bytes 1
           field f
           user_code lang=SV (new) {
                // Any register level constructor code can
come here.
           }
      }
}
```

This will generate the following RAL classes with `user_code` inlined appropriately.

```
class ral_reg_b_r extends vmm_ral_reg;
   rand vmm_ral_field f;

   function new(vmm_ral_block parent, string name,
                bit[`VMM_RAL_ADDR_WIDTH-1:0] offset,
                string domain, int cvr,
             bit[1:0] rights = 2'b11, bit unmapped = 0);
      super.new(parent, name, 8, offset, domain, cvr,
rights,
                unmapped, vmm_ral::NO_COVERAGE);
```

```
     this.f = new(this, "f", 1, vmm_ral::RW, 1'h0, 1'hx,
0, 0, cvr);

      ////
      // Begin - User Code
      ////
    // Any register level constructor code can come here.
      ////
      // End - User Code
      ////
   endfunction: new
endclass : ral_reg_b_r


class ral_block_b extends vmm_ral_block;
   rand ral_reg_b_r r;
   rand vmm_ral_field r_f;
   rand vmm_ral_field f;

   ////
   // Begin - User Code
   ////
   // Any block level code can come here.
   ////
   // End - User Code
   ////

   function new(int cover_on = vmm_ral::NO_COVERAGE,
string name = "b",
            vmm_ral_sys parent = null, integer base_addr
= 0);
      super.new(parent, name, "b", 1,
vmm_ral::LITTLE_ENDIAN, base_addr,
                "", cover_on, vmm_ral::NO_COVERAGE);
     this.r = new(this, "r", `VMM_RAL_ADDR_WIDTH'h0, "",
cover_on, 2'b11, 0);
      this.r_f = this.r.f;
      this.f = this.r.f;
      this.Xlock_modelX();
   endfunction : new
endclass : ral_block_b
```

UML Diagram

The following UML diagram shows the relationships of the various classes in a complete RAL model. For detailed information, refer to Appendix B which contains detailed descriptions of each class, in alphabetical order.

Figure 5-1 UML Diagram of Complete RAL Model

6

Physical-layer Transactors

The generated abstraction layer provides a high-level interface mechanism to the registers and memories in a design. Testbenches can thus be written in terms of register and memory access without having to worry about addresses or bus cycles. However, the abstraction layer must eventually turn these accesses into physical read and write cycles to the design under verification.

It is not possible to write a generic abstraction layer on top of a specific physical interface. Thus, the RAL assumes a generic physical interface that can perform read and write cycles at specified addresses and return read and cycle completion status information. The RAL assumes it is connected to a generic transactor `vmm_rw_xactor` (see “vmm_rw_xactor”), extended from `rvm_xactor`. This generic transactor executes generic transactions described by the `vmm_rw_access` transaction descriptor. The

behavior of the generic transactor is predefined in its own extension of the `vmm_xactor::main()` method which invokes two new virtual methods to execute burst or single transactions.

These generic transactions must be translated and executed on the actual physical interface of the design under verification. You accomplish this by instantiating an appropriate bus-functional model in an extension of the `vmm_rw_xactor` class and translating the generic `vmm_rw_access` transactions to appropriate transactions on the bus-functional model in an extension of the `vmm_rw_xactor::execute_single()` or `vmm_rw_xactor::execute_burst()` virtual methods. It is not necessary to further extend the `vmm_xactor::main()` method, although it can be done if you require more autonomous behavior.

The structure of a translation transactor is shown in Figure 6-1. The UML diagram of a translation transactor is shown in Figure 6-2.

Figure 6-1 Structure of Translation Transactors

Figure 6-2 UML Diagram of Translation Transactor

The `vmm_xactor::start_xactor()`, `vmm_xactor::stop_xactor()`, and `vmm_xactor::reset_xactor()` methods must be overloaded to appropriately start, stop and reset the physical-level BFM instantiated in the generic transactor and used to execute the generic transactions when the generic transactor is started, stopped or reset.

The `execute_single()` task has no default implementation and must be overloaded. The `execute_burst()` task has a default implementation that executes burst transactions as equivalent single transactions. If the physical protocol does not support burst transactions, the `execute_burst()` task need not be overloaded.

You must define a translation transactor for each unique physical protocol on the design. Multiple physical interfaces using the same physical protocol can use multiple instances of the same translation transactor class.

Example 6-1 shows a translation transactor that executes the generic transactions on a AHB bus.

Example 6-1 Executing Generic Transactions on AHB Interface

```
class ahb_rw_xlate extends vmm_rw_xactor;
   ahb_master ahb;
   ahb_tr     randomized_tr;

   function new(string                instance,
                int unsigned          stream_id,
                ...
                vmm_rw_access_channel exec_chan = null);
      super.new("AHB RAL Master", instance, stream_id,
                exec_chan);

      this.ahb = new(...);
   endfunction: new

   virtual task execute_single(vmm_rw_access tr);
      // Translate the generic RW into a AHB RW
      ahb_tr xtr = new;

      xtr.data_id     = tr.data_id;
      xtr.scenario_id = tr.scenario_id;
      xtr.stream_id   = tr.stream_id;

      this.randomized_tr.randomize() with {
         addr == tr.addr << 2;
         if (tr.kind == vmm_rw::WRITE) {
            kind == ahb_tr::WRITE;
            data == tr.data;
         } else kind == ahb_tr::READ;
      };
      $cast(xtr, this.randomized_tr.copy());
      // Execute physical transaction,
      // assuming a blocking completion model.
      this.ahb.in_chan.put(xtr);
      if (tr.kind == vmm_rw::READ) tr.data = xtr.data;
      if (xtr.status == ahb_tr::NO_ERR)
         tr.status = vmm_rw::IS_OK;
      else tr.status = vmm_rw::ERROR;
    endtask: execute_single

   virtual void function start_xactor();
      super.start_xactor();
      this.ahb.start_xactor();
   endfunction: start_xactor
   ...
```

```
endclass: ahb_rw_xlate
```

Concurrently Executing Generic Transactions

By default, RAL assumes all physical transactions to be atomic, i.e. a transaction is completed before the next cycle is started. If the DUT and physical transactor support pipelined read/write accesses, the generic transactions can also be executed concurrently. You must first enable this capability by defining the compile time macro, `VMM_RAL_PIPELINED_ACCESS`.

When the physical transaction is no longer atomic, the RAL transactor cannot rely on the return of the `execute_single()` method to indicate the completion of the read/write access.

When a transaction is dispatched onto the physical interface to complete execution, the `execute_single()` method might return to start another transaction.

While transactions are being executed by the physical interface, you must set the status of the corresponding generic transaction to `vmm_rw::PENDING`.

After the transaction execution has completed on the physical interface, the response and final status can be annotated onto the generic transaction descriptor. It is also required to indicate the `vmm_data::ENDED` notification in the generic transaction after it is completed to indicate to the RAL transactor that the pipelined transaction has completed its execution.

Example 6-2 explains on how the `vmm_rw_xactor::execute_single()` task should be extended/implemented for supporting concurrency.

Example 6-2 Concurrently Executing Generic Transactions on PBUS Interface

```
   virtual task execute_single(vmm_rw_access tr);
      // Translate the generic RW into a BFM specific RW,
in this case say a pbus transaction
      pbus_tr pb = new;
      pb.addr = tr.addr;
      if (tr.kind == vmm_rw::WRITE) begin
         pb.kind = pbus_tr::WRITE;
         pb.data  = tr.data;
      end
      else begin
         pb.kind = pbus_tr::READ;
      end

      // Pipelined Execution Model
      tr.status = vmm_rw::PENDING;

      fork
         begin
            this.exec_chan.put(pb);
            pb.notify.wait_for(vmm_data::ENDED);

            // Send the result back to RAL
            if (tr.kind == vmm_rw::READ) begin
               tr.data = pb.data;
            end
            tr.status = vmm_rw::IS_OK;
            tr.notify.indicate(vmm_data::ENDED);
         end
      join_none
   endtask: execute_single
```

As shown in Example 6-2, before any BFM transaction is executed, the status of the generic RW transaction is set to `vmm_rw::PENDING` to indicate the processing of that transaction is not yet complete, but has started. And then, for supporting concurrency, `execute_single` should immediately return after

forking off a thread which should execute the transaction and wait for that transaction to complete and update the corresponding generic RW transaction accordingly.

Note: `execute_single` does not wait for the forked-off thread to finish.

The forked-off thread executes the BFM transaction and waits for the completion of that transaction (waits for the DUT to mark the status of that BFM transaction to `vmm_data::ENDED`).

When it is done, the generic RW transaction is updated as required from the BFM transaction, and then, the status of the generic RW transaction is set to `vmm_data::ENDED` to indicate the completion of the generic RW transaction.

Resetting Transactors and Disabling Threads

You should avoid resetting transactors involved in executing physical accesses in the middle of an operation. This may cause a mismatch between the mirrored value in the RAL model and the hardware. If an instance of an "vmm_rw_xactor" is reset using the method while an access transaction is executed, the transaction is forcibly completed with a `vmm_rw::RETRY` status.

A similar problem arises when a thread performing RAL operations is disabled or terminated. Information internal to the RAL model may be left in a spurious state. It will be necessary to invoke the "vmm_ral_reg::reset()" method on the register that was being accessed by the disabled or terminated thread.

7

Verification Environment

Testbenches and tests need a verification environment on which to execute. The predefined tests that come with RAL require some specific elements to be present in the verification environment on which they are built.

A RAL-based verification environment integrates a RAL model with the physical-level transactors that perform the read and write operation and with the design under verification.

A RAL-based environment must be derived from the `vmm_ral_env` class, which is itself derived from the `rvm_env` or `vmm_env` class. The `vmm_ral_env` class is a simple extension of its base class and only serves to introduce two predefined elements: a reset task to reset the design under verification repeatedly during simulation, and an instance of the `vmm_ral_access` transactor. For additional information about the `vmm_ral_env` class, see "vmm_ral_env" .

Once the RAL-based verification environment is suitable for executing the predefined RAL tests, you can expand it to be used as the verification environment for all user-defined tests. However, it is important that the design under verification be as idle and inactive as possible after the completion of the `vmm_ral_env::reset_dut()` method. The predefined tests built on top of the verification environment expect that the value of registers will remain unchanged between accesses. If a field value cannot be guaranteed to remain constant between accesses, its access mode should be specified as `other`.

A RAL-based verification environment is VMM compliant.

Top-level Module

The first step in creating a RAL-based verification environment is to create a top-level module instantiating the design under verification. The top-level module also instantiates all signals needed to connect to the pins of the design as well as clock generators.

Example 7-1 Top-level Module

```
module tb_top;

reg clk = 0;
wire rst = 0;
...
design dut(..., clk, rst, ...);

always #5ns clk = ~clk;

endmodule:tb_top
```

SystemVerilog Interfaces

If you are using SystemVerilog, the top-level module also includes all interface instances required by the physical-level transactors. The wires in the interface may be directly connected to the pins of the design under verification or they may be connected to the wires or variables inside the interface instances used by the design.

Example 7-2 SystemVerilog Interface Instances in Top-level Module

```
module tb_top;
...
ahb_if ahb0();

design dut(..., ahb0.hadr, ahb0.hdat, ...);
...
endmodule:tb_top
```

OpenVera Interfaces

If you are using OpenVera, the next step is to specify the `interface` instances and the binding of `interface` signals to virtual port instances. They are used to connect to the design under verification.

The physical-level transactors may come with macros to facilitate the specification of the `interface` and virtual port bindings.

Example 7-3 OpenVera Interface and Virtual Port Bindings

```
interface ahb0 {
   input         clk CLOCK       hdl_node "tb_top.pclk";
   output [31:0] adr PHOLD   #1  hdl_node "tb_top.haddr";
   inout  [31:0] dat PSAMPLE #-1
                     PHOLD   #1  hdl_node "tb_top.hdat"
   ...
}
```

```
bind ahb_ma_port ahb0_ma {
   adr ahb0.adr;
   dat ahb0.dat;
   ...
}
```

Environment Class

The next step is to create the environment class. The environment class must be derived from the `vmm_ral_env` class.

Example 7-4 RAL-Based Environment Class

```
class tb_env extends vmm_ral_env;
   ...
endclass: tb_env
```

The RAL-based environment base class has similar requirements to the `rvm_env` and `vmm_env` base classes. Additionally, there are three requirements for its implementation: RAL Model, Physical-level Transactors and Reset Tasks. The following sections describe these requirements.

RAL Model

An instance of the generated RAL model must exist. The name of the top-level class in the RAL model depends on the RALF description used to generate it.

You must then register the RAL model with the RAL access transactor instantiated by the base class, using the `vmm_rw_access::set_model()` method.

Example 7-5 RAL Model Instance for Block Design

```
class tb_env extends vmm_ral_env;
   ral_block_uart ral_model;
   ...
   function new();
      this.ral_model = new;
      this.ral.set_model(this.ral_model);
   endfunction: new
   ...
endclass: tb_env
```

Example 7-6 RAL Model Instance for System Design

```
class tb_env extends vmm_ral_env;
   ral_sys_SoC ral_model;
   ...
   function new();
      this.ral_model = new;
      this.ral.set_model(this.ral_model);
   endfunction: new
   ...
endclass: tb_env
```

Physical-level Transactors

The physical-level transactors required to translate between the generic read/write transactions and the actual physical interfaces on the design, must also be instantiated. According to the VMM guidelines, they are instantiated in the `build()` method.

An additional requirement of the RAL-based environment is that their instances must be registered with the RAL access transactor instantiated in the base class. If there is more than one domain (for example, physical interface) in the design under verification, the domain name corresponding to the physical interface driven by the transactor instance must be specified.

Example 7-7 and Example 7-8 show physical interface transactors for single-domain and two-domain designs.

Example 7-7 Physical Interface Transactor Instance for Single-domain Design

```
class tb_env extends vmm_ral_env;
   ...
   ral_ahb_master ahb;
   ...
   virtual function void build();
      super.build();

      this.ahb = new(...);
      this.ral.add_xactor(this.ahb);
      ...
   endfunction: build
   ...
endclass: tb_env
```

Example 7-8 Physical Interface Transactor Instance for Two-domain Design

```
class tb_env extends vmm_ral_env;
   ...
   ahb_rw_xlate ahb;
   pci_rw_xlate pci;
   ...
   virtual function void build();
      super.build();

      this.ahb = new(...);
      this.ral.add_xactor(this.ahb, "ahb");
      this.pci = new(...);
      this.ral.add_xactor(this.pci, "pci");
      ...
   endfunction: build
   ...
endclass: tb_env
```

Reset Task

Finally, instead of overloading the `vmm_env::reset_dut()` or `rvm_env::reset_dut_t()` method, the hardware reset sequence is specified by overloading the `vmm_ral_env::hw_reset()` method. The `vmm_ral_env::reset_dut()` method calls this new method to perform a hardware reset during the simulation process. Example 7-7 illustrates a hardware reset.

Example 7-9 Specifying Hardware Reset Sequence

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
      tb_top.rst <= 1;
      repeat (3) @ (posedge tb_top.clk);
      tb_top.rst <= 0;
   endfunction: hw_reset
   ...
endclass: tb_env
```

Implicitly Phased Environment

For creating an implicitly phased environmet, you need to make the following changes in your environment and the test file:

- The compile time option, `+define+VMM_12_UNDERPIN_VMM_RAL` should be used.
- The top-level environment should be extended from `vmm_group`.
- This environment class should have an explicit instance of `vmm_ral_xactor` proxy.
- The `vmm_ral_access` proxy should be constructed in `build_ph()`.

- Test should be extended from `vmm_test`. Thus by calling `vmm_simulation::run_tests()`, the complete test is run executing various phases.

Example 7-10 : Implicitly Phased Environment

```
class tb_env extends vmm_group;
  vmm_ral_access ral;
  .............
  virtual function void build_ph();
    ral = new;
    .............
    this.ral.set_model(ral_model);
    this.ral.add_xactor(ral_xactor);
  endfunction
  ..............
endclass

program test;
.........
class my_test extends vmm_test;
   tb_env env = new;

   function new();
      super.new("RAL-based Verif Env", "RAL test", null);
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.env = new("tb_env", this);
   endfunction
   ..............
endclass

initial begin
  my_test tst = new;
  vmm_simulation::run_tests();

end
endprogram
```

8

Executing Pre-defined Tests

It is now possible to execute any of the predefined tests that come with RAL to verify the proper operation of the registers and memories in the design under verification. We recommend that you start with the simplest test—the hardware reset test—to debug the RAL-based environment, the physical transactors, and the design under verification to a level where they can be taken through more complicated tests.

Some of the predefined tests require that back-door access be available for registers or memories. See Chapter 10, "Back-door Access" for details on providing back-door access.

The predefined tests expect a file to be included before the `program` statement. In OpenVera, that file must be named `ral_env.vrh`. In SystemVerilog, the name of that file may be specified by defining the VMM_RAL_TEST_PRE_INCLUDE symbol. If that symbol is not defined, the file named ral_env.svh is includedthat file must be named `ral_env.svh`. You can use it to include other header or

source files necessary to compile the test without errors according to your compilation strategy. You can also use it to set the timescale for the test. In SystemVerilog, a second file may be included inside the program block. the name of that file is specified by defining the VMM_RAL_TEST_POST_INCLUDE symbol. If that symbol is not defined, no file is included. You can use this second inclusion point to include files that must be outside of the $unit package or $root module, such as the generated RAL model if it contains hierarchical references for backdoor accesses.the default name of the verification environment used by the predefined tests is `tb_env`. If the verification environment has a different name, specify it by defining the `RAL_TB_ENV` macro in the `ral_env.vrh` or `ral_env.svh` file as follows:

```
`define RAL_TB_ENV my_env
```

Note: The implicitly phased pre-defined tests are also available with the VCS/VMM open source installation.

SystemVerilog

To compile and simulate the hardware reset test (named `hw_reset`) using SystemVerilog, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vcs -sverilog -ntb_opts rvm \
      ... $VCS_HOME/etc/rvm/sv/RAL/tests/hw_reset.sv ...
% ./simv
```

To run implicitly phased `hw_reset` test, use,

```
% vcs -sverilog -ntb_opts rvm \
+define+VMM_12_UNDERPIN_VMM_RAL \
... $VCS_HOME/etc/rvm/sv/RAL/tests/implicit_ph/hw_reset.sv
% ./simv
```

Using a VMM Open Source Installation

To compile and simulate the hardware reset test (named `hw_reset`) using a VMM Open Source distribution, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% ... +incdir+$VMM_HOME/sv \
      $VMM_HOME/sv/RAL/tests/hw_reset.sv ...
```

To run implicitly phased `hw_reset` test, use,

```
% ... +incdir+$VMM_HOME/sv \
$VMM_HOME/sv/RAL/tests/implicit_ph/hw_reset.sv ...
```

OpenVera with VCS

To compile and simulate the hardware reset test (named `hw_reset`) using OpenVera under VCS (NTB), use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vcs -ntb -ntb_opts rvm \
      $VCS_HOME/etc/rvm/vmm_ral.vrp \
      ... $VCS_HOME/etc/rvm/ov/RAL/tests/hw_reset.vr ...
% ./simv
```

OpenVera with Vera

To compile and simulate hardware reset test (named `hw_reset`) using OpenVera under Vera, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vera -cmp -vlog -I. -I $VERA_HOME/include ... \
       $VERA_HOME/pub_src/RAL/tests/hw_reset.vr .
% vcs -vera +vera_load=hw_reset.vro \
      +vera_mload=ral_env.vrl \
      ... vera_shell.v ...
% ./simv
```

Predefined Tests

The following predefined tests are included in RAL. You can augment or modify them to better verify your design.

`gen_html (SV only)`

Generates (crude) HTML documentation of the RAL model.

`hw_reset`

Applies hardware reset to the design and read all registers in the design. Verifies that the value read for each register corresponds to the specified reset value.

`bit_bash`

Verifies that all bits operate as specified, assuming that the DUT is completely idle. Does not test bits of mode *`vmm_ral::OTHER`* and *`vmm_ral::USERn`*.

`reg_access (SV only)`

Exercises all registers with a back-door access available using the following process:

- Skip the register if it contains unpredictable fields, such as `OTHER`
- Write `~reset` using frontdoor
- Check register content using backdoor
- Write reset value using backdoor
- Check register content using frontdoor

`mem_walk`

Walks through all addresses in `vmm_ral::RW` memories using the following process for address `k`:

- Write `~k` at address `k`
- If k>0, read address `k-1` and expect `~(k-1)`
- if k>0, write `k-1` at address `k-1`
- if last address, read address `k` and expect `~k`

`mem_access`

Walks through all addresses in memories with a back-door access available using the following process for address `k`:

- Write random value `v` at address `k` through frontdoor
- If memory is RW, expect `v` through back-door read
- Write `~v` through back-door write

- Read and expect `~k` through front-door read

`shared_access`

Exercises all shared registers and memories using the following process for each domain (requires at least one domain with READ capability or back-door access).

- Write a random value using the domain
- Check the content using all other domains, checking access rights

9

DUT Configuration

Once the correct operation of the registers and memories inside the DUT have been confirmed using the predefined tests, you must implement functional tests to verify the actual functionality of the design. This usually requires that you configure the DUT in some way.

The `vmm_ral_env` verification environment is a VMM-compliant environment, therefore, the configuration of the DUT is implemented with an extension of the `vmm_ral_env::cfg_dut()` method, as shown in Example 9-1.

Example 9-1 Extension of the cfg_dut() Method in the Environment

```
class tb_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      ...
   endtask: cfg_dut
   ...
```

```
endclass: tb_env
```

The verification environment configures the DUT by writing to configuration fields by using the RAL model. All fields, registers, and memories are directly accessible through the RAL abstraction model by using the appropriate hierarchical reference to the field, register, or memory abstraction model. See "Understanding the Generated Model" for a description of how the structure of the RALF specification is used to generate the structure of the RAL abstraction model. For example, a UART with a configuration register as shown in Figure 9-1, could be configured through its RAL model as shown in Example 9-2.

Figure 9-1 UART Configuration Register

CONFIG	Unused	7/8	Unused	PAR	Unused
	15 12	11		3 2	1 0

Example 9-2 Configuration Through the RAL Model

```
virtual task cfg_dut();
   vmm_rw::status_e status;
   super.cfg_dut();
   case (this.cfg.parity)
      NONE: this.ral_model.PAR.write(status, 2'b00);
      ODD : this.ral_model.PAR.write(status, 2'b01);
      EVEN: this.ral_model.PAR.write(status, 2'b10);
   endcase
   this.ral_model.DATA8.write(status,
                              this.cfg.use_8_bits);
endtask: cfg_dut
```

You can minimize the number of write cycles by using the mirror update capability of RAL. In the implementation of the DUT configuration process shown in Example 9-3, a single physical write cycle is performed instead of two, because both fields are located in

the same register. Should the selected configuration correspond to the default configuration of the DUT, no write cycles are necessary or executed.

Example 9-3 Minimizing Physical Configuration Transactions

```
virtual task cfg_dut();
   vmm_rw::status_e status;
   super.cfg_dut();
   case (this.cfg.parity)
      NONE: this.ral_model.PAR.set(2'b00);
      ODD : this.ral_model.PAR.set(2'b01);
      EVEN: this.ral_model.PAR.set(2'b10);
   endcase
   this.ral_model.DATA8.set(this.cfg.use_8_bits);
   this.ral_model.update(status);
endtask: cfg_dut
```

If it is necessary to reuse the block configuration procedure between the block-level environment and the system-level environment, you should implement the bl ock configuration procedure in a task, as shown in Example 9-4. The block configuration task, takes as an argument, the block RAL model used to configure the block.

Example 9-4 Block Configuration Task

```
task uart_config(uart_cfg       cfg,
                 ral_block_uart ral_model);
   vmm_rw::status_e status;
   case (cfg.parity)
      NONE: ral_model.PAR.set(2'b00);
      ODD : ral_model.PAR.set(2'b01);
      EVEN: ral_model.PAR.set(2'b10);
   endcase
   ral_model.DATA8.set(cfg.use_8_bits);
   ral_model.update(status);
endtask: uart_config

virtual task cfg_dut();
   super.cfg_dut();
   uart_config(this.cfg, this.ral_model);
endtask: cfg_dut
```

Should the UART be present in a system, you can reuse the configuration task in the system environment. As shown in Example 9-5, you can use the same configuration task to configure multiple instances of the block.

Example 9-5 Reusing Block Configuration Task in System Configuration

```
virtual task cfg_dut();
   super.cfg_dut();
   uart_config(this.cfg.uart0, this.ral_model.uart0);
   uart_config(this.cfg.uart1, this.ral_model.uart1);
endtask: cfg_dut
```

10

Back-door Access

A back-door access to registers and memory locations is an important tool for efficiently verifying their correct operation.

A back-door access can uncover bugs that may be hidden because write and read cycles are performed using the same access path. For example, if the wrong memory is accessed or the data bits are reversed, whatever bug is introduced on the way in (during the write cycle) will be undone on the way out (during the read cycle).

A backdoor improves the efficiency of verifying registers and memories because it can access registers and memory locations with little or no simulation time. Later, once the proper operation of the physical interface has been demonstrated, you can use back-door access to completely eliminate the simulation time required to configure the DUT, which can sometimes be a lengthy process.

A back-door access operates by directly accessing the simulation constructs that implement the register or memory model through a hierarchical name within the design hierarchy. The main challenges of implementing a back-door access are the identification and maintenance of that hierarchical path and the nature of the simulation constructs used to implement the register or memory model.

Back-door access is limited only by the capabilities of the underlying language and simulation environment.

Back-door Read/Write vs. Peek/Poke

You can perform back-door access to registers and memory by using either the following read/write methods:

- “vmm_ral_field::read()”
- “vmm_ral_field::write()”
- “vmm_ral_mem::read()”
- “vmm_ral_mem::write()”
- “vmm_ral_reg::read()”
- “vmm_ral_reg::write()”

...or the following peek/poke methods:

- “vmm_ral_field::peek()”
- “vmm_ral_field::poke()”
- “vmm_ral_mem::peek()”
- “vmm_ral_mem::poke()”

- “vmm_ral_reg::peek()”
- “vmm_ral_reg::poke()”

The `peek()` methods return the actual value read using the backdoor without modifying the content of the register or memory. Should the register content be modified upon a normal read operation, such as a `clear-on-read` field, it will not be modified. Therefore, reading using `peek()` methods may yield different results than reading through `read()` methods.

The `poke()` methods deposit the specified value directly in the register or memory. Should the register contain non-writable bits or bits that do not reflect the exact value written, such as a `read-update` or `write-1-to-clear` fields, they will contain a different value than if the same value had been written through normal means. All field values, regardless of their access mode, will be forced to the poked value. Therefore, writing using `poke()` methods may yield different results than writing through the frontdoor.

Note: Some design implementations may not allow values to be poked. For example, a read-only register that is implemented by sampling signal values or constants cannot be poked. In such instances, a `vmm_rw::ERROR` status should be returned.

When using the `read()` methods with a back-door access path, the behavior of the register or memory access mimics the same access performed using a front-door access. For example, reading a register containing a `clear-on-read` field will cause the field value to be cleared by poking zeroes into it.

When using the `write()` method with a back-door access path, the behavior of the register or memory access mimics the same access performed using a front-door access. For example, writing to a read-only field using back-door access will cause the field value to be

maintained by first peeking its current value then poking it back in instead of the specified value (unless the entire register is composed of read-only fields, in which case, the entire write operation is ignored).

Generated Backdoors

Automatically-generated back-door mechanisms are associated with their corresponding register or memory abstraction class when the RAL model containing these registers and memories is instantiated. However, in order to enable the automatic generation of back-door access, it is necessary to specify the hierarchical path to the HDL structures that implement the register or memory. This is accomplished by using the `hdl_path` attributes in "field" , "register" , "regfile" , "memory" , "block" and "system" instantiations of the RALF specification.

The generated backdoor simply concatenates the path elements specified in the individual `hdl_path` attributes to form the complete path to the target register or memory. For example, the RALF file shown in Example 10-1 would yield the path `S1_TOP_PATH.b1_i.dec.r1_reg` to the register `r1`.

Example 10-1 RALF Description with hdl_path Specifications

```
system s1 {
   ...
   block b1 (b1_i) @'h1000 {
      ...
       register r1 dec.r1_reg {
          ...
       }
   }
}
```

For a path to be well-formed, a RALF "regfile" , "block" or "system" must correspond to a design module or entity instance. For example, the (partial) RTL code shown in Example 10-2 represents the structure of the design matching the specification in Example 10-1.

Example 10-2 RTL Structure

```
module b1(...);
   ...
   always @ (posedge clk)
   begin: dec
      reg [7:0] r1_reg;
      if (rst) r1_reg <= 0;
      else if (...) r1_reg <= ...;
   end
   ...
endmodule

module s1(...);
   ...
   b1 b1_i(...);
   ...
endmodule

module tb_top;
   ...
   s1 dut(...);
   ...
endmodule
```

The absolute path to the instance of the DUT that corresponds to the RAL model is specified by defining the *name*`_TOP_PATH` symbol where `name` is the uppercase name of the top-level block or system in the RAL model. Using the structure shown in Example 10-2, the `S1_TOP_PATH` symbol must be defined to `tb_top.dut`, as shown in the following:

```
% vcs ... +define+S1_TOP_PATH=tb_top.dut ... \
      ral_s1.sv ...
```

Arrays

If the RALF specification contains arrays of "system" , "block" , "regfile" , "register" or "field" instances (see "Arrays and Register Files" for more details on arrays of instances), the `hdl_path` attribute must contain a `%d`, `[%d]` or `[%g]` format specifier.

Example 10-3 shows the key differences between them.

Example 10-3 A RALF Description Using Different Types of Array Backdoor

```
system s1 {
   bytes 1
   block b1[2] (b1_i%d) {
      bytes 1
      register r1 (dec.r1_reg) {
         field f
      }
   }
   block b2 (blk2) {
      bytes 1
      register r2[2] (r2_array[%d]) {
         field f
      }
   }
   block b3[2] (b3_gen_array[%g].blk) {
      bytes 1
      register r3 (dec.r3_reg) {
         field f
      }
   }
}
```

%d Format Specifier

You should use the `%d` format specifier when the corresponding backdoor RTL implementation of the RALF array is not really an array, rather a series of similarly named non-array signals, for example, block `b1` array in Example 10-3.

Example 10-4 shows that the generated backdoor path of such an RALF array does not have any array in it.

Example 10-4 Generated Code During the %d Format Specifier Usage

```
class ral_reg_s1_b1_r1_bkdr extends
vmm_ral_reg_backdoor;
   int b1;

   function new(int b1);
      this.b1 = b1;
   endfunction

   virtual task read(output vmm_rw::status_e status,
               output bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                input int data_id, input int scenario_id,
                      input int stream_id);
      case (b1)
         0: data = `S1_TOP_PATH.b1_i0.dec.r1_reg;
         1: data = `S1_TOP_PATH.b1_i1.dec.r1_reg;
      endcase
      status = vmm_rw::IS_OK;
   endtask

   virtual task write(output vmm_rw::status_e status,
                input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                input int data_id, input int scenario_id,
    input int stream_id);
      case (b1)
         0: `S1_TOP_PATH.b1_i0.dec.r1_reg = data;
         1: `S1_TOP_PATH.b1_i1.dec.r1_reg = data;
      endcase
      status = vmm_rw::IS_OK;
```

```
   endtask
endclass
```

An example of the RTL implementation of a RALF array, which is not an array in the backdoor/RTL, rather a series of similarly named non-array signals is as follows:

```
module s1(...);
   ...
   b1 b1_i0(...);
   b1 b1_i1(...);
   ...
endmodule
```

[%d] Format Specifier

You should use the `[%d]` format specifier only when the end signal/variable of the corresponding backdoor RTL path/implementation is actually an array, but not a generated instance array. For example, block `r2` as shown in Example 10-3.

The advantage of having a normal array instead of generated instance array is that you can access an each array element by indexing with a variable as shown in Example 10-5.

Example 10-5 Generated Code During the [%d] Format Specifier Usage

```
class ral_reg_s1_b2_r2_bkdr extends
vmm_ral_reg_backdoor;
   int r2;

   function new(int r2);
      this.r2 = r2;
   endfunction

   virtual task read(output vmm_rw::status_e status,
                     output bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                      input int data_id, input int scenario_id,
```

```
                        input int stream_id);
      data = `S1_TOP_PATH.blk2.r2_array[r2];
      status = vmm_rw::IS_OK;
   endtask

   virtual task write(output vmm_rw::status_e status,
                  input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                  input int data_id, input int scenario_id,
                         input int stream_id);
      `S1_TOP_PATH.blk2.r2_array[r2] = data;
      status = vmm_rw::IS_OK;
   endtask
endclass
```

[%g] Format Specifier

Use the `[%g]` format specifier when the corresponding backdoor RTL implementation of the RALF array is a generated instance array. For example, block `b3` array as shown in Example 10-6.

Note:Because the generated instance array cannot be indexed using any variable, numeric constants are used for indexing them.

Example 10-6 Generated Code During the [%g] Format specifier Usage

```
class ral_reg_s1_b3_r3_bkdr extends
vmm_ral_reg_backdoor;
   int b3;

   function new(int b3);
      this.b3 = b3;
   endfunction

   virtual task read(output vmm_rw::status_e status,
                 output bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                  input int data_id, input int scenario_id,
                        input int stream_id);
      case (b3)
         0: data =
```

```
`S1_TOP_PATH.b3_gen_array[0].blk.dec.r3_reg;
         1: data =
`S1_TOP_PATH.b3_gen_array[1].blk.dec.r3_reg;
      endcase
      status = vmm_rw::IS_OK;
   endtask

   virtual task write(output vmm_rw::status_e status,
                  input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                  input int data_id, input int scenario_id,
    input int stream_id);
      case (b3)
        0: `S1_TOP_PATH.b3_gen_array[0].blk.dec.r3_reg
= data;
        1: `S1_TOP_PATH.b3_gen_array[1].blk.dec.r3_reg
= data;
      endcase
      status = vmm_rw::IS_OK;
   endtask
endclass
```

You should use the `[%g]` format specifier when the backdoor RTL path has an array which is not the end signal/variable of that backdoor RTL path. For example, block `b3` array shown in Example 10-6.

Note:You cannot index an XMR using variable, unless it is an end signal/variable. In this case you use numeric constants to index them.

VHDL

Automatic back-door access generation is supported for VHDL designs, or registers/memories located in a portion of the design where the end signal has been implemented/described using VHDL and a new RALF keyword called, `vhdl_path`. You use `vhdl_path` to specify backdoor RTL paths ending in a VHDL signal or a variable.

Note: Currently the VHDL variables are not supported as there is a dependency on `$hdl_xmr` for the support and `$hdl_xmr` does not support the VHDL variables currently.

Example 10-7 RAL Backdoor Support for VHDL Signals

```
system top {
     bytes 1
block blk vhdl_path=(vhd_top_mod:dut_blk) {
     bytes 1;
register reg vhdl_path=(reg) {
     bytes 1
                    …
            }
}
}
```

Note: You need to consider reading Example 10-7 in bold as RALF syntactic keywords.

In Example 10-7, the new RALF keyword, `vhdl_path` is used to specify the backdoor RTL path of the VHDL signals. Hierarchical concatenation rules of `hdl_paths` will also apply for `vhdl_path`.

For example, in the above-shown RALF file, the final synthesized/concatenated backdoor access path for the register `reg` is `vhd_top_mod:dut_blk:reg`. For a given RALF construct, you can either specify a `vhdl_path` or a Verilog `hdl_path` and not both.

If the RALF description of the top-level block/system has `vhdl_path` specified for it or any of its descendents and `ralgen` command line options, `-b` and `-top_path` are used, then `ralgen` will create a file, `ral_vhdl_bkdrs_<top>.v`. The `ralgen` generated file, `ral_vhdl_bkdrs_<top>.v` will have the definition of a module called, `ral_<top>_vhdl_bkdr_connector`.

For each VHDL signal for which backdoor access path needs to be generated using `$hdl_xmr`, this module will have the corresponding Verilog side `reg` definition (of appropriate size) for use/mapping with `$hdl_xmr` and an initial block which would have the corresponding `$hdl_xmr` connection commands.

For a read/write VHDL implemented register, you should create one Verilog `reg` definition for reading and one for writing. These intermediate Verilog `regs` are named after their corresponding synthesized/concatenated hierarchical/absolute `(v)hdl_path` names only after replacing ':' or '.' with '__' and adding a '__' prefix to it. To indicate the direction of data flow/assignment, a '__ip' or '__op' suffix is also added.

An example of such module for the RALF described above is shown in the following example:

```
module ral_top_vhdl_bkdr_connector;
   reg[7:0] __vhd_top_mod__dut_blk__reg__ip;
   reg[7:0] __vhd_top_mod__dut_blk__reg__op;

   initial begin
      $hdl_xmr("<top_path>.vhd_top_mod.dut_blk.reg",
"__vhd_top_mod__dut_blk__reg__ip");
      $hdl_xmr("__vhd_top_mod__dut_blk__reg__op",
"<top_path>.vhd_top_mod.dut_blk.reg");
   end
endmodule
```

Here, the `<top_path>` is the absolute top-level XMR path of the VHDL design instance which must be provided with the new `ralgen` command line option `-top_path` or `-p`. For generating RAL VHDL backdoor code, you should specify this option apart from `-b ralgen` command line option.

Take an instance of this ralgen generated module and specify the absolute top-level XMR path of this instance in the compile time macro `<top>_VHDL_BKDR_CONNECTOR`. By doing this, the RAL backdoor implementation classes can access the VHDL DUT registers/memories through this connection module instance as shown in the following example:

```
class ral_reg_top_blk_reg_bkdr extends
vmm_ral_reg_backdoor;
   virtual task read(output vmm_rw::status_e status,
                 output bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                  input int data_id, input int scenario_id,
                      input int stream_id);
      :
      data =
`TOP_VHDL_BKDR_CONNECTOR.__vhd_top_mod__dut_blk__reg__ip;
      status = vmm_rw::IS_OK;
   endtask
   virtual task write(output vmm_rw::status_e status,
                   input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                  input int data_id, input int scenario_id,
                       input int stream_id);

`TOP_VHDL_BKDR_CONNECTOR.__vhd_top_mod__dut_blk__reg__op =
data;
      :
:
      status = vmm_rw::IS_OK;
   endtask
endclass
```

Target Structures

The automatically-generated back-door access code must make certain assumptions about the nature of the HDL code used to implement the register and memory being accessed. Although there are almost unlimited ways you can implement a register, there are

only a few styles that are supported by the back-door access generator. It is important that, when implementing registers and memories in RTL code, a suitable coding style be used.

The following guidelines outline the restrictions on RTL structures used to implement registers and memories to enable automatic generation of their back-door access. Some of these restrictions may be removed in the future as the capabilities of the back-door access generator are improved.

If the target structures do not meet the requirements for automatic generation of back-door access, a user-defined back-door access mechanism must be created, as specified in "User-defined Backdoors" on page 29.

Writable Fields and memories must be implemented using "reg".

When performing a back-door write operation, a blocking procedural assignment is used. This requires that the target of the assignment be a *reg*.

Read-only fields may be implemented using wire, parameter or Boolean expression.

Such structures cannot be written to, therefore, only the read back-door access to a read-only field is generated. Attempting a back-door write to a read-only field will result in an error.

Example 10-8 Read-only Field Implemented Using an Expression

```
always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
      ...
      16'h0010: rdat = {fifo_fl, fifo_mt};
```

```
      ...
   endcase
end
```

Example 10-9 RALF Description for Read-only Field

```
register r1 @'h0010{
   bytes 2;
   field mt (fifo_mt) {
      bits   1;
      reset  1;
      access ro;
   }
   field fl (fifo_fl) {
      bits   1;
      reset  0;
      access ro;
   }
}
```

Example 10-10 Alternative RALF Description for Read-only Field

```
register r1 (fifo_fl, fifo_mt) @'h0010{
   bytes 2;
   field mt {
      bits   1;
      reset  1;
   }
   field fl {
      bits   1;
      reset  0;
   }
}
```

A register may implement all of its fields in a single "reg".

A register may be composed of more than one field. All these different fields may be implemented in the same *reg* that implements the overall register. This implies that all bits in the register, up to the most-significant bits of the most-significant field,

are implemented and there are no reserved or unused bits between fields. In that case, no *hdl_path* should be specified in field instantiations in the register specification.

For example, the register specified using the *register* definition shown in Example 10-11, can be implemented using the RTL code shown in Example 10-12. The *reg* named r1_reg is used to implement fields f1 and f2.

Example 10-11 Register with Multiple Fields

```
register r1 (r1_reg) @'h0010{
   bytes 2;
   field f1 {
      bits   4;
      reset  4'hA;
   }
   field f2 {
      bits   8;
      reset  8'h55;
   }
}
```

Example 10-12 Single-reg Implementation of Register with Multiple Fields

```
reg [11:0] r1_reg;
always @ (posedge clk)
begin
   if (rst) r1_reg <= {8'h55, 4'hA};
   else if (wr) case (addr)
      ...
      16'h0010: r1_reg <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = r1_reg;
     ...
```

```
   endcase
end
```

If per-field peek()/poke() operations are required (not yet supported), each field instance should have its respective bit slice specified in its *hdl_path* attribute. For example, the register specified using the *register* definition shown in Example 10-13, can also be implemented using the RTL code shown in Example 10-12.

Example 10-13 Register with Multiple Fields

```
register r1 @'h0010{
   bytes 2;
   field f1 (r1_reg[3:0]) {
      bits   4;
      reset  4'hA;
   }
   field f2 (r1_reg[11:4]) {
      bits   8;
      reset  8'h55;
   }
}
```

A register may implement its fields in separate "reg".

A register may be composed of more than one field. All these different fields may be implemented in different *regs* that each implement one field. The register is the concatenation of these individual *regs*. This implementation allows reserved or unused bits between fields. In that case, the *hdl_path* must be specified in field instantiations in the register specification.

For example, the register specified using the *register* definition shown in Example 10-14, can be implemented using the RTL code shown in Example 10-15. The *regs* named `f1_reg` and `f2_reg` are used to implement fields `f1` and `f2` respectively. Additionally, both Example 10-8 and Example 10-9 show an example of a register implemented using separate constructs for separate read-only fields.

Example 10-14 Register with Multiple Fields

```
register r1 @'h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4'hA;
   }
   field f2 (f2_reg) @8 {
      bits   4;
      reset  4'h5;
   }
}
```

Example 10-15 Multiple-reg Implementation of Register with Multiple Fields

```
reg [3:0] f1_reg, f2_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4'hA;
      f2_reg <= 4'h5};
   end
   else if (wr) case (addr)
      ...
      16'h0010: begin
         f1_reg <= wdat[3:0];
         f2_reg <= wdat[11:8];
      end
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = {f2_reg, 4'h0, f1_reg};
     ...
   endcase
end
```

A field may be implemented using multiple "reg".

Like registers, a field may be implemented as separate *regs*. For example, the register specified using the *register* definition shown in Example 10-16, can be implemented using the RTL code shown in Example 10-17. The *regs* named `f2a_reg` and `f2b_reg` are used to implement field `f2`.

Example 10-16 Field Implemented with Multiple regs

```
register r1 @'h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4'hA;
   }
   field f2 (f2a_reg, f2b_reg) @8 {
      bits   4;
      reset  4'h5;
   }
}
```

Example 10-17 Multiple-reg Implementation of a Fields

```
reg [3:0] f1_reg, f2a_reg, f2b_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4'hA;
      {f2a_reg, f2b_reg} <= 4'h55};
   end
   else if (wr) case (addr)
      ...
      16'h0010: begin
         f1_reg <= wdat[3:0];
         {f2a_reg, f2b_reg} <= wdat[11:4];
      end
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
```

```
   else case (addr)
     ...
     16'h0010: rdat = {f2a_reg, f2b_reg, f1_reg};
     ...
   endcase
end
```

A register may have a mix of read-only and writable fields.

Read-only fields cannot be written to, even with a backdoor. A register containing a mix of read-only and writable fields will skip the read-only fields during a back-door write operation.

A memory must be implemented using a single unpacked array.

A memory is accessed using the offset of the memory as the index of the array storing its content. Two memories cannot be modeled using the same array nor can a memory be implemented using the concatenation of multiple arrays (either bit-wise or address-wise).

For example, the memory specified using the *memory* definition shown in Example 10-18, can be implemented using the RTL code shown in Example 10-19. The *reg* named `m1_reg` is used to implement the entire memory.

Example 10-18 Memory Specification

```
memory m1 (m1_reg) @'h1000{
   size 1k;
   bits 16;
}
```

Example 10-19 Implementation of Memory with Unpacked Array

```
reg [15:0] m1_reg[1024];
always @ (posedge clk)
begin
   if (wr) casex (addr)
      ...
```

```
      16'b0001_00xx_xxxx_xxxx: m1_reg[addr[9:0]] <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else casex (addr)
     ...
     16'b0001_00xx_xxxx_xxxx: rdat = m1_reg[addr[9:0]];
     ...
   endcase
end
```

Note: Automatic generation of back-door access to memories modeled using DesignWare models is not yet supported.

Reserved RALF Keywords in Backdoor Path

`ralgen` will error out if any RALF reserved keyword is found in any RALF backdoor HDL path specification. That means, the following RALF description in Example 10-20 will error out.

Note: There are two RALF keywords, `block` and `register` in the HDL path of register `reg`.

Example 10-20

```
register reg (block.register) {

      ...

}
```

If any of your RALF description has got RALF reserved keywords used in any of its backdoor HDL path specification, then use the following RALF syntax for specifying your RALF backdoor HDL path:

Example 10-21

```
register reg hdl_path = (block.register) {

        …

}
```

The semantics of `hdl_path` usage is functionally/completely equivalent to the original HDL path specification style (used in Example 10-20), except the fact that RALF reserved keywords checking will be disabled when the `hdl_path` syntax is used for specifying backdoor HDL path.

Support for Separate Compile

To compile cross module references (XMRs) with VCS separate compile (VCS 2009.12 or later versions), you need an XMR configuration file which will have a list of all the XMRs in a pre-defined format. In case of ralgen generated backdoor code, this configuration file should have a list of all XMRs implementing RAL backdoor signals.

You use the `-sep_cmp` and `-top_xmr_path <top_xmr_path>` `ralgen` command line options to generate the XMR configuration file.

User Interface

Use the `ralgen` command line option, `-sep_cmp`, to invoke the generation of the XMR configuration file when using the `-b` option for the generation of RAL backdoor access code. The `-sep_cmp` option will need the command line option, `-top_xmr_path <top_xmr_path>`.

When using the `-top_xmr_path` option, you need to specify the same top-level XMR path which defines the compile time macro `<top>_TOP_PATH` when compiling the generated RAL model classes. RAL top path/xmr information is required by `ralgen` because, RALF does not capture the information which is required by `ralgen` for building up the absolute top-level XMR path.

You can also use the `ralgen` command line option, `-sep_cmp` and `-top_xmr_path <top_xmr_path>` with other `ralgen` command line options without changing the functionality or meaning of the options.

The name of the generated XMR configuration file is, `ral_<top>.xmr` and will be located in your current working directory.

Example 10-22 Usage example

```
vgamddual100> cat test.ralf
block b {
   bytes 1
   register r1 (reg1) {
      field f
   }
   register r2 (regf.r2) {
      field f
   }
}
vgamddual100> ralgen -q -l sv -t b -b -sep_cmp -
top_xmr_path blk_top test.ralf
Synopsys VMM Register Abstraction Layer Code Generator
Copyright (c) 2006-2009 by Synopsys, Inc.
All Rights Reserved.
vgamddual100>
```

Generated XMR configuration file is as follows:

```
vgamddual100> cat ral_b.xmr
xmr {blk_top} {reg1};
```

```
xmr {blk_top.regf} {r2};
vgamddual100>
```

The format of the above shown separate compile XMR configuration file is the format required by VCS 2009.12 and later versions.

For more details on this format and use-model of separate compile XMR configuration files, see VCS separate compile documentation of their respective releases.

RAL Backdoor Callbacks

RAL backdoor callbacks can be implemented by creating extension of the following callback façade classes, depending on whether the callback extension is required for register or memory read/write backdoor and overriding one or more of their callback methods.

Callback Façade Class for Register Backdoor

```
class vmm_ral_reg_backdoor_callbacks;
   virtual task pre_read(input vmm_ral_reg rg,
                         input int         data_id,
                         input int         scenario_id,
                         input int         stream_id);
   virtual task post_read(input vmm_ral_reg rg,
                         inout vmm_rw::status_e status,
                          inout bit [63:0]       data,
                         input int             data_id,
                       input int            scenario_id,
                        input int            stream_id);

   virtual task pre_write(input vmm_ral_reg rg,
                          inout bit [63:0]  data,
                          input int         data_id,
                         input int         scenario_id,
                          input int         stream_id);
   virtual task post_write(input vmm_ral_reg rg,
                         inout vmm_rw::status_e status,
```

```
                                input bit [63:0]       data,
                              input int             data_id,
                            input int            scenario_id,
                             input int            stream_id);

   virtual function bit [63:0] encode(bit [63:0] data);
   virtual function bit [63:0] decode(bit [63:0] data);
endclass
```

Callback Façade Class for Memory Backdoor

```
class vmm_ral_mem_backdoor_callbacks;
   virtual task pre_read(input vmm_ral_mem mem,
                         inout bit [63:0]  offset,
                         input int         data_id,
                         input int         scenario_id,
                         input int         stream_id);
   virtual task post_read(input vmm_ral_mem mem,
                         inout vmm_rw::status_e status,
                         input bit [63:0]       offset,
                          inout bit [63:0]       data,
                         input int             data_id,
                       input int            scenario_id,
                        input int            stream_id);

   virtual task pre_write(input vmm_ral_mem mem,
                          inout bit [63:0]  offset,
                          inout bit [63:0]  data,
                          input int         data_id,
                         input int         scenario_id,
                          input int         stream_id);
   virtual task post_write(input vmm_ral_mem mem,
                         inout vmm_rw::status_e status,
                         input bit [63:0]       offset,
                           input bit [63:0]       data,
                         input int             data_id,
                       input int            scenario_id,
                        input int            stream_id);
```

```
   virtual function bit [63:0] encode(bit [63:0] data);
   virtual function bit [63:0] decode(bit [63:0] data);
endclass
```

After implementing the necessary callback class extensions, you need to register the instances of these callback class extensions in the RAL register/memory backdoor class instance by using any of the following methods:

```
function void vmm_ral_reg_backdoor::append_callback(
   vmm_ral_reg_backdoor_callbacks cb,
   string                         fname  = "",
   int                            lineno = 0);
function void vmm_ral_reg_backdoor::prepend_callback(
   vmm_ral_reg_backdoor_callbacks cb,
   string                         fname  = "",
   int                            lineno = 0);


function void vmm_ral_mem_backdoor::append_callback(
   vmm_ral_mem_backdoor_callbacks cb,
   string                         fname  = "",
   int                            lineno = 0);
function void vmm_ral_mem_backdoor::prepend_callback(
   vmm_ral_mem_backdoor_callbacks cb,
   string                         fname  = "",
   int                            lineno = 0);
```

If needed, you can also cancel these registrations at later stages by using any of the following methods:

```
function void
vmm_ral_mem_backdoor::unregister_callback(
   vmm_ral_mem_backdoor_callbacks cb,
   string                         fname  = "",
   int                            lineno = 0);

function void
vmm_ral_reg_backdoor::unregister_callback(
```

```
vmm_ral_reg_backdoor_callbacks cb,
string                         fname  = "",
int                            lineno = 0);
```

After you have registered the necessary callback class instances, the `vmm_ral_reg/mem_backdoor::pre/post_read/write()` methods invokes the corresponding callback methods in the order of the registration of their corresponding callback class instances, except for the `vmm_ral_reg/m_backdoor_callbacks::decode()` method, which is invoked in the reverse order of registration.

All the `vmm_ral_reg/mem_backdoor_callbacks::encode()` methods are invoked in the order of registration after all the registered `vmm_ral_reg/mem_backdoor_callbacks::pre_write()` methods are called.

Similarly, all the `vmm_ral_reg/mem_backdoor_callbacks::decode()` methods are invoked in the reverse order of registration before any of the registered `vmm_ral_reg/mem_backdoor_callbacks::post_read()` methods are called.

A call to the appropriate `vmm_ral_reg/mem_backdoor::pre/post_read/write()` method shall be included in the automatically-generated backdoor access code by ralgen, which, in turn, will invoke all the registered callback methods. For example,

```
class ral_reg_oc_ethernet_MODER_bkdr extends
vmm_ral_reg_backdoor;
   virtual task read(output vmm_rw::status_e status,
                     output bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                            input int data_id,
                            input int scenario_id,
                            input int stream_id);
      super.pre_read(data_id, scenario_id, stream_id);
      data =
```

```
{`OC_ETHERNET_TOP_PATH.ethreg1.MODER_2.DataOut,
         `OC_ETHERNET_TOP_PATH.ethreg1.MODER_1.DataOut,
        `OC_ETHERNET_TOP_PATH.ethreg1.MODER_0.DataOut};
      status = vmm_rw::IS_OK;
     super.post_read(status, data, data_id, scenario_id,
                        stream_id);
   endtask

   virtual task write(output vmm_rw::status_e status,
                input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                       input int data_id,
                       input int scenario_id,
                       input int stream_id);
      super.pre_write(data, data_id, scenario_id,
stream_id);
      {`OC_ETHERNET_TOP_PATH.ethreg1.MODER_2.DataOut,
       `OC_ETHERNET_TOP_PATH.ethreg1.MODER_1.DataOut,
       `OC_ETHERNET_TOP_PATH.ethreg1.MODER_0.DataOut
      } = data;
      status = vmm_rw::IS_OK;
    super.post_write(status, data, data_id, scenario_id,
                         stream_id);
   endtask
endclass
```

Encoding/Decoding of Data read/write Through Backdoor

By default, data read/write through automatically-generated backdoor is returned or applied as is. However, in certain applications, the data might not be stored in the same form as what the host processor ultimately sees. For example, parity bits might be added to registers or ECC bits might be added to the memories or the content of memories might be encrypted.

You can use the following RAL backdoor callback methods, `vmm_ral_reg_backdoor_callbacks::encode()`, `vmm_ral_reg_backdoor_callbacks::decode()`,

`vmm_ral_mem_backdoor_callbacks::encode()` and `vmm_ral_mem_backdoor_callbacks::decode()` to do such alterations to the data before or after it is read/ written through the RAL backdoor.

This allows a sequence of data encoding operation to be executed before the data is physically written and the reverse sequence be applied to the data that was physically read. With this, the encoding and corresponding decoding operation can be implemented in the same callback extension.

User-defined Backdoors

User-defined back-door mechanisms are instantiated and associated with their corresponding register or memory abstraction class in the implementation of the `vmm_ral_env::build()` method.

A user-defined register backdoor is provided through an extension of the "vmm_ral_reg_backdoor" class. A back-door write operation is implemented in the "vmm_ral_reg_backdoor::write()" virtual method whereas a back-door read operation is implemented in the "vmm_ral_reg_backdoor::read()" virtual method. This back-door access is then associated with a specific register through the "vmm_ral_reg::set_backdoor()" method.

A user-defined memory backdoor is provided through an extension of the "vmm_ral_mem_backdoor" class. A back-door write operation is implemented in the "vmm_ral_mem_backdoor::write()" virtual method whereas a back-door read operation is implemented

in the “vmm_ral_mem_backdoor::read()” virtual method. This back-door access is then associated with a specific memory through the “vmm_ral_mem::set_backdoor()” method.

If a memory contains error detection and correction codes (ECC), the memory backdoor must handle the generation of ECC bits on write operations. For more information, see “ECC Backdoor Access” .

Implementing a Register Backdoor in OpenVera with Verilog DUT

The following steps detail and illustrate how to implement a register backdoor when using an OpenVera verification environment on a Verilog DUT. Note that it is a different approach than the one used in the automatically generated backdoor in OpenVera because this approach requires less typing and maintenance.

1. Create a port-less module containing read and write tasks. If a port-less module containing register back-door access tasks already exists, you can use the same module. If this is not the first register backdoor, go directly to step 2.

```
module backdoors;

   task reg_read(input  integer id,
                 output [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
```

```
   endtask

   task reg_write(input integer id,
                  input [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

endmodule
```

2. Add a new choice to both `case` statements using the same integer value. The integer value must be unique and different from the other choices in the `case` statements. That integer value is now the unique integer identifier for that register.

```
task reg_read(...);
begin
   case (id)
      ...
      4: ...
      default: ...
   endcase
end
endtask

task reg_write(...);
begin
   case (id)
      ...
      4: ...
      default: ...
   endcase
end
endtask
```

3. Identify the absolute hierarchical access path to the register, starting with the topmost module. Identify the simulation constructs used to implement the register. The path and simulation constructs are subject to change should the structure of the DUT change. The path and constructs may also change between an RTL model and a gate-level model.

   For illustration purposes, the register is implemented using two `regs` named `field1` and `field2`, with an unused bit between them.

4. In the new choice in the read task, implement the necessary statements to read the current value of the register and return it in the `data` argument.

```
task reg_read(...);
begin
   case (id)
      ...
      4: data = {tb_top.dut...field2, 1'b0,
                 tb_top.dut...field1};
      default: ...
   endcase
end
endtask
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the register to the value specified in the `data` argument.

```
task reg_write(...);
begin
   case (id)
      ...
      4: begin
            reg unused;
            {tb_top.dut...field2, unused,
             tb_top.dut...field1} = data;
         end
      default: ...
   endcase
```

```
end
endtask
```

6. In the file that defines the verification environment class (extended from `vmm_ral_env`), define the register back-door read and register back-door write tasks as available external tasks. If this definition already exists, because this is not the first register backdoor to be implemented, go to step 8.

```
hdl_task reg_bkdr_rd(integer        id,
                     var bit [63:0] data)
   "backdoors.reg_read";

hdl_task reg_bkdr_wr(integer    id,
                     bit [63:0] data)
   "backdoors.reg_write";
```

7. In the same file, create an extension of the "vmm_ral_reg_backdoor" class with an integer identifier assigned at construction time. Use this integer identifier when calling the Verilog tasks in the extensions of the virtual methods.

```
class reg_backdoors extends vmm_ral_reg_backdoor {
   local id;

   task new(integer id)
   {
      this.id = id;
   }

   virtual function vmm_rw::status_e
      write_t(bit [63:0] data,
              integer    data_id,
              integer    scenario_id,
              integer    stream_id)
   {
      reg_bkdr_wr(this.id, data);
      write_t = vmm_rw::IS_OK;
   }

   virtual function vmm_rw::status_e
      read_t(var bit [63:0]  data,
             integer    data_id,
```

```
                integer    scenario_id,
                integer    stream_id)
    {
       mem_bkdr_rd(this.id, data);
       read_t = vmm_rw::IS_OK;
    }
}
```

8. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_backdoor" class with the unique integer identifier used in the `case` statement choices created in step 2. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
task tb_env::build()
{
   super.build();
   ...
   {
      reg_backdoors bkdr = new(4);
      this.ral_model.regname.set_backdoor(bkdr);
   }
}
```

Implementing a Register Backdoor in SystemVerilog

The following steps detail and illustrate how to implement a register backdoor when using a SystemVerilog verification environment and DUT:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_reg_backdoor" class with an integer identifier assigned at construction time. If this class extension exists because this is not the first register backdoor, go directly to step 2.

```
class reg_backdoors extends vmm_ral_reg_backdoor;
   local int id;
```

```
   function new(int id);
      this.id = id;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       data,
                      input  int              data_id,
                     input  int             scenario_id,
                     input  int             stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     output bit [63:0]       data,
                     input  int              data_id,
                    input  int             scenario_id,
                    input  int             stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write
endclass: reg_backdoors
```

2. Add a new choice to both `case` statements using the same integer value. This integer value is now the unique integer identifier for that register.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
   case (id)
```

```
      ...
      4: ...
      default: ...
   endcase
endtask: write

virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   case (id)
      ...
      4: ...
      default: ...
   endcase
endtask: read
```

3. Identify the absolute hierarchical access path to the register, starting with the topmost module. Identify the simulation constructs used to implement the register. The path and simulation constructs are subject to change should the structure of the DUT change. They may also change between an RTL model and a gate-level model.

   For illustration purposes, the register is implemented using two `reg` named `field1` and `field2`, with an unused bit between them.

4. In the new choice in the read virtual task, implement the necessary statements to read the current value of the register and return it in the `data` argument.

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data);
   case (id)
      ...
      4: data = {tb_top.dut...field2, 1'b0,
                 tb_top.dut...field1};
      default: ...
   endcase
endtask: read
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the register to the value specified in the `data` argument.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                  input  int              scenario_id,
                   input  int              stream_id);
   case (id)
      ...
      4: begin
            bit unused;
            {tb_top.dut...field2, unused,
             tb_top.dut...field1} = data;
         end
      default: ...
   endcase
endtask: write
```

6. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_backdoor" class with the unique identifier used in the case statement choice created in step 2. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      reg_backdoors bkdr = new(4);
      this.ral_model.regname.set_backdoor(bkdr);
   end
endfunction: build
```

Implementing a Memory Backdoor in OpenVera with DWMM

The following steps detail and illustrate how to implement a memory backdoor when using an OpenVera verification environment on a DesignWare Memory Model (DWMM):

1. Include the file `${VMM_HOME}/ov/RAL/example/dwmm/vmm_ral_dwmm_backdoor.vr` in the file that implements the verification environment (based on `vmm_ral_env`). If this file has already been included because this is not the first memory backdoor, go directly to step 2.

2. Identify the model ID of the `DWMM` instance.

3. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the `vmm_ral_dwmm_backdoor` class with the `DWMM` instance identifier identified in step 2. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
task tb_env::build()
{
   super.build();
   ...
   {
      vmm_ral_dwmm_backdoor bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   }
}
```

Implementing a Memory Backdoor in OpenVera with Verilog DUT

The following steps detail and illustrate how to implement a memory backdoor when using an OpenVera verification environment on a Verilog DUT:

1. Create a port-less module containing read and write tasks. If a port-less module containing register back-door access tasks already exists, you can use this module. If these tasks exist because this is not the first memory backdoor, go directly to step 2.

```
module backdoors;

   task mem_read(input  integer id,
                 input  [63:0]  offset,
                 output [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

   task mem_write(input integer id,
                  input [63:0]  offset,
                  input [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
```

```
   end
   endtask

endmodule
```

An integer identifier—arbitrarily assigned but *unique*—is used to identify the memory requiring back-door access. It will be necessary to translate from that identifier to the actual hierarchical access path in the `case` statement.

2. Add a new choice to both `case` statements using the same integer value. The integer value must be unique from all of the other integer values in the respective `case` statements. This integer value is now the unique integer identifier for that memory.

```
task mem_read(...);
begin
   case (id)
      ...
      2: ...
      default: ...
   endcase
end
endtask

task mem_write(...);
begin
   case (id)
      ...
      2: ...
      default: ...
   endcase
end
endtask
```

3. Identify the absolute hierarchical access path to the memory, starting with the topmost module. Identify the simulation constructs used to implement the memory. The path and simulation constructs are subject to change should the structure of the DUT changes. They may also change between an RTL model and a gate-level model.

For illustration purposes, the memory is implemented using two 8-bit arrays named `byte0` and `byte1`, with a parity bit. Bit #16 is used as an indication of the parity validity.

4. In the new choice in the read task, implement the necessary statements to read the current value of the memory at the specified offset and return it in the `data` argument.

```
task reg_read(...);
begin
   case (id)
      ...
      2: begin
           reg par;
           data[15:0] = {tb_top.dut...byte1[offset],
                         tb_top.dut...byte0[offset]};
           par = ^data[15:0];
           data[16] = par ^
                      tb_top.dut...parity[offset];
         end
      default: ...
   endcase
end
endtask
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the memory at the specified address to the value specified in the `data` argument.

```
task reg_write(...);
begin
   case (id)
      ...
      2: begin
            reg par = (^data[15:0]) ^ data[16];
            tb_top.dut...parity[offset] = par;
            {tb_top.dut...byte1[offset],
            tb_top.dut...byte0[offset]} = data;
         end
      default: ...
   endcase
end
endtask
```

6. In the file that defines the verification environment class (extended from `vmm_ral_env`), define the memory back-door read and write tasks as available external tasks. If this definition already exists, because this is not the first memory backdoor to be implemented, go to step 8.

```
hdl_task mem_bkdr_rd(integer        id,
                     bit [63:0]     offset,
                     var bit [63:0] data)
   "backdoors.mem_read";

hdl_task mem_bkdr_wr(integer    id,
                     bit [63:0] offset,
                     bit [63:0] data)
   "backdoors.mem_write";
```

7. In the same file, create an extension of the "vmm_ral_mem_backdoor" class with an integer identifier assigned at construction time. Use this integer identifier when calling the Verilog tasks in the extensions of the virtual methods.

```
class mem_backdoors extends vmm_ral_mem_backdoor {
   local id;

   task new(integer id)
   {
      this.id = id;
   }

   virtual function vmm_rw::status_e
      write_t(bit [63:0] offset,
              bit [63:0] data,
              integer    data_id,
              integer    scenario_id,
              integer    stream_id)
   {
      mem_bkdr_wr(this.id, offset, data);
      write_t = vmm_rw::IS_OK;
   }

   virtual function vmm_rw::status_e
      read_t(bit [63:0] offset,
```

```
                var bit [63:0]  data,
                integer    data_id,
                integer    scenario_id,
                integer    stream_id)
    {
       mem_bkdr_rd(this.id, offset, data);
       read_t = vmm_rw::IS_OK;
    }
}
```

8. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_mem_backdoor" class with the unique identifier used in the case statement choice created in step 2. Then, associate that new instance with the corresponding memory abstraction class in the RAL model.

```
task tb_env::build()
{
   super.build();
   ...
   {
      mem_backdoors bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   }
}
```

Implementing a Memory Backdoor in SystemVerilog

The following steps detail and illustrate how to implement a memory backdoor when using a SystemVerilog verification environment and DUT:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_mem_backdoor" class with an integer identifier assigned at construction time. If this class extension exists, because this is not the first memory backdoor, go directly to step 2.

```
class mem_backdoors extends vmm_ral_mem_backdoor;
   local int id;
```

```
   function new(int id);
      this.id = id;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       offset,
                      input  bit [63:0]       data,
                      input  int              data_id,
                    input  int             scenario_id,
                     input  int             stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     input  bit [63:0]       offset,
                     output bit [63:0]       data,
                     input  int              data_id,
                    input  int             scenario_id,
                     input  int             stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write
endclass: mem_backdoors
```

2. Add a new choice to both `case` statements using the same integer value. This integer value is now the unique integer identifier for that memory.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
```

```
                    input  int                data_id,
                    input  int                scenario_id,
                    input  int                stream_id);
   case (id)
      ...
      2: ...
      default: ...
   endcase
endtask: write

virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   case (id)
      ...
      2: ...
      default: ...
   endcase
endtask: read
```

3. Identify the absolute hierarchical access path to the memory, starting with the topmost module. Identify the simulation constructs used to implement the memory. The path and simulation constructs are subject to change should the structure of the DUT change. They may also change between an RTL model and a gate-level model.

   For illustration purposes, the memory is implemented using an associative array.

4. In the new choice in the read virtual task, implement the necessary statements to read the current value of the memory at the specified location and return it in the `data` argument.

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data);
   case (id)
      ...
      2: begin
```

```
            if (tb_top.dut...ram.exists(offset))
               data = tb_top.dut...ram[offset];
            else data = '0;
         end
      default: ...
   endcase
endtask: read
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the memory at the specified location to the value specified in the `data` argument.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
   case (id)
      ...
      2: tb_top.dut...ram[offset] = data;
      default: ...
   endcase
endtask: write
```

6. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_mem_backdoor" class with the unique identifier used in the `case` statement choice created in Step 2. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      mem_backdoors bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   end
endfunction: build
```

11

User-defined Field Access

The mirror included with RAL is not intended to be a scoreboard for the field values. It is a best guess effort at the current value of the fields based on observed read and write access to the DUT. If the value of a field is modified by the DUT, or otherwise updated, it is not possible to constantly update it to reflect field values updated or modified by the design. However, if the content of the field can be predicted based on the write and read operations to it, then the mirror can accurately predict its content and its correct operation can be verified.

The RAL contains several predefined field access modes, as specified in "field". The access modes are used, in combination with observed read and write operations, to determine the expected value of a field. You can use the `OTHER` and `USERn` access modes to specify that the behavior of the field does not fall within any of the predefined access modes.

Although most fields fall within one of the predefined categories, it is possible to design a field that behaves predictably but differently from the predefined access modes. It is possible to implement any user-defined behavior through "vmm_ral_field" callback extensions of the "vmm_ral_field_callbacks::pre_write()" and "vmm_ral_field_callbacks::post_read()" methods.

For example, a field that can only be written once it has been read, can, at least once, be modelled using the `vmm_ral::OTHER` or `vmm_ral::USERx` access mode and the following callback extensions:

```
class write_after_read extends vmm_ral_field_callbacks;
   local bit ok_to_write = 1;
   virtual task pre_write(vmm_ral_field         field,
                          ref bit [63:0]        wdat,
                          ref vmm_ral::path_e   path,
                          ref string            domain);
      if (!ok_to_write) wdat = field.get();
      ok_to_write = 0;
   endtask: pre_write

   virtual task post_read(input vmm_ral_field     field,
                          ref   bit [63:0]        rdat,
                          input vmm_ral::path_e   path,
                          input string            domain,
                         ref   vmm_rw::status_e  status);
      ok_to_write = 1;
   endtask: post_read
endclass: write_after_read
```

You can modify the behavior of any field to the user-specified behavior by simply registering the callback extension with the appropriate field abstraction class instance as follows:

```
begin
   write_after_read cb = new;
   this.ral_model.blk.magic_field.append_callback(cb);
end
```

```
this.ral_model.get_fields(flds);
foreach (flds[i]) begin
   if (flds[i].get_access() == ram_ral::USER0) begin
      write_after_read cb = new;
      flds[i].append_callback(cb);
   end
end
```

Field Usage vs. Field Behavior

The access mode of a field is used to specify the physical behavior of the field so the mirror can track, as best as it can, the value of the field. However, it is questionable whether or not it is suitable or functionally correct to use the field in that fashion.

For example, a configuration field could be designed to be written only once by the software after the design comes out of reset. If the design does not support dynamic reprovisioning, it may not be proper to subsequently modify the value of that configuration field. Whether the field should be specified as write-once depends on the hardware functionality. If the *hardware* does not prevent the subsequent write operation, then the field should be specified as read-write because that would accurately reflect the actual behavior of the field.

If you wish to include usage assertions to specific fields (for example, specifying that a configuration field is never written to more than once, despite the fact that it is physically possible), use a callback extension registered with the field as shown in the following example:

```
class config_once extends vmm_ral_field_callbacks;
   local bit written = 0;
   virtual task pre_write(vmm_ral_field         field,
                          ref bit [63:0]        wdat,
                          ref vmm_ral::path_e   path,
```

```
                         ref string            domain);
      if (this.written) `vmm_error(field.log, "...");
   endtask: pre_write

   virtual task post_read(input vmm_ral_field     field,
                         ref   bit [63:0]        rdat,
                         input vmm_ral::path_e   path,
                         input string            domain,
                        ref   vmm_rw::status_e  status);
      this.written = 1;
   endtask: post_read

   task reset();
      this.written = 0;
   endtask: reset
endclass: config_once
```

These usage assertions should be registers in the `vmm_env::config_dut()` method extension to avoid them from being triggered by the predefined tests. For more information, see “Predefined Tests” .

12

Memories with ECC

By default, every bit in each location in a memory is assumed to be user-defined. However, in certain high-reliability designs, the content of memory locations may be protected by extra error detection and correction codes (ECC). Each memory location contains additional ECC bits that the design creates during write operations. The design checks the ECC bits during read operations and indicates an error if a discrepancy is detected.

The presence and value of the ECC bits are transparent to the outside world. However, it is important that their functionality be verified. Furthermore, if you use backdoor write access, it is important to set the ECC bits properly so that an error is not indicated if the memory location is subsequently read through a physical access.

The number of possible memory protection mechanisms is potentially infinite and only limited by the imagination and requirements of the designers. To support arbitrary memory

protection mechanisms, it is possible to augment the default unprotected access mechanism with any user-defined protection scheme.

ECC Backdoor Access

When performing backdoor write operations, it is necessary to correctly set the ECC bits to avoid ECC errors when a physical interface subsequently reads these memory locations. It may also be useful or necessary to have direct access to the ECC bits because these bits are created and used entirely within the design, and can only be accessed through backdoor access.

The default backdoor data path is 64-bits wide. This width is defined by the size of the `data` argument in all of the `read()` and `write()` methods in RAL, not the hardware being verified. As long as data and ECC bits for each memory location fit in the 64-bit data path, it is possible to access ECC bits through the existing RAL memory backdoor mechanism. If more than 64 bits are necessary to access the ECC bits along with the data, simply define the `VMM_RAL_DATA_WIDTH` symbol to the required width (see "Maximum Data Size" for more details). For example, a 64-bit memory with an 8-bit ECC protection code would require that the `VMM_RAL_DATA_WIDTH` be defined as 72 or greater.

When using a memory backdoor access, simply concatenate the ECC bits to the data bits, as shown in Example 12-1. To avoid having the users compute the ECC bit values before or after each backdoor access, it is better to transfer an indication of the ECC bit correctness rather than the ECC bit value itself. ECC is about correctness, not absolute value. This further allows the encapsulation of the ECC algorithm within the memory model.

Example 12-1 Reading and Writing ECC Bits Through Backdoor Access

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
    // Corrupt ECC bit if it is set in the data to write
    data[35:32] = this.ecc.compute(data[31:0])
                  ^ data[35:32];
   tb_top.dut.mem0[offset] = data;
   status = vmm_rw::IS_OK;
endtask: write

virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   // Report invalid ECC bits
   data = tb_top.dut.mem0[offset];
   data[35:32] = this.ecc.compute(data[31:0])
                 ^ data[35:32];
   status = vmm_rw::IS_OK;
endtask: read
```

ECC Physical Access

It may be useful or necessary to inject errors in the ECC or data bits to verify the functionality of the error detection circuitry in the design.

Due to the fact that these bits are created and used entirely within the design, they are not visible or accessible when performing physical access. However, it is possible to gain visibility over the ECC bits before and after a physical access through extensions of the `vmm_ral_mem` callback methods "vmm_ral_mem_callbacks::post_write()" and "vmm_ral_mem_callbacks::pre_read()" .

After the completion of a physical write operation, it is possible to modify the data or ECC bits that were stored in the memory through a direct access, as shown in Example 12-2. Any subsequent read operation to that memory location indicates an ECC error.

Example 12-2 Injecting ECC Errors

```
virtual task post_write(vmm_ral_mem           mem,
                        bit [63:0]            offset,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain,
                        ref vmm_rw::status_e  status);
   bit [35:0] ecc_wdat;
   ecc_wdat = {this.ecc.compute(wdat), wdat[31:0]};
   if (error_on_write) begin
      ecc_wdat ^= 1'b1 << error_on_bit;
   end
   tb_top.dut.mem0[offset] = ecc_wdat;
endtask: post_write
```

Before a physical read operation starts, it is possible to modify the data or ECC bits that are stored in the memory location that is about to be read through a direct access. The subsequent read operation to that memory location indicates an ECC error. It is a good idea to restore the original content of the memory location after the read operation is complete to avoid further ECC errors when reading that memory location. Otherwise, this error injection mode would be no different than an error injected during the write operation.

```
virtual task pre_read(vmm_ral_mem          mem,
                      ref bit [63:0]       offset,
                      ref vmm_ral::path_e  path,
                      ref string           domain);
   if (error_on_read) begin
      ecc_rdat = tb_top.dut.mem0[offset];
      tb_top.dut.mem0[offset] = ecc_rdat
                                ^ (1'b1 << error_on_bit);
   end
endtask: pre_read
```

13

Non-linear, Non-mapped Access

By default, the entire address space of registers and memories is assumed to be linearly mapped into the address space of the block that instantiates it. Each register or location in a memory corresponds to a unique address in the block.

However, you can use different access mechanisms. For example, you could access a large memory in a limited address space using an indexing mechanism: the desired offset within the memory is written into a register, then the data at that memory offset is read or written by reading or writing another register.

The number of possible access mechanisms is potentially infinite and only limited by the imagination and requirements of the designers. To support arbitrary access mechanisms, it is possible to replace the default linearly mapped access mechanism with any user-defined access mechanism.

Non-mapped register or memory does not consume any address space. Their offset should be specified as `@none` when their specification is instantiated in a "block" or "regfile" .

User-defined Register Access

A user-defined register access is provided by an extension of the "vmm_ral_reg_frontdoor" class. A user-defined write operation is implemented in the "vmm_ral_reg_frontdoor::write()" virtual method whereas a user-defined read operation is implemented in the "vmm_ral_reg_frontdoor::read()" virtual method. This user-defined register access mechanism is then associated with a specific memory by the "vmm_ral_reg::set_frontdoor()" method.

User-defined register access mechanisms are instantiated and associated with their corresponding register abstraction class in the implementation of the `vmm_ral_env::build()` method.

The following steps detail and illustrate how to implement a user-defined register access mechanism, using an indexed access mechanism:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_reg_frontdoor" class. Provide a reference to any element of the RAL model or environment that needs to be used to perform read or write access through the constructor.

```
class indexed_reg extends vmm_ral_reg_frontdoor;
   local vmm_ral_reg offset;
   local vmm_ral_reg data;
   local bit [7:0]   addr;

   function new(vmm_ral_reg offset,
                vmm_ral_reg data
```

```
                bit [7:0]   addr);
      this.offset = offset;
      this.data   = data;
      this.addr   = addr;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       data,
                      input  int              data_id,
                     input  int             scenario_id,
                      input  int             stream_id);
      this.offset.write(status, this.addr,
                      data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.write(status, data,
                      data_id, scenario_id, stream_id);
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     output bit [63:0]       data,
                     input  int              data_id,
                    input  int             scenario_id,
                     input  int             stream_id);
      this.offset.write(status,this. addr,
                      data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.read(status, data,
                     data_id, scenario_id, stream_id);
   endtask: read
endclass: indexed_reg
```

2. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_frontdoor" class. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      indexed_reg idxreg = new(this.ral_model.mem_idx,
                         this.ral_model.mem_data, 'h0F);
      this.ral_model.regname.set_frontdoor(idxreg);
   end
endfunction: build
```

User-defined Memory Access

A user-defined memory access is provided by an extension of the "vmm_ral_mem_backdoor::prepend_callback()" class. A user-defined write operation is implemented in the "vmm_ral_mem_frontdoor::write()" virtual method whereas a user-defined read operation is implemented in the "vmm_ral_mem_frontdoor::read()" virtual method. This user-defined memory access mechanism is then associated with a specific memory by the "vmm_ral_mem::set_frontdoor()" method.

User-defined memory access mechanisms are instantiated and associated with their corresponding memory abstraction class in the implementation of the `vmm_ral_env::build()` method.

The following steps detail and illustrate how to implement a user-defined memory access mechanism, using an indexed access mechanism:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_mem_backdoor::prepend_callback()" class. Provide a reference to any element of the RAL model that needs to be used to perform read or write access through the constructor.

```
class indexed_mem extends vmm_ral_mem_frontdoor;
   local vmm_ral_reg offset;
   local vmm_ral_reg data;

   function new(vmm_ral_reg offset,
                vmm_ral_reg data);
      this.offset = offset;
      this.data   = data;
   endfunction: new
```

```
   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       offset,
                      input  bit [63:0]       data,
                      input  int              data_id,
                     input  int             scenario_id,
                     input  int             stream_id);
      this.offset.write(status, offset,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.write(status, data,
                        data_id, scenario_id, stream_id);
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     input  bit [63:0]       offset,
                     output bit [63:0]       data,
                     input  int              data_id,
                    input  int             scenario_id,
                    input  int             stream_id);
      this.offset.write(status, offset,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.read(status, data,
                     data_id, scenario_id, stream_id);
   endtask: read
endclass: indexed_mem
```

2. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_mem_backdoor::prepend_callback()" class. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      indexed_mem idxram = new(this.ral_model.mem_idx,
                               this.ral_model.mem_data);
      this.ral_model.memname.set_frontdoor(idxram);
   end
endfunction: build
```

14

Functional Coverage Model

Optionally, you can generate a RAL model with one or more predefined functional coverage models to measure how thoroughly the various host-accessible elements are exercised by your functional verification suite.

The default generated RAL model does not contain any functional coverage model. To generate a coverage model, `ralgen` must be invoked with the `-c` option. The argument to the `-c` option determines which coverage model is included in the RAL model:

Use `-c b` to generate the register bits coverage model.

-c a

Generate the address map coverage model.

-c f

Generate the field value coverage model.

Multiple functional coverage models can be generated in the same RAL model by specifying the `-c` option multiple times or specifying multiple arguments to a single `-c` option. For example, the following commands are equivalent:

```
% ralgen -c b -c a ...
% ralgen -c ba ...
```

Even though the generated RAL model may contain one or more functional coverage models, they are not enabled by default. This is necessary in order to reduce the memory footprint of a RAL model, as some functional coverage models can be significant in size, and to improve the runtime performance of simulations as the collection of coverage metrics and the writing of functional coverage databases incurs a significant overhead. Therefore, It is necessary to explicitly enable a functional coverage model when a RAL model is first constructed.

The functional coverage models are enabled through the `cover_on` argument of the "vmm_ral_block::new()" or "vmm_ral_sys::new()" methods as shown in Example 14-1. Once enabled, measurement for all functional coverage models is enabled by default.

Example 14-1 Enabling Functional Coverage Models

```
ral_my_block ral_model = new(vmm_ral::REG_BITS +
vmm_ral::ADDR_MAP);

ral_my_block ral_model = new(vmm_ral::ALL_COVERAGE);
```

If it is necessary to dynamically turn off functional coverage measurement, every block and system RAL abstraction class has a `set_cover()` method that dynamically turns functional coverage measurement on or off, as specified by the argument of the method. If the `set_cover()` method is called on a higher-level abstraction

class using the "vmm_ral_block_or_sys::set_cover()" method, it is automatically invoked for all the lower-level abstraction classes it contains.

The following methods are useful for controlling the functional coverage model available in a RAL abstraction model:

- vmm_ral_block_or_sys::has_cover() ..... page B-164
- vmm_ral_block_or_sys::set_cover() ..... page B-165
- vmm_ral_block_or_sys::is_cover_on() ..... page B-167
- vmm_ral_block::new() ..... page B-183
- vmm_ral_block::has_cover() ..... page 164
- vmm_ral_block::set_cover() ..... page 165
- vmm_ral_block::is_cover_on() ..... page 167
- vmm_ral_sys::new() ..... page B-442
- vmm_ral_sys::has_cover() ..... page 164
- vmm_ral_sys::set_cover() ..... page 165
- vmm_ral_sys::is_cover_on() ..... page 167

Predefined Functional Coverage Models

The following functional coverage models are available to be generated in the RAL model. Different models target a different perspective of the register verification process and should be used when appropriate.

Because functional models can be large in size and significantly impact runtime performance, they should be used carefully, at the right level of design granularity and only when their coverage points are targeted. Once filled to satisfaction, functional coverage models should no longer be generated—although their metrics should be preserved and continued to be reported.

Register Bits

This model is generated using the `-c b` command-line option for every register specified with a "+b" cover attribute. The coverage model is constructed by specifying the `vmm_ral::REG_BITS` symbol.

This model is designed to confirm that every specified bit in a RAL model has been thoroughly exercised and is implemented as specified. This functional model can be quite large and is, therefore, best used at the block level.

This functional coverage model is implemented by instances of `ral_cvr_reg_regname::reg_bits` coverage groups. In a block, there is one coverage group instance per register, for each domain instantiating the register. There is a coverage point for every field defined in the register and a bin to measure whether each individual bit of a field has been read and written through the domain physical interface as a 0 and a 1, respectively. For field arrays, a coverage point will be generated for each and every field in the field array and those coverpoints will be named in the <field_name>_<array_index> format where, <array_index> will range from 0 to field array size - 1.

This model does not measure backdoor accesses. The coverage model does not include unused or reserved bits.

Address Map

This model is generated using the `-c a` command-line option for every register and memory specified with a "+a" cover attribute. The coverage model is constructed by specifing the `vmm_ral::ADDR_MAP` symbol.

This model is designed to confirm that the address map of a design has been thoroughly exercised. It is best used at the top-level.

Address map coverage is implemented at the block level and supports address coverage of registers (including any registers in register files) and memories. Because fields cannot be physically accessed, they are not considered in the address map coverage. Virtual registers, being a logical structure imposed on a memory, are not included in the address map coverage either: it is assumed that if the address map coverage model of the memory containing the virtual registers is covered, the address map coverage model for the virtual registers can be considered covered as well.

The address map functional coverage model is composed of the ral_cvr_block_<block_name>::[<domain_name>_]addr_map coverage groups. For each block, there is one coverage group instance per domain in each block instance. In each coverage group (i.e. domain), there is a coverage point for each register (including each registers in register arrays and register files) and a coverage point for each memory in the block.

A register coverage point contains only one bin named "accessed". The bin is covered whenever the register is accessed using a read or a write operation.

A memory coverage point contains three bins. The first bin, named "first_location_accessed", is covered when the first location in the memory is accessed using a read or a write operation. The second bin, named "last_location_accessed", is covered when the last location in the memory is accessed using a read or a write operation. The third bin, named "other_locations_accessed", is covered when anyone of the remaining locations in the memory is accessed using a read or write operation.

Address map coverage measurement happens automatically during any front door read or write operation. Back-door accesses do not contribute toward the address map functional coverage.

Field Values

This model is generated using the `-c f` command-line option for every register specified with a "+f" cover attribute. The coverage model is constructed by specifing the `vmm_ral::FIELD_VALS` symbol.

This model is designed to confirm that every configuration of a design has been verified. It is best used at the top-level.

Field value coverage model is implemented at the register level and supports value coverage of all fields and cross coverage between fields and other cross coverage points within the same register. Field value coverage is not supported for virtual fields/registers.

The field value functional coverage model is composed of the ral_reg_<reg_name>::field_values coverage groups. There is one coverage group instance per register instance. In each coverage group, there is a coverage point for each field in the register, except for "unused" and "reserved" fields. For field arrays, a coverage point

will be generated for each and every field in the field array, and those coverpoints will be named in the <field_name>_<array_index>_value format where, <array_index> will range from 0 to field array size - 1.

By default, if the size of a field is 4 bits or less, the corresponding coverage point contains a bin for each possible value of that field. If the size of the field is greater than 4 bits, the corresponding coverage point contains three bins: the first bin, named "min", corresponds to the minimum value of that field (or ’0); the second bin, named "max", corresponds to the maximum value of that field (or ’1); and the third bin, named "others" corresponds to all other values of that field. The weight of a coverpoint is equal to the number of bins in that point.

You can sample field value coverage by using the `sample_field_values()` function within the RAL registers.

By using this method, you will be able to sample field values within the RAL register itself, which would sample field coverage for all the fields within the register by calling `field_values.sample()` for the register.

For more information on `sample_field_values()` function, see “vmm_ral_block::sample_field_values()” and “vmm_ral_reg::sample_field_values()” **.**

User-Defined Field Value Coverage Bins

If the default field value bins are not suitable, there are many ways coverage bins can be defined for a coverage corresponding to a field value. In all cases, the weight of the coverage point will be equal to the number of bins.

If symbolic values are defined for a field using the "enum" property, a bin is implicitly defined for each symbolic value. The field specification shown in Example 14-2 will create three bins, named "AA", "BB" and "CC", each corresponding to field values 0, 1 and 15 respectively.

Example 14-2 Defining implicit coverage bins via symbolic field values

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
   }
```

User-defined bins can be explicitly specified using the "coverpoint" attribute. Example 14-3 illustrates how multiple coverage bins and bin arrays can be defined using numerical as well as symbolic field values, sets of values and ranges of values. The semantics of the bin specification is identical to the equivalent bin specification in SystemVerilog, as specified in the section named “*Defining coverage points*” in the 1800-2009 SystemVerilog Language Reference Manual.

Example 14-3 Defining explicit coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
     }
   }
```

User Defined Cross Coverage Specification

A cross coverage point between different field values within the same register can be specified using the "cross" attribute. If a user-defined cross-coverage point is labelled, it is possible to use that cross-coverage point in another cross-coverage point.

Example 14-4 User-defined cross-coverage point

```
register r {
    field f1 {...}
    field f2 {...}
    field f3 {...}

    cross f1 f2 {
        label xyz;
    }
    cross xyz f3;
}
```

RALF cover attribute

By default, all applicable elements in a RAL models are included in the address map and register bits coverage models and all are excluded from the field value coverage model. The "cover" attribute can be used to specify the portions of the RAL model that should be included in or excluded from a coverage model.

All elements in a RAL model can be specified with a "cover" attribute to specify whether it and all of the sub-elements it contains are to be included in or excluded from a particular coverage mode. The address map, register bits and field value coverage models are identified by the letters "a", "b" and "f" respectively. A model element is included in or excluded from a coverage model by prefixing its identifying letter with a "+" or a "-’ respectively. For example, the

attribute "cover +a+b-f" specifies that this element is included in the address map and register bits coverage model but not in the field values coverage model.

The coverage attribute for a RAL element are automatically inherited from the higher-level element. If a coverage model is not specified in a "cover" attribute, the inclusion or exclusion for that model is inherited from the higher level. For example, the attribute "cover +f" specified that this element (and all of its lower-level elements) are to be included in the field value coverage model but it does not say anything about the inclusion or exclusion of this element with respect to the other coverage models.

It is important to note that, unless a system, block, register file or register contains a "cover +f" attribute, no field value coverage model will be generated.

Example 14-5 Inherited cover attributes

```
system top {
     block b {
          cover -a+f              #-a+b+f
          …
          register r1 {
               cover -f           #+a+b-f
          }
          register r2 {
               cover -b           #+a-b+f
          }
     }
     system sub {
          cover +f                #+a+b+f
          …
     }
}
```

If a "cover" attribute is specified outside the "domain" attribute of a multi-domain block or system, it applies to all domains specified in that block or system. A "cover" attribute specified inside a "domain" attribute applies to all registers and memories instantiated in that domain.

User-defined Functional Coverage Model

Additional functional coverage group instances can be added to a RAL abstraction model to implement a user-defined functional coverage model. Sampling of user-defined functional coverage points is best implemented in callback extensions.

The sampling of user-defined functional coverage points should be dynamically controlled by checking the return value of the `is_cover_on()` method of the appropriate RAL abstraction class.

15

Randomizing Field Values

A RAL model can specify constraints on field values. If a field is specified with a `constraint` attribute, its value can be randomized. If a field is specified with no `constraint` attributes, it is a constant field that is never randomized. If you require an unconstrained field that can be randomized, specify the field with an empty `constraint` attribute. For example, fields `f1` and `f2` in Example 15-1 are randomized but field `f3` is not.

Within a field specification, the constraints specify the valid values for the field independently of any other field value. Within a register specification, the constraints specify constraints on field values based on the register where the field is instantiated or other field values within the register. Within a block or system specification, the constraints specify constraints on field values based on the block or system where the field is instantiated or other field values within the block or system.

Example 15-1 Field Constraints

```
field f1 {
   bits 8;
   constraint spec {
      value <= 'h80;
   }
}

register r {
   field f1;
   field f2 {
      bits 8;
      constraint consistency {
         f1.value == f2.value;
      }
   }
   field f3 {
      bits 2;
   }
}
```

Example 15-2 RAL Model for Example 15-1

```
class ral_r1 extends vmm_ral_reg;
   rand vmm_ral_field f1;
   rand vmm_ral_field f2;

   constraint f1_spec {
      f1.value < 'h80;
   }
   constraint consistency {
      f1.value == f2.value;
   }
   constraint user_defined;
}
```

Field constraints are inlined in the register class that instantiates the field to minimize the possibility of randomly selecting inconsistent field values. Constraints declared in a `field` property in the RAL description are not visible in the field abstraction class because they are inlined in the register class that instantiates the field and not in the field itself. If a field descriptor is directly randomized, it is

therefore unconstrained. Therefore, do not directly randomize field descriptors. To randomize the content of fields subject to their constraints, the register, block, or system descriptor must be randomized. Once randomized, the field values can be written or updated into the DUT.

Example 15-3 Improperly Randomizing Fields

```
ral_model.r1.f1.randomize();
```

Example 15-4 Properly Randomizing Fields

```
ral_model.r1.randomize();
```

The content of memories cannot be randomized.

Adding Constraints

When constraining a field value, the class property to be constrained is named `value`. This is *not* the class property that is eventually mirrored or updated and used by the `get()` and `set()` methods. It cannot be used for purposes other than random constraints.

You can add additional constraints by using `randomize() with { }` when randomizing a register, block and system abstraction class.

Example 15-5 Randomizing a Register Descriptor with Additional Constraints

```
ok = ral_model.r1.randomize() with {
   f1.value == 0;
};
```

You may also add additional constraints by defining the undefined `user_defined` external constraint block available in register abstraction classes containing randomized fields, and all block and

system abstraction classes. The undefined external constraint block is included in the generated RAL model only if the `-ext_ud` command-line option has been used when invoking `ralgen`. This undefined external constraint block can then be defined by simply adding the necessary out-of-class constraint definition to the simulated code. No tests need to be modified.

Example 15-6 Adding External Constraints

```
constraint ral_r1::user_defined {
   f1.value == 0;
}
```

Undefined external constraint block declarations are not included by default in the RAL model to avoid the many warning messages that are produced when such a constraint is loaded. It may be possible (but not necessarily advisable) to disable these warning messages. For example, the warning message shown in Example 15-7 can be disabled using the "`+warn=noBCNACMBP`" command-line option with VCS.

Example 15-7 Undefined Constraint Block Warning Message from VCS

```
Warning-[BCNACMBP] Body of constraint in non-abstract class
must be present
        Constraint 'user_defined' has no body
```

Example 15-8 Undefined Constraint Block Warning Message from Vera

```
Vera Warning: Method "user_defined()" is declared in class
"ral_r1", but not defined
```

You cannot add constraints by extending a register, block or system abstraction class. It is important that you do not replace references to abstraction class instances in the RAL model by instances of extended classes because internal references are maintained in addition to the occurrences explicitly defined in the RAL model.

Relaxing Constraints

You can relax constraints specified in a RAL definition to inject errors by turning the corresponding constraint block OFF. The name of the constraint blocks are the same as the name of the constraints specified in the RAL description and are found in the corresponding block, system or register descriptor, except for field-specific constraints. Field-specific constraints are inlined in the register descriptor where the field is instantiated and the name of the constraint block is prefixed with the name of the unique field name within that register.

Example 15-9

```
ral_model.r1.consistency.constraint_mode(0);
```

16

Maximum Data Size

By default, the maximum size of fields, registers and memories is limited to 64 bits. This limitation is enforced by all methods having `bit [63:0]` data arguments. Smaller fields, registers and memories are intrinsically supported by using the SystemVerilog and OpenVera automatic value extension and truncation.

RAL version 1.1.0 and greater supports variable maximum data size. The maximum data size may be reduced to save memory in large RAL models. It may also be increased to support larger fields, register or memory values. The following instructions use a 128-bit maximum data size example as an illustration. The number of bits does not have to be a power of two and can be smaller than 64.

`ralgen` will issue a warning about the necessity of defining a new maximum data size should a RALF file specify fields, registers or memories with more than 64 data bits.

The maximum data size is defined globally, for all RAL model components. If different RAL models, with different maximum data size requirements are included in the same simulation, you must use the largest maximum data size.

When specifying a different maximum data size, all data arguments in the methods documented in Appendix B are redefined using `bit [VMM_RAL_DATA_WIDTH-1:0]` instead of `bit [63:0]`.

Maximum Data Size in SystemVerilog

By default, the maximum data size in SystemVerilog is 64 bits. You can change this default to a larger or smaller value by defining the `VMM_RAL_DATA_WIDTH` symbol to a suitable value.

You can define the symbol value using a command-line option, as shown in the following example:

```
% vcs ... +define+VMM_RAL_DATA_WIDTH=128 ...
```

You can also specify the symbol value in SystemVerilog source files, before the first inclusion of the `vmm_ral.sv` file, as shown in the following example:

```
`define VMM_RAL_DATA_WIDTH 128
`include "vmm_ral.sv"
```

Maximum Data Size in OpenVera

By default, the maximum data size in OpenVera is 64 bits. You can change this default to a larger or smaller value by defining the `VMM_RAL_DATA_WIDTH` symbol to a suitable value.

Maximum Data Size using Vera

The VRO file shipped with Vera or in a separate VMM installation, is precompiled for a maximum data size of 64 bits. If a different maximum data size is required, a new VRO file must first be created.

The VRO file is created by compiling the `vmm_ral.vrp` file with the symbol value defined using a command-line option. If compiling from a VMM installation shipped with Vera, use the following command:

```
vera -cmp -vip -DVMM_RAL_DATA_WIDTH=128 \
     ${VERA_HOME}/vrp/vmm_ral.vrp .
```

If compiling from a separate VMM installation, use the following command:

```
vera -cmp -vip -DVMM_RAL_DATA_WIDTH=128 \
     ${VMM_HOME}/ov/vrp/vmm_ral.vrp .
```

You must specify the <u>same</u> maximum data size when compiling any source code that includes the `vmm_ral.vrh` file. You can accomplish this through a command-line symbol definition, or by explicitly defining the symbol in a source file before the first inclusion of the VRH file, as shown in one of the following Example:

```
vera -cmp -DVMM_RAL_DATA_WIDTH=128 ...
```

or:

```
#define VMM_RAL_DATA_WIDTH 128
#include "vmm_ral.vrh"
```

Important: When using a separate VMM installation, you must use a fully qualified path to include the VRH file, as shown in the following example:

```
#define VMM_RAL_DATA_WIDTH 128
#include "../path/to/vmm/installation/ov/include/
vmm_ral.vrh"
```

Maximum Data Size Using VCS

You can define the symbol by using a command-line option, as shown in the following example:

```
vcs ... -ntb_define VMM_RAL_DATA_WIDTH=128 ...
```

You can also specify the symbol value in OpenVera source files, before the first inclusion of the `vmm_ral.vrp` file, as shown in the following example:

```
#define VMM_RAL_DATA_WIDTH 128
#include "vmm_ral.vrp"
```

17

C Interface

The RAL C interface allows firmware and application-level code[1] to be developed and debugged on a simulation of the design.

You can access the fields, registers and memories included in a RAL model in C code through a C API. The C code is executed natively on the same workstation that is running the SystemVerilog simulation, eliminating the need for an instruction set simulator or a RTL model of the processor. You can compile the same C code later for the target execution processor.

As illustrated in Figure 17-1, two versions of the RAL C API can be generated. One is designed to interface to the RAL model running in the SystemVerilog simulator using the Direct Programming Interface. The other is pure stand-alone C code and is designed to

1. Clearly, runtime performance allows only the lower layers of an application to be simulated.

be compiled on the target processor in the final application. This allows the firmware and application-level code to be verified against a simulation and then used, unmodified, in the final application.

Figure 17-1 RAL-Generated APIs

Both versions of the RAL C API implement the API that is specified in Appendix C, "RAL C API Macros and Functions".

Entry Point

When executing the C code within a simulation, it is necessary for the C code to be called by the simulation to be executed. To that end, the application software's `main()` routine must be replaced by one or more entry points known to the simulation.

There are two kinds of entry points: application entry points and service entry points. Both use the same interface mechanism. They only differ in their intent and timing of the invocation.

All entry points must take at least one argument that will receive the base address of the RAL model to be used by the C code. The RAL model reference is a `size_t` on the C side and refers to the value

returned by the "vmm_ral_block_or_sys::C_addr_of()" method. The C-side reference is then used by the RAL C API to access required fields, registers, or memories, as specified in Appendix C.

Entry points must be declared in the `$unit` scope, typically in the same file that defines the verification environment, as shown in Example 17-1. The arguments must require at least one instance of a block or system base address.

Example 17-1 Entry point declaration

```
`include "vmm-ral.sv"
import "DPI-C" context task C_func_name(int blk);
```

The entry point must correspond to a `void` C function with a corresponding `size_t` argument for the base address of the top-level block or system, as shown in Example 17-2.

Example 17-2 Entry point declaration in C, corresponding to Example 17-1

```
void C_func_name(size_t blk)
{
   ...
}
```

Application Entry Point

An application entry point is invoked only once, at the beginning of the simulation. It replaces the application's `main()` routine that would normally be automatically invoked by the host operating system. An application entry point may or may not complete and return control of the execution thread back to the SystemVerilog side.

If a simulation contains more than one "application", or if the application software is fragmented into different functions, the entry point for each application or application fragment may be called by the simulation or internally by a top-level entry point.

An application entry point should be invoked by the testcase or verification environment at a point where the simulated design is ready to be used by the application software. If the application software includes the device configuration firmware, or is the actual device configuration firmware, the entry point should be invoked as the implementation of the `cfg_dut()` method, as shown in Example 17-3. If the application software requires a fully functional and configured model of the design, its entry point should be invoked after the environment has been started, as shown in Example 17-4. Notice how each entry point invocation is forked to allow for an application software that never completes.

Example 17-3 Invoking application containing configuration firmware

```
import "DPI-C" context task appsw(vmm_ral_block blk);

class my_env extends vmm_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      fork
         appsw(this.ral_model.C_addr_of());
      join_none
   endtask
   ...
endclass
```

Example 17-4 Application requiring pre-configured simulation

```
import "DPI-C" context task appsw(vmm_ral_sys sys);

program test;

tb_env env = new;
```

```
initial
begin
   env.start();
   fork
      appsw(env.ral_model.C_addr_of());
   join_none
   env.run();
end
endprogram
```

Service Entry Point

A service entry point is invoked multiple times whenever a condition requiring software servicing is detected. Unlike an application entry point, a service entry point always returns, at which point it may be invoked again.

A typical service entry point is the invocation of the interrupt service code in the application software, as illustrated in Example 17-5. Notice how its invocation is embedded in a `forever` loop to ensure that interrupts will be serviced whenever the `tb_top.int` signal is asserted.

Example 17-5 Invoking application containing configuration firmware

```
import "DPI-C" context task isr(vmm_ral_sys sys);

class my_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      fork
         forever begin
            wait (tb_top.int === 1'b1);
            isr(this.ral_model);
         end
      join_none
   endtask
   ...
endclass
```

Execution Timeline

When executing with a simulation of the design, all C code executes atomically. It is unlike the real application code running as object code on a real processor, where the execution of the code happens concurrently with other processing in the neighboring hardware. When C code executes, only that code performs any form of processing and the simulation of the rest of the design is frozen. The only way for the design simulation to proceed, is for the C code to return, or for the C code to perform a read or write operation through the RAL model. In the latter case, once the read or write operation completes and the control is returned back to the C code, the simulation is again frozen.

Figure 17-2 illustrates the execution timeline. The entire execution timeline in the C code occurs in zero-time in the simulation timeline. The execution timeline is the cause of the important impact on runtime performance of how the C code interacts with the design.

Figure 17-2 C code and Simulated Design Execution Timeline

If a polling strategy is used, the simulation will have the opportunity to advance only during the execution of the repeated polling read cycles. It would likely require many hundreds of such read cycles for the design to reach a state that is relevant and significant for the application software. With physical device, that is not an issue as this can happen in less than a microsecond. However, in a

simulation, this would require a lot of processing for simulating essentially useless read cycles and exchanging data between the C world and the simulation world.

If an interrupt-driven strategy is used, the simulation will proceed until something of interest to the application software has happened before transferring the control to the C code (through a service entry point), and only the necessary read and write operations would need to be performed. Therefore, it is important that you use a service-based approach as much as possible.

It is also very important that the execution of the C code not be blocked by an external event—such as waiting for user input or a file to be unlocked—as it will prevent the simulation from moving forward while it is blocked. If the application software requires such synchronization, it should similarly use an asynchronous service-driven approach.

Writing Firmware Code

The RAL C API is designed to ultimately yield compact and efficient firmware code, while preserving as much as possible of the abstraction offered by the RAL model. To that effect, the RAL C API hides the physical addresses of registers and the position and size of fields. The hiding is performed by functions and macros rather than an object-oriented structure like the native RAL model in SystemVerilog. This is eliminates the need to compile a complete object-oriented model in an embedded processor object code with limited amount of memory.

Figure 17-3 Example register layout

Example 17-6 Example RALF specification

```
block myblk {
   bytes 4;
   endian big;
   register R1 @0x0100 {
      bytes 4;
      field a1 { bits 4; access rw; }
      field X  { bits 7; access rw; }
      field a2 { bits 21; access rw; }
   }
   register R2 @0x104 {
      bytes 12;
      field a1 { bits 32; access rw; }
      field Y @32 { bits 7; access rw; }
      field a2 @64 { bits 30; access rw; }
   }
   register R3 @0x110{
      bytes 4;
      field a1 { bits 32; access rw; }
   }
   memory m {
      bits 32
      size 1024
   }
}
```

Accessing Registers

Ideally, registers and memories should be of the same size as the native data bus size of the target process—typically the `int` type in the C code—and, therefore, live at a single physical address (called

"segment"). This allows each register to be accessed or updated using a single read or write operation. The RAL C interface supports single-segment, as well as, multi-segment registers.

Registers can be accessed using the `ral_read_<reg>_in_<block>` and `ral_write_<reg>_in_<block>` macros. Example 17-7 shows how a single-segment, such as register R1 specified in Figure 17-3, would be accessed.

Example 17-7 Accessing single-segment register from C

```
void C_func_name(void* blk)
{
   unsigned int r1 = 0xABCD;
   ral_write_R1_in_myblk(blk, &r1);
   r1 = 0;
   ral_read_R1_in_myblk(blk, &r1);
}
```

If a register spans multiple physical locations, such as register `R2` in Figure 17-3, the entire register can be accessed using the same macros, but by using an array of `int` with one element per physical location, as shown in Example 17-8.

Example 17-8 Accessing an entire multi-address register

```
int r2[3];
r2 = {0xABCD, 0x1234, 0x5678};
ral_write_R2_in_myblk(blk, r2);
r2 = {0x0, 0x0, 0x0};
ral_read_R2_in_myblk(blk, r2);
```

Endian Support

The RAL C interface always reads multi-segment registers into the `int` array, (`r2` in Example 17-8), in little-endian order, irrespective of their actual hardware layout, as specified in the RALF file. This

means that the segments of a multi-segment register in a big-endian block or system will be in the reverse order in the C `int` array. RAL models that use FIFO ordering are not supported.

Accessing Fields

You can access individual fields using the corresponding `ral_read_<field>_in_<block>_<reg>` and `ral_write_<field>_in_<block>_<reg>` macros. If the field has a unique name in its enclosing block, then there will also be macros named `ral_read_<field>_in_<block>` and `ral_write_<field>_in_<block>` with identical respective functionality as the previous set of macros. Example 17-9 and Example 17-10 show how the field named `X`, as illustrated in Figure 17-3, would be accessed.

Example 17-9 Accessing a block-unique field

```
unsigned int x = 0x0F;
ral_write_X_in_myblk(blk, &x);
x = 0;
ral_read_X_in_myblk(blk, &x);
```

Example 17-10 Accessing field in a specific register

```
unsigned int x = 0x0F;
ral_write_X_in_myblk_R1(ral_addr_of_R1_in_myblk(blk), &x);
x = 0;
ral_read_X_in_myblk_R1(ral_addr_of_R1_in_myblk(blk), &x);
```

Currently, fields with a size greater than `sizeof(int)` and fields spanning multiple physical addresses are not supported. If larger fields are required, or fields need to span more than one physical address, it will be necessary to access them using the multi-segment register access procedures and the resulting array-of-int value.

Accessing Memories

Memories can be accessed using the `ral_read_<mem>_in_<block>` and `ral_write_<mem>_in_<block>` macros. Example 17-11 shows how to read and write a memory location, offset 0x100 of memory `m`, specified in Example 17-6.

Example 17-11 Accessing a single-segment memory location

```
unsigned int mem_loc = 0xAABBCCDD;
ral_write_m_in_myblk(blk, 0x100, &mem_loc);
mem_loc = 0;
ral_read_m_in_myblk(blk, 0x100, &mem_loc);
```

Note that a multi-segment memory location would be accessed by using the same macro. The only difference is that a suitably-sized array of `int` would need to be supplied to hold the memory location value to be read or written.

18

Using the RALF C++ API

The RALF specification contains all of the necessary information to generate RTL code that implements the specified registers. There are many ways to code and implement RTL, so it is not possible to provide a general purpose solution. Using the RALF C++ API, you have full access to parsed RALF data (through C++ routines), which you can use to implement a customized RTL code generator, or any other feature that needs RALF data.

This chapter contains the following sections:

- “RALF C++ API Code Example”
- “Compiling”
- “Specifying the Library Path”

RALF C++ API Code Example

To use the RALF C++ API, you include `$VCS_HOME/include/ralf.hpp` in your C++ source code, as shown in Example 18-1, and you apply the API as explained in this chapter.

Example 18-1

```
#include <iostream>
#include "ralf.hpp"

int main(int argc, char *argv[])
{
   // Check basic command line usage…
   if (argc < 3) {
      fprintf(stderr, "Error: Wrong Usage.\n");
      // Show Correct Usage …
      exit(1);
   }

   /**
   * Parse command line arguments to get the essential
   * constructor arguments. See Appendix E for documentation
   * of class ralf::vmm_ralf's constructor parameters.
   */
   const char * top = argv[1];
   const char * ralf_file = argv[2];
   const char * rtl_dir = NULL;
   const char * inc_dir = NULL;
   if (argc == 4) rtl_dir = argv[3];
   if (argc == 5) {
      rtl_dir = argv[3];
      inc_dir = argv[4];
   }

   /**
   * Create a ralf::vmm_ralf object by passing in proper
   * constructor arguments. See Appendix E for documentation
   * of class ralf::vmm_ralf's constructor parameters.
   */
```

```
   ralf::vmm_ralf ralf_data(ralf_file, top, rtl_dir,
      inc_dir);

   /**
   * Get the top level object storing the parsed RALF
   * block/system data and traverse that, top-down, to get
   * access to complete RALF data.
   */
   const ralf::vmm_ralf_blk_or_sys_typ * top_lvl_blk_or_sys
      = ralf_data.getTopLevelBlockOrSys();

   #ifndef GEN_RTL_IN_SNPS_STYLE
      /*
      * TODO-Traverse the parsed RALF data structure top-down
   * using/starting-from 'top_lvl_blk_or_sys' for getting
   * complete access to the RALF data and then, do whatever
   * you would want to do, with the parsed RALF data. One
   * typical usage of parsed RALF data could be, to generate
   * RTL code in your own style.
   */
   //
   // TODO - Add your RTL generator code here.
   //
   #else
      /*
      * As part of this library, Synopsys also provides a
      * default RTL generator, which can be invoked by
      * invoking 'generateRTL()' method of 'ralf::vmm_ralf'
      * class, as demonstrated below.
      */
      ralf_data.generateRTL();
   #endif
}
```

Compiling

This section contains the following topics:

- "Compilation Command"

- “Compiling in RHEL32”
- “Compiling in Solaris”

Compilation Command

Use the following command to compile the C++ source file:

```
$CPP_COMPILER $CFLAGS -I${VCS_HOME}/include -L${VCS_HOME}/
lib -lralf your-file.cpp $LDFLAGS
```

Compiling in RHEL32

$CPP_COMPILER

- The compiler in RHEL32 is typically `g++`.
- This library has been tested with the C++ compiler and library available with gcc 4.2.2.

$CFLAGS

- For 32-bit `$VCS_HOME/lib/libralf.so`, ensure that `$CFLAGS` has the necessary flag for creating a 32-bit binary.
- For 64-bit `$VCS_HOME/lib/libralf.so`, ensure that `$CFLAGS` has the necessary flag for creating a 64-bit binary.

$LDFLAGS

- Ensure that **`-ldl`** is specified in `$LDFLAGS`.

Compiling in Solaris

$CPP_COMPILER

- The compiler in Solaris is typically `cc`.
- This library has been tested with Sun CC 5.8 compiler.

$CFLAGS

- For 32-bit `$VCS_HOME/lib/libralf.so`, ensure that `$CFLAGS` has the necessary flag for creating a 32-bit binary.
- For 64-bit `$VCS_HOME/lib/libralf.so`, ensure that `$CFLAGS` has the necessary flag for creating a 64-bit binary.

$LDFLAGS

- Ensure that **`-lsocket`** is specified in `$LDFLAGS`.

Specifying the Library Path

Before you run the executable, ensure that `${VCS_HOME}/lib` is present in your `LD_LIBRARY_PATH`.

19

Generating RALF from IP-XACT

The registers and memories in the design under verification are usually described in a RALF file for VMM RAL. You create this description based on your design register specification. The register specification is part of an architecture/design document usually created in a format such as FrameMaker, Microsoft Word, or a spreadsheet. Since there is no common standard text format that is used in the industry, every user has slightly different variations in describing the register specifications. IP-XACT is becoming a standard for describing register specifications.

After the register specification is converted to a common meta-data model, such as the IP-XACT schema, then you can use the **`ralgen`** utility to automatically create a RALF file description. As discussed in "RALF File Description Mechanism" , the RALF model is used by **`ralgen`** to generate the corresponding RAL model for verification.

Definition of the IP-XACT Schema

IP-XACT is a standard specification for eXtensible Markup Language (XML) meta-data and tool interfaces that is an industry intermediate specification format.

The IP-XACT standard specification is a mechanism to document and exchange information about design IP, its characteristics and its required configuration and integration. The memory and register specification is also described using the IP-XACT schema. The IP-XACT meta-data was conceived by the SPIRIT consortium.

The IP-XACT XML description is generated by the user from the original register specification using a user-supplied conversion script.

RALF File Description Mechanism

The default generated RALF model maps the XML specification file to generic RALF syntax format. To generate the RALF file from an IP-XACT file, **`ralgen`** is invoked with the **`-ipxact2ralf`** option.

For example, the following command can be used to generate a RALF model for cpu_regs registers, if the cpu_reg.xml file exists:

```
% ralgen -ipxact2ralf cpu_regs.xml
```

The generated file will be named cpu_regs.ralf, which contains RALF descriptions of the registers. Example 19-1 shows the register description in IP-XACT schema, and its equivalent RALF format.

Example 19-1 Generating RALF from IP-XACT

cpu_regs.xml:

```
...
<spirit:register>
      <spirit:name>r2</spirit:name>
      <spirit:addressOffset>0x8</spirit:addressOffset>
      <spirit:size>64</spirit:size>
      <spirit:access>read-write</spirit:access>
   <spirit:field>
      <spirit:name>f2</spirit:name>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:bitWidth>1</spirit:bitWidth>
      <spirit:access>read-write</spirit:access>
   </spirit:field>
   ...
</spirit:register>
```

cpu_regs.ralf:

```
...
register r2 @'h8 {
   field f2 {
      bits 1;
      access rw;
   }
...
}
```

In the above, only a RALF file is generated. The next step is to generate all necessary RAL files by invoking **`ralgen`** a second time, with appropriate switches, using the generated RALF file.

Figure 19-1 shows the steps involved in this process.

Figure 19-1 RALF generation and RAL generation

Supported IP-XACT Schema

The **`ralgen`** utility accepts IP-XACT schema version 1.4 descriptions for the registers and memories of the design. The conversion utility supports the XSD schema, as this is the schema used for IP-XACT descriptions.

Generic RALF Features and IP-XACT Mapping

Table 19-1 lists the generic IP-XACT features and their RALF equivalents supported by this conversion utility.

Table 19-1 RALF Equivalents of IP-XACT Features

Spirit IP-XACT 1.4 Description	**RALF Generic Feature**
<spirit:**name**>***name***</spirit:**name**>	name
<spirit:**description**>***description***</spirit:**description**>	description, doc
<spirit:**access**>*access_mode*</spirit:**access**>	access
<spirit:**reset**> ... </spirit:**reset**>	reset, hard_reset
<spirit:**value**>***reset_value***</spirit:**value**>	reset_value

Table 19-2 lists the generic IP-XACT access modes and their RALF equivalents supported by this conversion utility:

Table 19-2 RALF Equivalents of IP-XACT Access Modes

IP-XACT Definition access_mode	RALF Register Access Mode
read-write	rw
read -only	ro
write -only	wo

Specific RALF Features and IP-XACT Mapping

The following tables list the specific RALF syntax and the equivalent IP-XACT features supported by this conversion utility.

field

`field` *`name`* `[{`*`properties`*`}]`

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**field**> ... </spirit:**field**>	field
<spirit:**bitOffset**>***field_bit_offset***</spirit:**addressOffset**>	*@field_bit_offset*
<spirit:**bitWidth**>***number_of_bits_in_field***</spirit:**bitWidth**>	bits
<spirit:**access**>***access_mode***</spirit:**access**>	access

register

`register` *`name {properties}`*

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**register**> ... </spirit:**register**>	register
<spirit:**addressOffset**>*register_bit_offset*</ spirit:**addressOffset**>	@'*register_bit_offset*

block

`block` *`name {property}`*

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**addressBlock**> ... </spirit:**addressBlock**>	block
<spirit:**baseAddress**>'*block_start_address*</ spirit:**baseAddress**>	@'*block_start_address*

memory

`memory` *`name {property}`*

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**register**> ... </spirit:**register**>	register
<spirit:**baseAddress**>'*memory_start_offset* </spirit:**baseAddress**>	@'*memory_start_offset*
< spirit:**size**>*number_of_rows*</spirit:**size**>	[size]
<spirit:**bitWidth**>*number_of_bits_in_each_row* </spirit:**bitWidth**>	bits, bytes

register array

`register array`

Spirit IP-XACT Equivalent	RALF Feature
<spirit:**register**> ... </spirit:**register**>	register
<spirit:**baseAddress**>'*array_start_offset* </spirit:**baseAddress**>	@'*array_start_offset*
< spirit:**size**>*number_of_array_elements*</spirit:**size**>	[size]
<spirit:**bitWidth**>*number_of_bits_in_each_element* </spirit:**bitWidth**>	bits / bytes

system

`system` *`name`* {*`property`*}

Spirit IP-XACT Equivalent	RALF Feature
<spirit:memoryMap> ... </spirit:memoryMap>	system

Limitations of IP-XACT to RALF Feature Mapping

The **`ralgen`** utility has no mapping for the IP-XACT memory schema features or syntax listed in Table 19-3.

Table 19-3 IP-XACT Memory Schema Features with No ralgen Mapping

bank parallel
reserved
ref: volatile
ref: parameters
dim
mask

Table 19-3 IP-XACT Memory Schema Features with No ralgen Mapping

values: value/name/description
ref:vendorExtensions
suspaceMap
masterRef
nameGroup

There are some RALF features with no direct equivalence as yet in the IP-XACT version 1.4 memory/registers schema. Table 19-4 lists the RALF syntax items that are not available in IP-XACT version 1.4 syntax.

Table 19-4 RALF Features with No Direct IP-XACT 1.4 Equivalent

regfile
virtual register
domain
constraint
initial
initial_value
hdl_path
reset_type
soft_reset
little
big
fifo_ls

Table 19-4 RALF Features with No Direct IP-XACT 1.4 Equivalent

fifo_ms
endian
endian_value
RALF Access Modes:
w1, ru, w1c, rc, a1, a0, other, user0, user1, user2, user3, dc

20

RAL C Interface

The RAL C interface allows firmware and application-level code to be developed and debugged on a simulation of the design. For runtime performance reasons, only the lower layers of an application are simulated.

You can access the fields, registers, and memories included in a RAL model in C code through a C API. The C code is executed natively on the same workstation that is running the SystemVerilog simulation, eliminating the need for an instruction set simulator or a RTL model of the processor. You can compile the same C code later for the target execution processor.

This chapter contains the following sections:

- “RAL C API Versions”
- “Entry Point”
- “Execution Timeline”

- “Writing Firmware Code”

RAL C API Versions

As illustrated in Figure 20-1, two versions of the RAL C API can be generated. One is designed to interface to the RAL model running in the SystemVerilog simulator using the Direct Programming Interface. The other is pure stand-alone C code and is designed to be compiled on the target processor in the final application. This allows the firmware and application-level code to be verified against a simulation and then used, unmodified, in the final application.

Figure 20-1 RAL-Generated APIs

Note:

The “RAL C interface” to “OV RAL model” is not supported. Only “RAL C interface” to “SV RAL model” and “RAL C interface” to “RAL pure C model” are supported.

Entry Point

This section contains the following topics:

- “Overview of Entry Points”
- “Application Entry Point”
- “Service Entry Point”

Overview of Entry Points

When executing the C code within a simulation, it is necessary for the C code to be called by the simulation to be executed. To that end, the application software’s `main()` routine must be replaced by one or more entry points known to the simulation.

There are two kinds of entry points: application entry points and service entry points. Both use the same interface mechanism. They only differ in their intent and timing of the invocation.

All entry points must take at least one argument that will receive the base address of the RAL model to be used by the C code. The RAL model reference is a `size_t` on the C side and refers to the value returned by the **`vmm_ral_block_or_sys::C_addr_of()`** method. The C-side reference is then used by the RAL C API to access required fields, registers, or memories, as specified in Appendix C.

Entry points must be declared in the `$unit` scope, typically in the same file that defines the verification environment, as shown in Example 20-1. The arguments require at least one instance of a block or system base address.

Example 20-1 Entry point declaration

```
`include "vmm-ral.sv"
import "DPI-C" context task C_func_name(int blk);
```

The entry point must correspond to a `void` C function with a corresponding `size_t` argument for the base address of the top-level block or system, as shown in Example 20-2.

Example 20-2 Entry point declaration in C, corresponding to Example 20-1

```
void C_func_name(size_t blk)
{
   ...
}
```

Application Entry Point

An application entry point is invoked only once, at the beginning of the simulation. It replaces the application's `main()` routine that would normally be automatically invoked by the host operating system. An application entry point may or may not complete and return control of the execution thread back to the SystemVerilog side.

If a simulation contains more than one "application", or if the application software is fragmented into different functions, the entry point for each application or application fragment may be called by the simulation or internally by a top-level entry point.

An application entry point should be invoked by the testcase or verification environment at a point where the simulated design is ready to be used by the application software. If the application software includes the device configuration firmware, or is the actual device configuration firmware, the entry point should be invoked as the implementation of the `cfg_dut()` method, as shown in Example 20-3. If the application software requires a fully functional and configured model of the design, its entry point should be invoked

after the environment has been started, as shown in Example 20-4. Notice how each entry point invocation is forked to allow for an application software that never completes.

Example 20-3 Invoking application containing configuration firmware

```
import "DPI-C" context task appsw(vmm_ral_block blk);

class my_env extends vmm_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      fork
         appsw(this.ral_model.C_addr_of());
      join_none
   endtask
   ...
endclass
```

Example 20-4 Application requiring pre-configured simulation

```
import "DPI-C" context task appsw(vmm_ral_sys sys);

program test;

tb_env env = new;

initial
begin
   env.start();
   fork
      appsw(env.ral_model.C_addr_of());
   join_none
   env.run();
end
endprogram
```

Service Entry Point

A service entry point is invoked multiple times whenever a condition requiring software servicing is detected. Unlike an application entry point, a service entry point always returns, at which point it may be invoked again.

A typical service entry point is the invocation of the interrupt service code in the application software, as illustrated in Example 20-5. Notice how its invocation is embedded in a `forever` loop to ensure that interrupts will be serviced whenever the `tb_top.int` signal is asserted.

Example 20-5 Invoking application containing configuration firmware

```
import "DPI-C" context task isr(vmm_ral_sys sys);

class my_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      fork
         forever begin
            wait (tb_top.int === 1'b1);
            isr(this.ral_model);
         end
      join_none
   endtask
   ...
endclass
```

Execution Timeline

When executing with a simulation of the design, all C code executes atomically. It is unlike the real application code running as object code on a real processor, where the execution of the code happens concurrently with other processing in the neighboring hardware.

When C code executes, only that code performs any form of processing and the simulation of the rest of the design is frozen. The only way for the design simulation to proceed, is for the C code to return, or for the C code to perform a read or write operation through the RAL model. In the latter case, once the read or write operation completes and the control is returned back to the C code, the simulation is again frozen.

Figure 20-2 illustrates the execution timeline. The entire execution timeline in the C code occurs in zero-time in the simulation timeline. The execution timeline is the cause of the important impact on runtime performance of how the C code interacts with the design.

Figure 20-2 C code and Simulated Design Execution Timeline

If a polling strategy is used, the simulation will have the opportunity to advance only during the execution of the repeated polling read cycles. It would likely require many hundreds of such read cycles for the design to reach a state that is relevant and significant for the application software. With a physical device, that is not an issue as this can happen in less than a microsecond. However, in a simulation, this would require a lot of processing for simulating essentially useless read cycles and exchanging data between the C world and the simulation world.

If an interrupt-driven strategy is used, the simulation will proceed until something of interest to the application software has happened before transferring control to the C code (through a service entry

point), and only the necessary read and write operations would need to be performed. Therefore, it is important that you use a service-based approach as much as possible.

It is also very important that the execution of the C code not be blocked by an external event—such as waiting for user input or a file to be unlocked—as it will prevent the simulation from moving forward while it is blocked. If the application software requires such synchronization, it should similarly use an asynchronous service-driven approach.

Writing Firmware Code

This section contains the following topics:

- “Firmware Code Overview”
- “Accessing Registers”
- “Accessing Fields”
- “Accessing Memories”

Firmware Code Overview

The RAL C API is designed to ultimately yield compact and efficient firmware code, while preserving as much as possible of the abstraction offered by the RAL model. To that effect, the RAL C API hides the physical addresses of registers and the position and size of fields. The hiding is performed by functions and macros rather than an object-oriented structure like the native RAL model in

SystemVerilog. This eliminates the need to compile a complete object-oriented model in an embedded processor object code with a limited amount of memory.

Figure 20-3 Example register layout

Example 20-6 Example RALF specification

```
block myblk {
   bytes 4;
   endian big;
   register R1 @0x0100 {
      bytes 4;
      field a1 { bits 4; access rw; }
      field X  { bits 7; access rw; }
      field a2 { bits 21; access rw; }
   }
   register R2 @0x104 {
      bytes 12;
      field a1 { bits 32; access rw; }
      field Y @32 { bits 7; access rw; }
      field a2 @64 { bits 30; access rw; }
   }
   register R3 @0x110{
      bytes 4;
      field a1 { bits 32; access rw; }
   }
   memory m {
      bits 32
      size 1024
   }
}
```

Accessing Registers

Ideally, registers and memories should be the same size as the native data bus size of the target process—typically the `int` type in the C code—and, therefore, live at a single physical address (called "segment"). This allows each register to be accessed or updated using a single read or write operation. The RAL C interface supports single-segment, as well as, multi-segment registers.

Registers can be accessed using the `ral_read_<reg>_in_<block>` and `ral_write_<reg>_in_<block>` macros. Example 20-7 shows how a single-segment, such as register R1 specified in Figure 20-3, would be accessed.

Example 20-7 Accessing single-segment register from C

```
void C_func_name(void* blk)
{
   unsigned int r1 = 0xABCD;
   ral_write_R1_in_myblk(blk, &r1);
   r1 = 0;
   ral_read_R1_in_myblk(blk, &r1);
}
```

If a register spans multiple physical locations, such as register `R2` in Figure 20-3, the entire register can be accessed using the same macros, but by using an array of `int` with one element per physical location, as shown in Example 20-8.

Example 20-8 Accessing an entire multi-address register

```
int r2[3];
r2 = {0xABCD, 0x1234, 0x5678};
ral_write_R2_in_myblk(blk, r2);
r2 = {0x0, 0x0, 0x0};
ral_read_R2_in_myblk(blk, r2);
```

Endian Support

The RAL C interface always reads multi-segment registers into the `int` array, (`r2` in Example 20-8), in little-endian order, irrespective of their actual hardware layout, as specified in the RALF file. This means that the segments of a multi-segment register in a big-endian block or system will be in the reverse order in the C `int` array. RAL models that use FIFO ordering are not supported.

Accessing Fields

You can access individual fields using the corresponding `ral_read_<field>_in_<block>_<reg>` and `ral_write_<field>_in_<block>_<reg>` macros. If the field has a unique name in its enclosing block, then there will also be macros named `ral_read_<field>_in_<block>` and `ral_write_<field>_in_<block>` with identical respective functionality as the previous set of macros. Example 20-9 and Example 20-10 show how the field named `X`, as illustrated in Figure 20-3, would be accessed.

Example 20-9 Accessing a block-unique field

```
unsigned int x = 0x0F;
ral_write_X_in_myblk(blk, &x);
x = 0;
ral_read_X_in_myblk(blk, &x);
```

Example 20-10 Accessing field in a specific register

```
unsigned int x = 0x0F;
ral_write_X_in_myblk_R1(ral_addr_of_R1_in_myblk(blk), &x);
x = 0;
ral_read_X_in_myblk_R1(ral_addr_of_R1_in_myblk(blk), &x);
```

Currently, fields with a size greater than `sizeof(int)` and fields spanning multiple physical addresses are not supported. If larger fields are required, or fields need to span more than one physical address, it will be necessary to access them using the multi-segment register access procedures and the resulting array-of-int value.

Accessing Memories

Memories can be accessed using the `ral_read_<mem>_in_<block>` and `ral_write_<mem>_in_<block>` macros. Example 20-11 shows how to read and write a memory location, offset 0x100 of memory `m`, specified in Example 20-6.

Example 20-11 Accessing a single-segment memory location

```
unsigned int mem_loc = 0xAABBCCDD;
ral_write_m_in_myblk(blk, 0x100, &mem_loc);
mem_loc = 0;
ral_read_m_in_myblk(blk, 0x100, &mem_loc);
```

Note that a multi-segment memory location would be accessed by using the same macro. The only difference is that a suitably-sized array of `int` would need to be supplied to hold the memory location value to be read or written.

21

Automatic Mirror Updating Techniques

By default, the mirrored field values in a RAL model are updated only when the fields are accessed through the RAL model based on the current mirrored value, the accessed data value and the type of the field. Any changes to the field value performed by the design itself or through read/write cycles performed directly on the physical interface are not visible to the RAL model.

The RAL auto mirror update feature provides two mirror updating capabilities:

First, by observing the SystemVerilog constructs which is used to store the field values in the design itself (direct update).

And the next, by observing bus transactions through a monitor on the physical interface (passive update).

The former can be used to automatically update the mirror value of fields that are modified by the design itself, for example, a counter. The latter can be used to automatically update the mirror value of fields that are accessed outside of the RAL model, for example, another master on the bus.

Direct Update

The direct update mechanism is implemented as part of the register backdoor access mechanism. It can only be used when the backdoor access classes are generated with the `-b` option. Any field part of a register where the registered backdoor access class indicates that auto updating is provided has its mirror value updated strictly and only through the direct update mechanism. Executed or observed transactions are not used to update its mirrored value.

When using `ralgen` to automatically generated backdoor access classes, the direct update mechanism will not be generated by default. The direct update code is generated only with the use of the `-auto_mirror` command-line option.

The direct update thread immediately updates the mirror value of all the fields in the register, then, waits for any change in any of the fields in the register. As soon as a change is observed, the mirror value is updated again. The direct update thread is sensitive to value changes in the storage constructs used to model the fields, and not on the clock signal used to synchronize the update; therefore, simulation performance will only be affected when an actual change occur.

However, it is possible that fields have a high update frequency, for example, free-running counters would adversely affect the simulation performance if the direct updating capability was required for such fields. Therefore, it is possible to turn off direct update for individual fields, registers, register files, blocks and systems.

When you specify the `-auto_mirror` option with `ralgen`, generation of the direct updating mechanism is enabled by default for all fields. However, it can be hierarchically disabled and enabled for individual field, register, register file, block and system using the `automirror on/off` option in the RALF file.

The `automirror on` option enables direct updating for all the fields in the register/file/block/system where the property is specified.

The `automirror off` option disables direct updating.

Note:The `automirror` option is declarative; only one `automirror` option can be specified and its location in the enclosing element is not significant.

For example, the following RALF file would disable direct update in all fields in block `b` but enable it for all field in register `r2` except for field `f3`.

```
block b {
   register r1 {…}
   register r2 {
      field f1 {…}
      field f2 {…}
      field f3 {
       automirror off;
         …
      }
       automirror on;
   }
   automirror off;
```

```
   register r3 {…}
}
```

For manually implemented register backdoor access classes, you can implement the direct update mechanism through the following methods in the `vmm_ral_reg_backdoor` class:

```
virtual function bit is_auto_updated(string fieldname);
virtual task wait_for_change();
```

The `vmm_ral_reg_backdoor::is_auto_updated()` must return TRUE if the specified field in the register is to be auto-updated. Else, it must return FALSE.

The `vmm_ral_reg_backdoor::wait_for_change()` method must block until a value change is observed in any of the fields in the register that are auto-updated.

For details, refer to the following RALF file:

```
block b {
   bytes 1
   automirror on

   register r1 (reg1) {
      bytes 1

      field f1 {
         bits 4
      }
      field f2 {
         automirror off
         bits 4
      }
   }
   register r2 (reg2) {
      bytes 1

      automirror off
```

```
        field f {
           bits 8
        }
     }
  }
```

For this RALF file, while manually implementing the register backdoor access class for register 'r1', you need to implement/override the `vmm_ral_reg_backdoor::is_auto_updated()` and `vmm_ral_reg_backdoor::wait_for_change()` base class methods as shown in the following example:

```
class manually_implemented_r1_bkdr extends
vmm_ral_reg_backdoor;
   :
  virtual function bit is_auto_updated(string fld_name);
      if (fld_name == "f1") begin
         return 1;
      end
      else if (fld_name == "f2") begin
         return 0;
      end
   endfunction

   virtual task wait_for_change();
      @(`B_TOP_PATH.reg1);
   endtask
endclass
```

You then start the direct update thread for this register 'r1' by invoking the `vmm_ral_reg_backdoor::start_update_thread(vmm_ral_reg rg)` method of its backdoor access class once an instance of that backdoor access class is created as shown in the following example:

```
class my_ral_block_b extends ral_block_b;

   function new(…);
```

```
      super.new(…)
      :
      //
      // Setting up backdoor access for r1...
      //
      begin
         manually_implemented_r1_bkdr bkdr = new;
         bkdr.start_update_thread(this.r1);
         this.r1.set_backdoor(bkdr);
      end
   endfunction : new
endclass : my_ral_block_b
```

Note:You do not need any of the above.....for register 'r2' because direct update mechanism has been disabled by using "automirror off" construct.

Passive Update

The passive update mechanism is implemented by overloading the `observe_single()` method in the `vmm_rw_xactor` proxy transactor.This method when overloaded must return only to report individual transactions observed on the physical interface.

Any reported transaction that was created by RAL itself by calling `execute_single()` is automatically filtered out and not considered to update the mirror. Any residual transaction is assumed to have been directly injected on the physical interface through other means and is used to update the mirrored value of the fields located at the physical address reported in the observed transaction.

Fields that straddle multiple physical addresses are accessed using multiple transactions. The mirror value of such fields is partially updated based on the portion of the field that was accessed in each individual transaction.

It is possible to use the passive update mechanism by overloading only `observe_single()` and not `execute_single()`. A passive RAL model would not perform any transaction, thus all observed transaction would be forwarded to the RAL model for updating its mirrored values.

Note: This feature is only supported for RAL SystemVerilog.

22

Sub-Register Access

In previous versions of RAL, registers were the smallest unit of physical accesses that can be performed onto the DUT. Accessing a field inside a register caused the entire register to be accessed. This is inefficient when the register must be accessed using multiple read/write cycles whereas the field could be accessed using fewer cycles.

Consider the 128-bit registers as shown in Figure 22-1. Assuming a 32-bit data bus with a little-endian layout, accessing any field in this register traditionally required four cycles at addresses 0x00, 0x01, 0x02 and 0x03 respectively. With this release of RAL, only the required accesses are performed.

Figure 22-1

For example, field `D` can be accessed using a single cycle at address 0x01. Because this field occupies an entire physical address, it does not pose a challenge.

Accessing field `C` can similarly be done using a single access at address 0x00. However, it will also access fields `B` and `A`. Accessing field `F` requires two physical accesses, at addresses 0x02 and 0x03, but would also access fields `E` and `G` at the same time.

Accessing adjacent fields might not be an issue, but if any of these field is a clear-on-read field, it will have unintended consequences.

By adding support for byte enabling (assuming the underlying physical protocol also supports it), it becomes possible to access field `C` (at address 0x00, lane #2) without affecting the other fields at the same address. And because field `F` is byte-aligned, it is possible to access it without side effects by accessing address 0x02, lane #3 and address 0x03, lane #0.

However, fields `B` and `A` would remain inaccessible without mutual side effects because they do not individually occupy an entire byte lane.

Thus, individual field access is supported for fields that are the sole occupant of one or more byte lane(s). Only unused or reserved bits are allowed to share a byte lane with an individually-accessible field. If any other field is found in a byte lane occupied by the field to be accessed, the entire register is accessed.

Individual field access is only supported for front-door accesses. When using backdoor accesses, the entire register-and thus all of the fields it contains--will always be accessed.

Support for Sub-Register Accesses

To be able to access a subset of registers, the protocol must support byte enabling across all of the data bytes. Furthermore, the `vmm_rw_xactor,` a proxy transactor must indicate to the RAL model, that the underlying protocol support byte enabling.

If the physical protocol supports byte enabling, the `vmm_rw_xactor::supports_byte_enable()` must be overloaded to return TRUE. By default, it returns FALSE and fields cannot be individually accessed.

The `vmm_rw_xactor::execute_single()` method in the proxy transactor must interpret the `vmm_rw_access:byte_enable` property as enabling the byte lane(s) corresponding to the bit(s) that are set. `byte_enable[0]` enables the least-significant bytes (for example, `data[7:0]`).

The alignment of the `data` property is the same regardless of which bytes are enabled. For example, the data for byte lane #2 is always located at `data[23:16]`, whether byte lanes #0 or #1 are enabled. When returning data, the data value for disabled byte lanes is unspecified.

By default, the value of the `vmm_rw_access:byte_enable` class property will be all 1's to enable all bytes.

The value of the `vmm_rw_access::nbits` class property will be set to the total number of enabled bits (i.e. # of enabled bytes times eight).

Backward Compatibility

The other fields located in the register that contains the accessed field, but located in different physical addresses, is no longer read or written as a side effect of accessing a field. This might cause their mirror value to be out-of-date and/or the value of the field in the DUT to be different because it would not have been implicitly written to using the current mirror value.

RAL was designed to minimize the impact of the side effects by using the mirrored values to preserve, as best as possible, the current value in other fields, so the discrepancy should be minimal.

Switching the default access path from BFM to BACKDOOR causes the other fields to be updated based on the mirrored value during `write()` operations. These same fields would not have been updated using the front-door access path.

The total number of read/write cycles executed will be smaller. This might impact random stability.

Due to the functional differences potentially created by this new functionality, compile-time macro `VMM_RAL_NO_INDIVIDUAL_FIELD_ACCESS` can be defined to restore the old behavior.

A

RALF Syntax

A RALF description is a Tcl 8.5 file. Therefore, it is possible to use programming constructs such as loops and variables to rapidly and concisely construct large register sets and memory definitions. You can also use the Tcl `source` command to perform multiple and hierarchical register specification management. Also, you can use Tcl expressions to specify register offset values, base values and register names.

The semi-colon is used as a separator and is not necessary immediately after or before a closing curly brackets.

This appendix contains the following topics:

- "Grammar Notation"
- "Useful Tcl Commands"
- "RALF Construct Summary"

Grammar Notation

The following notations are used to specify the exact syntax of RALF descriptions:

`normal`	Literal items
italics	User-specified identifiers
[...]	Optional items
<...>	Repeated items, 1 to *N* times
[<...>]	Optional repeated items, 0 to *N* times
...|...	A choice of items

This section contains the following topic:

- "Reserved Words"

Reserved Words

In addition to the SystemVerilog and OpenVera reserved words, the following words are reserved and cannot be used as user-defined identifiers:

access	field	regfile
bits	hard_reset	register
block	hdl_path	reset
bytes	initial	shared
constraint	left_to_right	size
doc	memory	soft_reset
domain	noise	system
endian	read	virtual write
		write

Useful Tcl Commands

Considering a RALF description is a Tcl file, the full power of the Tcl language becomes available. The following Tcl commands are likely to be useful:

`#comment`

Indicates single-line comments with characters following a # considered as comments.

`set name value`

Sets the specified variable to the specified value. Allows the use of variable names as mnemonics, using Tcl syntax to set and get variable values.

`source filename`

Includes the specified Tcl file. Inclusion of files enable hierarchical RALF descriptions. The filename can have an absolute path or relative path.

```
for {set i 0} {$i < 10} {incr i} {
      ...
   }
```

For loops can be used to concisely create multiple fields, registers, memories and blocks specifications. Any RALF property value can be based on the value of the loop index variable or other variables.

```
if {$var} {
      ...
   }
```

Conditionally interprets Tcl statements or RALF specifications. Allows the selection or exclusion of elements in a RALF description.

You can view a complete list of available Tcl commands by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/contents.htm

This section contains the following topic:

- "Tcl Syntax and FAQ"

Tcl Syntax and FAQ

The Tcl syntax rules can be found by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/Tcl.htm

Note that ralgen preprocesses the RALF file to escape some of its syntax elements that have special meaning in Tcl. For example, the [and] used to specify arrays are properly escaped to avoid command substitution.

Whitespace

It is important to note how Tcl breaks a command into separate words on whitespaces, quoted (") and bracketed ({ and }) text. Therefore, a RALF file is sensitive to whitespace. Do not use whitespace in your code if none is shown in this appendix. Where a

whitespace is shown, at least one must be present. For example, the following syntax is invalid because the { is considered as part of the *field* command's second argument and not a separate token:

```
## This is wrong
   field REVISION_ID @2{
      bits 8;
   }
```

This example is valid because a space is required to separate the { from the preceding Tcl command argument:

```
## This is right
   field REVISION_ID @2 {
      bits 8;
   }
```

Trailing Comments

A common mistake occurs when trying to add a trailing comment to a RALF construct using the following (erroneous) syntax:

```
register my_reg {
      ...
   } # my_reg
```

Considering that Tcl commands terminate at the end-of-line, the trailing comment is considered part of the register command. To have the trailing comment be properly interpreted as a comment, the previous Tcl command should be explicitly terminated with a semicolon, as shown in the following (correct) syntax:

```
register my_reg {
      ...
   }; # my_reg
```

RALF Construct Summary

- field ..... page A-6
- register ..... page A-12
- regfile ..... page A-19
- memory ..... page A-23
- virtual register ..... page A-27
- block ..... page A-29
- system ..... page A-37

field

A field defines an atomic set of consecutive bits. Fields are concatenated into registers.

Syntax

```
field name [{
   <properties>
}]
```

Defines a field with the specified name. If you specify the name `unused` or `reserved`, it specifies unused or reserved bits within a register and you can specify only the `bits` property. Unused bits are assumed to be read-only and have a permanent value of zero. If another behavior is expected of unused or reserved bits, such as a different read-back value, you must specify an explicit field for them.

Properties

The following properties can be used to specify the field;

`[bits n;]`

Specifies the number of bits in the field. If not specified, defaults to 1. This property can only be specified once.

```
[access    rw|ro|wo|w1|ru|w1c|rc|a1|a0|
           other|user0|user1|user2|user3|dc;]
```

Specifies the functionality of all the bits in the field when the field is written or read.

A field can be:

read/write	`rw`
read-only	`ro`
write-only	`wo`
write-once	`w1`
read-only, but value updated by the design	`ru`
write a 1 to bitwise-clear	`w1c`
clear on read	`rc`
auto-set by the design	`a1`
auto-cleared by the design	`a0`
other	`other`

user-defined behavior	`user0`
	`user1`
	`user2`
	`user3`

don't care	`dc`

A write-only field always reads back as all zeroes when read, but reads back with the last written value when peeked.

If writing to a field may cause the content of other fields to be modified, or if the behavior of a field depends on the value of another field, then specify `other`, `userN` or `dc`. All other access modes assume that fields are independent of each other.

When using the "vmm_ral_field::mirror()" method with the `check` argument specified as `vmm_ral::VERB`, the mirrored value of the field is assumed to contain the expected value of the field for all field modes except don't-care (`dc`). If a field is specified as `dc`, its mirrored value behaves as if it were a `rw` field, but it is never compared against a value read from the design.

The access types such as, `other` and `userX` (`user0, user1, user2`, etc.) are used to model field access behavior that is different from the conventional types like `ro, rw, wo` etc.

The `ru` access type is used to model register fields that are "read-only" but can be modified by the design. They are subjected to change possibly due to the changes in DUT state but `ro` fields will never change. For example, a register field that represents the "Vendor ID" would typically be static.

You can use these access types to model such specific behavior different from the conventional types. The RAL pre-defined tests like the `reg_access` test and etc. will also skip these fields.

As explained in the following example, consider a register field, `STATUS` which is write protected by another bit field `PREVENT_WRITE` i.e., `STATUS` can be written only if `PREVENT_WRITE` is zero. This type of access behavior cannot be modeled using the conventional access types and should be handled by you.

Example A-1 Modeling Fields Using User Defined Behavior With Other and UserX Access Types

```
class STATUS_field_callbacks extends
vmm_ral_field_callbacks;
bit prevent_write;
virtual task post_write(..);
  prevent_write = ral_model.PREVENT_WRITE.get();
```

```
  // If the STATUS field is currently write protected
maintain its previous value in the RAL mirror.
  if(prevent_write)
    ral_model.STATUS.mirror( , vmm_ral::QUIET);
endtask
endclass
```

If a field has different access modes in different domain, specify the sum of all modes in the `field` property, then put access restrictions in the shared `register` property instantiation. For example, a field that is read-only in one domain and write-only in another must be specified as read-write, with the register instance in the first domain specified as write-only and in the second domain as read-only.

By default, a field is writeable (`rw`).

`[reset|hard_reset` *`value`*`;]`

Specifies the hard reset value for the field. By default, a value of 0 is used.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the reset *value* in the RAL Base Class is a 2-state value.

`[soft_reset` *`value`*`;]`

Specifies the soft reset value for the field. By default, a field is not affected by a soft reset.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the soft reset *value* in the RAL Base Class is a 2-state value.

```
[<constraint name [{
     <expressions>
}]>]
```

Specifies constraints on the field value when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `value` is used to refer to the value of the field.

If a `constraint` property is not specified, the field cannot be randomized. If an unconstrained but random field is required, simply specify an empty constraint block.

```
[enum { <name[=val],> }]
```

Defines symbolic names for field values. If a value is no explicitly specified for a symbolic name, the value is the value of the previous name plus one—or zero if it is the first name.

```
[cover <+|- b|f>
```

Specifies if the bits in this fields are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the field value coverage point for this field is an explicit goal (+f), in which case its weight will be equal to the number of specified or implicit bins. If it is specified as an implicit goal (-f) as part of a cross-coverage point, its coverage point weight will be equal to zero.

```
[<coverpoint {
   <bins name [[[n]]] = { <n|[n:n],> } | default>
}>]
```

Explicitly specifies the bins in the field value coverpoint for this field. The semantics of the bin specification is identical to the SystemVerilog coverage bin specification, as defined in the section named "*Defining coverage points*" in the SystemVerilog 1800-2009 Language Reference Manual.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the field, using HTML formatting tags. This feature is currently not supported.

Example

Example A-2 1-bit read/write Field

```
field tx_en;
```

Example A-3 2-bit Randomizable Field

```
field PAR {
   bits 2;
   reset 2'b11;
   constraint valid {
      value != 2'b00;
   }
}
```

Example A-4 Explicitly specified coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
     }
   }
```

register

A register defines a concatenation of fields. Registers are used in register files and blocks.

Syntax

```
register name {
   <properties>
}
```

Defines a register with the specified name.

Properties

The following properties can be used to specify the register.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specificied by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

```
[bytes n;]
```

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. If this property is not specified, the width of the register is the minimum integral number of bytes necessary to implement all fields contained in the register.

```
[left_to_right;]
```

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register, but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of and spacing between the other fields.

```
[<field name[=rename][[n]] [(hdl_path)]
   [@bit_offset[+incr]];
[<field name [[n]] [(hdl_path)]
   [@bit_offset[+incr]] {
   <field properties>
   }>]
```

Defines and instantiates the specified field in this register. The first form specifies an instance of a previously-defined field description. The second form defines a new field description and instantiates it in the register file.

Fields separated by unused or reserved bits can be separated by specifying a field named `unused` or `reserved` of the appropriate width or by using a bit offset. A bit offset, from the least-significant bit in the register can be specified. If no bit offset is specified, the field is located immediately to the left (or right if the `left_to_right` property is specified) of the previously instantiated field. If the numerical index n is specified, an array of fields is instantiated.

Field array elements are located at consecutive offsets in the register, starting with `field[0]`, separated by a specified offset increment. The offset increment is only valid when instantiating a

field array.

Instantiating an array of fields is logically equivalent to explicitly instantiating all the individual fields. The only difference is that they will be accessible as an array in the generated SystemVerilog code.

By default, the location of the low-index field will be in the LSB (least significant bit) position. If the `left_to_right` attribute is specified for the instantiating register, the low-index field is in the MSB (most significant bit) position.

A field array is generated into a fixed-sized array of `vmm_ral_field` instances in the `vmm_ral_reg` and `vmm_ral_block` class extensions using the same naming convention as a regular field. The array is populated with individual `vmm_ral_field` class instances, one per array element, appending `[%0d]` to the field name (where`%0d` is replaced with the field index). Each instance is registered with the parent register abstraction class as if they were individually-specified fields.

Arrays of fields can be interspersed with other arrays of fields or regular fields, as long as the field themselves do not overlap.

The optional (`hdl_path)` is the hierarchical reference, within the register, to the HDL structure implementing the field. If an (`hdl_path)` is specified, direct hierarchical access to the field can be automatically generated by concatenating it with the (`hdl_path)` of the enclosing register. The (`hdl_path)` can be an expression and it must be enclosed between parentheses.

By default, the bit offset represents the position of the least-significant bit of the field with respect to the least-significant bit of the register. A value of 0 indicates a field starting in the least-significant bit of the register. If the `left_to_right` property is specified, the bit offset is specified as the offset of the most-significant bit of the field from the most-significant used bit in the register. The position of the most-significant used bit in the register, is a function of the size of, and spacing between all specified fields as fields are always left-justified, even when specifying a left-to-right order.

You must specify at least one `field` property.

Any gap in the register before and after fields is assumed to be made of unused bits that are read-only and have a permanent value of zero. If another behavior is expected from unused or reserved bits, an explicit field must be specified for them.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints on the value of the fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `fieldname.value` refers to the value of a field.

```
[noise ro|rw|no;]
```

Specifies if and how this register can be accessed during normal operations of the design without affecting the configuration or functional correctness of the device. By default, a register can be read at any time (`ro`). If `rw` is specified, this register can also be written. If `no` is specified, this register cannot be accessed in any way during normal operations. Currently unsupported.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this register is physically shared by all domains in a block that instantiates it. This property can only be used in a stand-alone register specification.

The (`hdl_path`) specifies the hierarchical access path to the physical register. It is used instead of the (`hdl_path`) specified in the block instantiating it. If an (`hdl_path`) is specified, direct hierarchical access to the shared register can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing block. The (`hdl_path`) must be enclosed in parentheses.

`[cover <+|- a|b|f>`

Specifies if the address of this register should be excluded (-a) from the block's address map coverage model.

Specifies if the bits in this register are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the fields in this registers should be included (+f) in or excluded (-f) from the field value coverage model.

cross <cross_item1> <cross_item2> [<cross_item3> …

<cross_itemN>] [{

label <cross_label_name>

}]

Specifies a cross coverage point of two or more fields or of any previously defined cross coverage point. To use a previously defined cross coverage point in another cross coverage specification, the specification of the former cross coverage point must have a label, so that it can be referenced in a later cross coverage specification, if needed by using that label.

<cross_itemN> can be, either a previously defined nonarray field name or a previous defined <cross_label_name>. For field arrays, <cross_itemN> will need to specify the exact field (array element) to be used for calculating the cross, using <fieldarray-name>[<index>] syntax, where <index> will range from 0 to field array size - 1.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the register, using HTML formatting tags. Currently unsupported.

Example

Example A-5 Attribute specification for a register

```
register R {
   ...
   attributes {
      NO_RAL_TESTS 1,
      RETAIN       1
   }
}
```

The following examples are different ways to specify the register illustrated in Figure A-1.

Figure A-1 Register Specification

CTRL	Unused	CTS	DTR	Unused	PAR	RXE	TXE
	15	12	11		3 2	1	0

Example A-6 Specification for Register in Figure A-1

```
register CTRL {
   field TXE {}
   field RXE {}
   field PAR {
      bits 2;
      reset 2'b11;
   }
   field DTR @11 {
      access ru;
   }
   field CTS {
      access ru;
      reset 1;
   }
}
```

Example A-7 Specification for register in Figure A-1

```
source Example A-3
register CTRL {
   bytes 2;
   left_to_right;
   field CTS {
      access ru;
      reset 1;
   }
   field DTR {
      access ru;
   }
   field unused {
      bits 7;
   }
   field PAR;
   field RXE {}
```

```
   field TXE {}
}
```

Example A-8 User-defined cross-coverage point

```
register r {
    field f1 {...}
    field f2 {...}
    field f3 {...}

    cross f1 f2 {
        label xyz;
    }
    cross xyz f3;
}
```

---

regfile

A register file defines a collection of consecutive registers. Register files are used in blocks.

Syntax

```
regfile name {
   <properties>
}
```

Defines a register file with the specified name.

Properties

The following properties can be used to specify the register file.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the register file. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the register file. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block defining the register file. If a register is not mapped in the address space of the block (see "Non-linear, Non-mapped Access"), the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of registers is instantiated. Register arrays are located at consecutive address offsets, starting with register[0]. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the block, to the HDL structure implementing the register. If an `(hdl_path)` is specified, direct hierarchical access to the register can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing system and any HDL expression specified in the register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses. The `(hdl_path)` for a register array must include a `%d` placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same register file, it must be renamed to a unique name within the register file.

You must specify at least one register property.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints on the value of the registers and fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions.

```
[shared [(hdl_path)];]
```

Specifies that this register file is physically shared by all domains in a block that instantiates it. This property can only be used in a stand-alone register file specification.

The `(hdl_path)` specifies the hierarchical access path to the shared register file. For shared register files, this `(hdl_path)` is used, instead of the `(hdl_path)` specified, if any, while instantiating the register file in a block. If an `(hdl_path)` is specified, direct hierarchical access to the shared register file can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing block. The `(hdl_path)` must be enclosed in parentheses.

All the registers instantiated inside a shared register file must also be shared.

```
[cover <+|- a|b|f>
```

Specifies if the registers in this register file are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the register file, using HTML formatting tags. Currently unsupported.

Example

The primary purpose of register files is to define arrays of groups of registers. For example, the register group illustrated in Figure A-2 is used to configure a DMA channel. The block RALF specification shown in Example A-9 illustrates how a 16-channel DMA controller might be described.

Figure A-2 DMC Channel Configuration Registers Specification

Source Address				
Destination Address				
Word Count				
Status	DN	Unused	BSY	TXE
15	12	2	1	0

Example A-9 Specification for multi-channel DMA controller

```
block dma_ctrl {
   regfile chan[16] {
      register src {
         bytes 2;
         field addr {
            bits 16;
         }
      }
```

```
      register dst {
         bytes 2;
         field addr {
            bits 16;
         }
      }
      register count {
         bytes 2;
         field n_bytes {
            bits 16;
         }
      }
      register ctrl {
         bytes 2;
         field TXE {
            bits 1;
            access a0;
         }
         field BSY {
            bits 1;
            access ro;
         }
         field DN @12 {
            bits 1;
            access ro;
         }
         field status {
            bits 3;
            access ro;
         }
      }
   }
}
```

memory

A memory defines a region of consecutively addressable locations. Memories are used in blocks.

Syntax

`memory` *`name`* `{`

```
   <property>
}
```

Defines a memory with the specified name.

Properties

The following properties can be used to specify the memory.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specificied by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

`size` *m*[k|M|G]`;`

Specifies the number of consecutive addresses in the memory where each location has the number of bits specified by the *`bits`* property. The size may also include a unit. In that case, the specified size is multiplied by:

- 1024 (k)
- 2^20 (M)
- 2^30 (G)

This property is required.

`bits` *`n`*`;`

Specifies the number of bits in each memory location. The total number of bits in the memory is the specified number of bits multiplied by the specified size. This property is required.

`[access rw|ro;]`

Specifies if the memory is a RAM (`rw`) or a ROM (`ro`). By default, a memory is a RAM.

`[initial x|0|1|addr|`*`literal`*`[++|--];]`

Specifies the initial content of the memory is to be filled with unknowns (`x`), filled with zeroes (`0`), filled with ones (`1`), set to the physical address value (`addr`), or set to a constant (*`literal`*), incrementing (*`literal`*`++`) or decrementing (*`literal`*`--`) literal value.

The content of the memory is initialized to the specified pattern when the `vmm_ral_mem::initialize()` method in its abstraction class is invoked. By default, a memory is initialized with unknowns (`x`).

Initialization requires that backdoor access to the memory content be available.

`[noise ro|rw|unused|no;]`

Specifies if and how this memory can be accessed during normal operations of the design without effecting the configuration of the device. By default, a memory can be read at any time (`ro`). If `rw` is specified, this memory can also be written. If `unused` is specified, only unused memory locations can be read and written. If `no` is specified, this memory cannot be accessed in any way during normal operations. Currently unsupported.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this memory is physically shared by all domains in a block that instantiates it. Can only be used in a standalone memory specification.

The optional (`hdl_path`) specifies the hierarchical access path to the physical memory. It is used in lieu of the (`hdl_path`) specified in the block instantiating it. If an (`hdl_path`) is specified, direct hierarchical access to the shared memory can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing block. The (`hdl_path`) must be enclosed between parentheses.

```
[cover <+|- a>
```

Specifies if this memory is to be included (+a) in or excluded (-a) from the address map coverage model.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the memory, using HTML formatting tags.

Example

Example A-10 64 KB RAM

```
memory dma_bfr {
   bits 8;
   size  64k;
}
```

Example A-11 2 KB ROM

```
memory tx_bfr {
   bits    16;
   size    1024;
   access  ro;
   initial 0++;
}
```

virtual register

A `virtual register` defines a concatenation of virtual fields. Virtual registers are used in blocks.

Syntax

```
virtual register name {
   <properties>
}
```

Defines a virtual register with the specified name.

Properties

The following properties can be used to specify the virtual register.

`[bytes n;]`

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. The actual number of memory locations used by the virtual register is the minimum integral number of memory locations required to provide the specified number of bytes. If this property is not specified, the width of the register is the minimum integral number of memory locations necessary to implement all fields contained in the register.

`[left_to_right;]`

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of, and spacing between, the other fields.

```
[<field name[=rename] [@bit_offset];
[<field name [@bit_offset] {
      bits n;
      [doc {
         <text>
      }]
   }>]
```

Defines and instantiates the specified virtual field with the specified number of bits in this virtual register. The first form specifies an instance of a previously-defined field description where only the `bits` property is considered (all other properties are ignored). The second form defines a new field description and instantiates it in the register file.

Please refer to the specification of the `field` property in the "register" construct for more details on how they are physically laid out.

At least one `field` property must be specified.

All bits in a virtual register, including unused and reserved bits have their access modes defined by the access mode of the underlying memory used to implement it and the domain used to access them.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the register/field, using HTML formatting tags. Currently unsupported.

block

A block defines a set of registers and memories. Registers are concatenated into blocks. A block can have more than one physical interface. Registers and memories can be shared across physical interfaces within a block.

Syntax

```
block name {
   <property>
}
```

Specifies a design block with the specified name and a single physical interface.

```
block name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a design block with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the block specification can have a `doc` property outside of the `domain` specification.

The name of the block is used to generate block-specific unique identifiers.

Properties

The following properties can be used to specify the block and its domains.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specificied by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

```
bytes n;
```

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

```
[endian little|big|fifo_ls|fifo_ms;]
```

Specifies how wider registers and memories are mapped onto multiple accesses over the physical interface. See "Hierarchical Descriptions and Composition" for a description of the various mapping modes. By default, little endian is used.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [+incr] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the block. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the block. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block. If a register is not mapped in the address space of the block (see “Non-linear, Non-mapped Access”), the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of register is instantiated. Register arrays are located at consecutive address offsets, starting with register [0]. If an increment value is specified, the offset of each register in the register array is incremented by the specified increment. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the block, to the HDL structure implementing the register. If an `(hdl_path)` is specified, direct hierarchical access to the register is automatically generated by concatenating it with the `(hdl_path)` of the enclosing system and any `(hdl_path)` expression specified in the register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses. The `(hdl_path)` for a register array must include a "`%d`" placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

You must specify at least one register or memory property.

Registers must have unique addresses, therefore, it is not possible to describe a block containing a read-only register and a write-only register sharing the same physical address. If it is not possible to avoid this implementation structure, specify a single register with a field of `other` bits.

```
[<regfile name[=rename][[n]] [(hdl_path)] [@offset]
   [+incr] [read|write];>]
```

```
[<regfile name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register file description. The second form defines a new register file description and instantiates it in the block. An inlined register file description cannot contain the shared property. Access to a shared register file can be further restricted to read or write in a particular instance, which would essentially apply this restriction to all shared registers contained inside that shared register file.

If a numerical index is specified, an array of register files is instantiated. Register file arrays are located at consecutive address offsets, starting with register [0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a `regfile` array. Instantiating an array of register files is logically equivalent to explicitly instantiating all of the individual register files explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

Register files are usually used to specify arrays of register groups. Arrays of register files yield a different address map than register arrays. See “Arrays and Register Files” for more details.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the register file. Direct hierarchical access to the register file can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of the enclosing system and the (`hdl_path`) specified in the registers. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a register file array must include a "`%d`" placeholder that will be replaced with the decimal index of the register file in the array.

```
[<memory name[=rename] [(hdl_path)] [@offset]
       [read|write];>]
```

```
[<memory name [(hdl_path)] [@offset] {
      <property>
   }>]
```

The first form specifies an instance of a previously-defined memory description. The second form defines a new memory description and instantiates it in the block. An inlined memory description cannot contain the `shared` property. The access to a shared memory can be further restricted to read or write in a particular instance.

A memory may be instantiated at an explicit address offset within the block. If not specified, the memory is instantiated at the next available address, starting with 0. The number of addresses occupied by a memory depends on the size and width of the memory, and the endian property of the block. If a memory is not mapped in the address space of the block (see “Non-linear, Non-mapped Access”), the offset may be specified as `@none` to indicate that the memory does not consume any address locations.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the memory. If an (`hdl_path`) is specified, direct hierarchical access to the memory can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing system. The (`hdl_path`) must be enclosed between parentheses.

If more than one memory with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

At least one register or memory property must be specified.

```
[<virtual register name [=rename[n] mem@offset
   [+incr]];>]
```

```
[<virtual register name[[n] mem@offset [+incr]] {
      <property>
   }]
```

The first form instantiates an array of a previously-defined virtual register description in the block. The second form instantiates an array of a new virtual register description.

If a memory association is specified, the array of virtual register is statically implemented in the specified memory starting at the specified offset. If an increment value is specified, the implementation offset of each virtual register in the virtual register array is incremented by the specified increment. If a memory association is not specified, the virtual register is still instantiated in the block but must be dynamically associated with an implementation memory using the “vmm_ral_vreg::implement()” or “vmm_ral_vreg::allocate()” method before it can be used.

If more than one array of virtual registers with the same name is associated in the same block, it must be renamed to a unique name within the memory.

Because virtual registers are implemented in memory, it is possible to describe overlapping virtual register arrays.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints used when the content of the registers in the block is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

Constraints cannot be used to constrain the content of memories or virtual registers.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the block or domain, using HTML formatting tags.

Example

Example A-12 Block With Single Physical Interface

```
source Example A-7
source Example A-11
block uart {
   bytes 1;
   endian little;
   register CTRL;
   memory tx_bfr @'h00100;
}
```

Example A-13 Block With Register Array

```
block multi_chan {
   bytes 1;
   endian little;
   register CHAN_CTRL[32] @'h0200 {
      bytes 2;
      ...
   };
}
```

Example A-14 Block With Two Physical Interfaces

```
register data_xfer {
   bytes 4;
   field data {
      bits 32;
   }
   shared;
}
register flags {
   field cts {
      access ru;
      reset 1;
   }
   field dtr {
      access ru;
   }
}
block bridge {
   domain pci {
     bytes 4;
     register pci_flags;
     register data_xfer=to_ahb write;
```

```
     register data_xfer=frm_ahb read;
   }
   domain ahb {
      bytes 4;
      register ahb_flags;
      register data_xfer=to_pci write;
      register data_xfer=frm_pci read;
   }
}
```

system

A `system` defines a design composed of blocks or subsystems. A system can be used to create larger systems.

Syntax

```
system name {
   <property>
}
```

Specifies a system with the specified name and a single physical interface.

```
system name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a system with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the `system` specification can have a `doc` property outside of the `domain` specification.

The name of the system is used to generate system-specific unique identifiers.

Properties

The following properties can be used to specify the system and its domains.

```
[attributes {
      <name> <value>[, ...]
   }]
```

Specifies a value for the specified user-defined attribute. Multiple attributes may be specificied by separating each attribute-value pair with a comma. If the value contains white spaces, it must be included between double quotes.

```
bytes n;
```

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

```
[endian little|big|fifo_ls|fifo_ms;]
```

Specifies how wider blocks and subsystems are mapped onto multiple accesses over the physical interface. See "Hierarchical Descriptions and Composition" for a description of the various mapping modes. By default, little endian is used.

```
[<block name[[.domain]=rename][[n]] [(hdl_path)]
            @offset [+incr];>]
```

```
[<block name[[n]] [(hdl_path)] @offset [+incr] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined block description. The second form defines a new block description and instantiates it in the system.

A block must be instantiated at an explicit address offset within the system. If the base address of the block is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of blocks is instantiated. Block arrays are located at consecutive address offsets, starting with block[0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a block array. Instantiating an array of blocks is logically equivalent to explicitly instantiating all of the individual blocks explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the system, to the HDL structure implementing the block. Direct hierarchical access to the registers and memories in the block can be automatically generated by concatenating the specified `(hdl_path)` with the `(hdl_path)` of any enclosing system and the `(hdl_path)` to the registers and memories within the block. The `(hdl_path)` must be enclosed between parentheses. The `(hdl_path)` for a block array must include a "`%d`" placeholder that will be replaced with the decimal index of the block in the array.

If more than one block with the same name is instantiated in the same block, it must be renamed to a unique name within the block. A reference to a domain within a block uses a composite name and must be renamed to a single name that is a valid SystemVerilog or OpenVera user-defined identifier.

At least one `block` or `system` property must be specified.

```
[<system name[[.domain]=rename][[n]] [(hdl_path)]
            @offset [+incr];>]
[<system name[[n]] [(hdl_path)] @offset [+incr] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined subsystem description. The second form defines a new subsystem description and instantiates it in the system.

A subsystem must be instantiated at an explicit address offset within the system. If the base address of the subsystem is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of subsystems is instantiated. Subsystem arrays are located at consecutive address offsets, starting with `subsys[0]`, separated by the specified offset increment. The offset increment is required and only valid when instantiating a subsystem array. Instantiating an array of subsystems is logically equivalent to explicitly instantiating all of the individual subsystems explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the system, to the HDL structure implementing the subsystem. Direct hierarchical access to the registers and memories in the subsystem can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of any enclosing system and the (`hdl_path`) to the registers and memories within the subsystem. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a subsystem array must include a "`%d`" placeholder that will be replaced with the decimal index of the subsystem in the array.

If more than one subsystem with the same name is instantiated in the same system, it must be renamed to a unique name within the system. A reference to a domain within a subsystem uses a composite name and must be renamed to a single name that is a valid SystemVerilog or Openvera user-defined identifier.

At least one *block* or *system* property must be specified.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints used when the content of the registers and memories in the system is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the system or domain, using HTML formatting tags. Currently unsupported.

Example

Example A-15 System With Single Physical Interface

```
source Example A-12
system SoC {
   bytes 1;
   endian little;
   block uart[2] @'hF0000 +'h01000;
}
```

Example A-16 System With Two Physical Interfaces

```
source Example A-12
source Example A-14
system SoC {
   domain ahb {
      bytes 4;
      block uart[2] @'hF0000 +'h01000;
      block bridge.ahb=br @0;
   }
   domain pci {
      bytes 4;
      block bridge.pci=br @0;
   }
}
```

B

RAL Classes

This appendix provides detailed documentation of the classes that compose a RAL model. All RAL classes generated from a user specification are extended from one of the RAL base classes. Extensions do not add any methods, therefore, the documentation of the base class is the same as the documentation for the generated class.

The OpenVera and SystemVerilog classes have identical functionality and features, therefore, they are documented together. The heading used to introduce a method uses the SystemVerilog name. The OpenVera name will be identical, except for a few instances where a `_t` suffix is appended to indicate that it may be a blocking method.

Usage examples are usually specified in a single language but that should not deter users of the other language as they would be almost identical. This document prefers to provide more different examples than almost identical examples in each language.

This appendix documents the classes in alphabetical order, and documents methods in each class in a logical order, where methods that accomplish similar results are documented sequentially. Additionally, this appendix provides a summary of all available methods with cross references to their detailed documentation at the beginning of each class specification.

This section contains the following topic:

- "RAL Classes Based on vmm_object"

RAL Classes Based on vmm_object

All `vmm_ral` classes are based on `vmm_object`, since they do not need to be phased. The top-level `vmm_ral_block_or_sys` object instances are specified as the roots of the RAL namespace.

All of the following methods have two additional default arguments. The arguments `(string fname = "", int lineno = 0)` are added to these methods and are used in any error message issued by these methods:

```
vmm_mam::reserve_region();
vmm_mam::request_region();
vmm_mam_region::read();
vmm_mam_region::write();
vmm_mam_region::burst_read();
vmm_mam_region::burst_write();
vmm_mam_region::peek();
vmm_mam_region::poke();
vmm_ral_access::read();
vmm_ral_access::write();
vmm_ral_access::burst_read();
vmm_ral_access::burst_write();
vmm_ral_access::read_by_name();
vmm_ral_access::write_by_name();
```

```
vmm_ral_access::read_mem_by_name();
vmm_ral_access::write_mem_by_name();
vmm_ral_block_or_sys::update();
vmm_ral_block_or_sys::mirror();
vmm_ral_field/mem/reg/vfield/vreg::read();
vmm_ral_field/mem/reg/vfield/vreg::write();
vmm_ral_field/mem/reg/vfield/vreg::peek();
vmm_ral_field/mem/reg/vfield/vreg::poke();
vmm_ral_field/mem/reg::mirror();
vmm_ral_field/mem/reg::append_callback();
vmm_ral_field/mem/reg::prepend_callback();
vmm_ral_mem::burst_read();
vmm_ral_mem::burst_write();
vmm_ral_mem/reg::set_frontdoor();
vmm_ral_mem/reg::set_backdoor();
vmm_ral_field/reg::predict();
vmm_ral_field/reg::set();
vmm_ral_field/reg::get();
vmm_ral_mem/reg_frontdoor::read();
vmm_ral_mem/reg_frontdoor::write();
vmm_ral_mem_frontdoor::burst_read();
vmm_ral_mem_frontdoor::burst_write();
```

Example

```
ral_sys_dut ral_model = new;
vmm_log log = new("prog_blk","test");
vmm_ral_access acc = new();
vmm_ral_reg regs[];
vmm_ral_block_or_sys my_ral;
vmm_rw::status_e status;
initial begin
   acc.set_model(ral_model);
    ral_model.b[1].update(status, vmm_ral::DEFAULT,
      `__FILE__, `__LINE__);
```

RAL Class Summary

- vmm_mam ..... page B-5
- vmm_mam_allocator ..... page B-27
- vmm_mam_cfg ..... page B-38
- vmm_mam_region ..... page B-49
- vmm_ral ..... page B-72
- vmm_ral_access ..... page B-86
- vmm_ral_block_or_sys ..... page B-114
- vmm_ral_block ..... page B-182
- vmm_ral_env ..... page B-186
- vmm_ral_field ..... page B-195
- vmm_ral_field_callbacks ..... page B-235
- vmm_ral_mem ..... page B-244
- vmm_ral_mem_backdoor ..... page B-301
- vmm_ral_mem_burst ..... page B-312
- vmm_ral_mem_callbacks ..... page B-319
- vmm_ral_mem_backdoor::prepend_callback() ..... page B-330
- vmm_ral_reg ..... page B-350
- vmm_ral_reg_backdoor ..... page B-402
- vmm_ral_reg_backdoor::pre_write() ..... page B-407
- vmm_ral_reg_frontdoor ..... page B-436
- vmm_ral_sys ..... page B-441
- vmm_ral_vfield ..... page B-454
- vmm_ral_vfield_callbacks ..... page B-476
- vmm_ral_vreg ..... page B-485
- vmm_ral_vreg_callbacks ..... page B-521
- vmm_ral_tests ..... page B-530
- vmm_rw ..... page B-537
- vmm_rw_access ..... page B-542
- vmm_rw_burst ..... page B-551
- vmm_rw_xactor ..... page B-560
- vmm_rw_xactor_callbacks ..... page B-574

vmm_mam

This class is a memory allocation management utility class similar to C's `malloc()` and `free()`. A single instance of this class is used to manage a single, contiguous address space. This memory allocation management class is used by any application-level process that requires reserved space in the memory. The section of memory (called a region) will remain reserved until it is explicitly released.

Summary

- vmm_mam::new() ..... page B-6
- vmm_mam::log ..... page B-8
- vmm_mam::default_alloc ..... page B-10
- vmm_mam::reconfigure() ..... page B-12
- vmm_mam::reserve_region() ..... page B-14
- vmm_mam::request_region() ..... page B-16
- vmm_mam::release_region() ..... page B-18
- vmm_mam::release_all_regions() ..... page B-20
- vmm_mam::psdisplay() ..... page B-22
- vmm_mam::for_each() ..... page B-23
- vmm_mam::get_memory() ..... page B-25

vmm_mam::new()

Create a new instance of a memory allocation manager.

SystemVerilog

```
function new(string name,
   vmm_mam_cfg cfg,
   vmm_ral_mem mem = null);
```

OpenVera

```
task new(string name,
   vmm_mam_cfg cfg,
   vmm_ral_mem mem = null);
```

Description

Create an instance of a memory allocation manager with the specified name. This instance manages all memory region allocation within the address range specified in the configuration descriptor.

If a reference to a RAL memory abstraction class is provided, the memory locations within the regions can be accessed through the region descriptor, using the Xref and Xref methods.

The specified name is used as the instance name of the message interface found in the `"vmm_mam::log"` class property.

Example

Example B-1

```
program mem_test;
```

```
ral_mam_cfg ral_mam_cfg_inst;
vmm_mam ral_mam_inst;
vmm_ral_mem mem;
   ...
   initial
      begin
         ...
         if(ral_mam_cfg_inst.randomize())
         begin
            //Instantiating MAM
            ral_mam_inst = new("RAL memory manager",
               ral_mam_cfg_inst,mem);
         end
      else
      ...
   end

endprogram
```

vmm_mam::log

Message service interface for the memory allocation manager.

SystemVerilog

```
vmm_log log;
```

OpenVera

```
rvm_log log;
```

Description

Message service interface used to issue messages from this memory allocation manager. The name of the interface is hard-coded as "Memory Allocation Manager". The instance name of the interface is the name of the manager specified in the constructor. These names may be modified afterward using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

Example

Example B-2

```
program mem_test;

ral_mam_cfg ral_mam_cfg_inst;
vmm_mam ral_mam_inst;
vmm_ral_mem mem;

   initial
      begin
         ...  //Build and randomization of ral_mam_cfg_inst
          //Instantiating ral_mam_inst
```

```
         ral_mam_inst = new("RAL memory manager",
            ral_mam_cfg_inst,mem);
         //Setting the name of a ral_mam_inst's log
      ral_mam_inst.log.set_name("RAL memory manager log");
         ...
      end

endprogram
```

vmm_mam::default_alloc

Default memory region allocator.

SystemVerilog

```
vmm_mam_allocator default_alloc;
```

OpenVera

```
vmm_mam_allocator default_alloc;
```

Description

Default object instance that is randomized in the `request_region()` method when allocating a region at a random address and no allocator object is specified.

Example

Example B-3

```
class my_mam_allocator extends vmm_mam_allocator;
   ...
   constraint my_alloc_cons{
                            len == `MAX_LENGTH;
                            min_offset == `MIN_OFFSET;
                            max_offset == `MAX_OFFSET;
                           }
endclass : my_mam_allocator

class tb_env extends vmm_ral_env;
   ...
  ral_mam_cfg mam_cfg;  //Instance of RAL MAM configuration
      //object
  vmm_mam_region mam_reg;
  vmm_mam mam;
```

```
  my_mam_allocator my_alloc;
  vmm_ral_mem mem;    //Memory to be used by MAM
  ...
  function new();
     super.new();
     ...
  endfunction

   virtual function void build();
      super.build();
      ...
      //Instantiating MAM
      mam = new("my memory manager",mam_cfg,mem);
      ...
      if(!my_alloc.randomize())
       `vmm_fatal(log,"my_alloc randomization failed, check
          constraints");
      else
         begin
         //Setting my_alloc as the default allocator of mam
           mam.default_alloc = my_alloc;
         end

      //All the request_region calls on mam will use my_alloc
      //now
      mam_reg    = mam.request_region(mam_cfg.n_bytes);
   endfunction
   ...
endclass
```

vmm_mam::reconfigure()

Reconfigure the managed address space.

SystemVerilog

```
function vmm_mam_cfg reconfigure(vmm_mam_cfg cfg = null);
```

OpenVera

```
function vmm_mam_cfg reconfigure(vmm_mam_cfg cfg = null);
```

Description

Optionally modify the maximum and minimum addresses of the address space managed by the allocation manager, allocation mode, or locality. The number of bytes per memory location cannot be modified once an allocation manager has been constructed.

Returns the previous configuration.

All currently allocated regions must fall within the new address space.

Example

Example B-4

```
class ral_mam_cfg extends vmm_mam_cfg;
   constraint mam_cfg_const {
      end_offset == 64'h7f;
      n_bytes    == 8;
   }
endclass

class tb_env extends vmm_ral_env;
```

```
   ...
   vmm_mam mam;
   ral_mam_cfg mam_cfg;
   ...
   virtual function void build();
     super.build();
     ...
     if(!this.mam_cfg.randomize() with {start_offset ==
        `START_OFFSET_A;})
        begin
         `vmm_fatal(log,$psprintf("Failed to randomize RAL
            mem cfg for %0h", `START_OFFSET_A));
        end
     else
        mam = new("my memory manager",mam_cfg,mem);
     ...

     if(!this.mam_cfg.randomize() with {start_offset ==
        `START_OFFSET_B;})
        begin
            `vmm_fatal(log,$psprintf("Failed to randomize
               RAL mem cfg for %0h", `START_OFFSET_B));
        end
     else
        mam_cfg = mam.reconfigure(this.mam_cfg);
     ...
   endfunction : build
endclass
```

vmm_mam::reserve_region()

Reserve a specific memory region.

SystemVerilog

```
function vmm_mam_region reserve_region(bit [63:0]
start_offset,
     int unsigned n_bytes);
```

OpenVera

```
function vmm_mam_region reserve_region(bit [63:0]
start_offset,
     integer    n_bytes);
```

Description

Reserve a memory buffer of the specified number of bytes starting at the specified offset in the memory. A descriptor of the reserved region is returned. If the specified region cannot be reserved, *`null`* is returned.

It may not be possible to reserve a region because it overlaps with an already-allocated region or it lies outside the address range managed by the memory manager.

Example

Example B-5

```
class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
      super.build();
      ...
      mam_region =
```

```
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
      if(mam_region == null)
        begin
        `vmm_fatal(log,$psprintf({"Failed to reserve Memory
           Location from ","%0h location."},
           mam_cfg.start_offset));
        end
      ...
   endfunction
   ...
endclass
```

vmm_mam::request_region()

Request a memory region.

SystemVerilog

```
function vmm_mam_region request_region(
   int unsigned n_bytes, vmm_mam_allocator alloc = null);
```

OpenVera

```
function vmm_mam_region request_region(integer n_bytes,
    vmm_mam_allocator alloct = null);
```

Description

Request and reserve a memory buffer of the specified number of bytes starting at a random location in the memory. If an allocator is specified, it is randomized to determine the start offset of the region. If no allocator is specified, the allocator found in the "vmm_mam::default_alloc" class property is randomized.

A descriptor of the allocated region is returned. If no region can be allocated, `null` is returned. It may not be possible to allocate a region because there is no area in the memory with enough consecutive locations to meet the size requirements or because there is another contradiction when randomizing the allocator.

If the memory allocation is configured to `vmm_mam::THRIFTY` or `vmm_mam::NEARBY` (see the "vmm_mam_cfg::mode" and "vmm_mam_cfg::locality" class properties, respectively), a suitable region is first sought procedurally. If no suitable region is

found, then the allocator is randomized. The allocator is immediately randomized when the memory allocation is configured as `vmm_mam::BROAD` and `vmm_mam::GREEDY`.

Note: The default value of `vmm_mam_cfg::locality` is `BROAD` and the default value of `vmm_mam_cfg::mode` is `THRIFTY`. However, the `NEARBY` and `GREEDY` modes are not implemented.

Example

Example B-6

```
program test_regions;

ral_mam_cfg mam_cfg;
vmm_ral_mem mem;
vmm_mam mam;
vmm_mam_region mam_reg;

   ...
   mam = new("my memory manager",mam_cfg,mem);
   mam_reg    = mam.request_region(mam_cfg.n_bytes);
   if(mam_reg == null)
      begin
        `vmm_fatal(log,$psprintf("Failed to request Memory
           Location for %d bytes.", mam_cfg.n_bytes));
      end
   else
   begin
     ...
   end
   ...
endprogram
```

vmm_mam::release_region()

Release a previously allocated memory region.

SystemVerilog

```
function void release_region(vmm_mam_region region);
```

OpenVera

```
task release_region(vmm_mam_region region);
```

Description

Release the specified previously allocated memory region. An error is issued if the specified region has not been previously allocated or is no longer allocated.

Example

Example B-7

```
program test;

ral_mam_cfg ral_mam_cfg_inst,mam_cfg;
vmm_ral_mem mem;
vmm_mam mam;
vmm_mam_region mam_reg;

 ...
 mam = new("RAL memory manager",ral_mam_cfg_inst,mem);
 mam_reg = mam.request_region(256);
 ...
 //Releasing a region before next reconfiguration and
 //read/write operaions
 mam.release_region(mam_reg);
 if(!this.ral_mam_cfg_inst.randomize() with
```

```
      {start_offset == `START_OFFSET_A;})
    begin
      mam_cfg = mam.reconfigure(ral_mam_cfg_inst);
      mam_reg = mam.request_region(256);
      ...
    end
endprogram
```

vmm_mam::release_all_regions()

Release all allocated memory regions.

SystemVerilog

```
function void release_all_regions();
```

OpenVera

```
task release_all_regions();
```

Description

Release all allocated memory regions.

Example

Example B-8

```
program test;

ral_mam_cfg ral_mam_cfg_inst,mam_cfg;
vmm_ral_mem mem;
vmm_mam mam;

   ...
   mam = new("RAL memory manager",ral_mam_cfg_inst,mem);
   mam_reg    = mam.request_region(256);
   ...
  //Releasing all the regions before reconfiguring the MAM
   mam.release_all_regions();
     if(!this.ral_mam_cfg_inst.randomize() with
        {start_offset == `START_OFFSET_A;})
   begin
      mam_cfg = mam.reconfigure(ral_mam_cfg_inst);
      ...
   end
```

```
endprogram
```

vmm_mam::psdisplay()

Human-readable description of allocated memory regions.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

```
function string psdisplay(string prefix = "");
```

Description

Create a human-readable description of the state of the memory manager and the currently allocated regions. Each line of the description is prefixed with the specified prefix.

Example

Example B-9

```
...
mam_region =
   mam.reserve_region(mam_cfg.start_offset,
      mam_cfg.n_bytes);
//Display Allocated Memory [Starting Location:
   Ending Location]
`vmm_note(log,$psprintf("%s",mam.psdisplay("Allocated
   Memory")));
   ...
```

vmm_mam::for_each()

Iterate over allocated memory regions.

SystemVerilog

```
function vmm_mam_region for_each(bit reset = 0);
```

OpenVera

```
function vmm_mam_region for_each(bit reset = 0);
```

Description

Iterate over all currently allocated regions. If *reset* is non-zero, reset the iterator and return the first allocated region. Returns `null` when there are no additional allocated regions to iterate on.

Example

Example B-10

```
...
vmm_mam ral_mam_inst;
int mem_locations;

while(1)
   begin
      //Iterating on the memory regions in ral_mam_inst
      mam_region = ral_mam_inst.for_each();
      if(mam_region != null)
      begin
         //Checking the number of memory locations in each
         //region
         mem_locations = mam_region.get_n_bytes();
         `vmm_note(log, $psprintf({"Number of memory
            locations in this region", " are %0d"},
```

```
            mem_locations));
      end
      else
      begin
        `vmm_note(log, "No additional memory regions");
         break;
      end
   end
...
```

vmm_mam::get_memory()

Return the RAL abstraction class for the managed memory.

SystemVerilog

```
function vmm_ral_mem get_memory();
```

Description

Return the reference to the RAL memory abstraction class for the memory implementing the locations managed by this instance of the allocation manager. Returns `null` if no memory abstraction class was specified at construction time.

Example

Example B-11

```
...
//Accessing the memory associated with MAM instance
ral_mam_inst
   eth_ral_mem = ral_mam_inst.get_memory();
   if(eth_ral_mem != null)
      begin
      //Displaying the size of a memory
      `vmm_note(log,$psprintf("Size of memory is %0d",
            eth_ral_mem.get_size()));
     if(eth_ral_mem.get_access()== vmm_ral::RW) //Checking
                                  //the access on memory
       begin
          `vmm_note(log,"Memory has RW access");
          ...
         //Write operation to some location in memory
      end
      else if(eth_ral_mem.get_access() == vmm_ral::RO)
      `vmm_warning(log,"Memory has only RO access");
    end
```

```
 else
    begin
       `vmm_warning(log,"no memory class was specified at
          construction time");
    end
...
```

vmm_mam_allocator

An instance of this class is randomized to determine the starting offset of a randomly allocated memory region. This class can be extended to provide additional constraints on the starting offset, such as word alignment or location of the region within a memory page.

Summary

- vmm_mam_allocator::len ..... page B-28
- vmm_mam_allocator::min_offset ..... page B-30
- vmm_mam_allocator::max_offset ..... page B-32
- vmm_mam_allocator::in_use[$] ..... page B-34
- vmm_mam_allocator::start_offset ..... page B-36

vmm_mam_allocator::len

Number of addressable locations required.

SystemVerilog

```
int unsigned len;
```

OpenVera

```
integer len;
```

Description

Set by the memory manager before every randomization. Specifies the number of memory locations (not necessarily bytes) required in the region.

Example

Example B-12

```
class my_mam_allocator extends vmm_mam_allocator;
  ...
  constraint my_alloc_cons{
                           len <= `MAX_LENGTH;
                           min_offset == `MIN_OFFSET;
                           max_offset == `MAX_OFFSET;
                          }
endclass : my_mam_allocator

class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
      super.build();
      ...
      //Instantiating MAM
```

```
      mam = new("my memory manager",mam_cfg,mem);

      if(!my_alloc.randomize() with {len == 100;})
        `vmm_fatal(log,"my_alloc randomization failed,
           check constraints");
      else
        begin
         //Setting my_alloc as default allocator of mam
         mam.default_alloc = my_alloc;
         `vmm_note(log,$psprintf(
            "vmm_mam_allocator::len ==> %0d",
            mam.default_alloc.len));
       end
       ...
      mam_reg    = mam.request_region(mam_cfg.n_bytes);
   endfunction
   ...
endclass
```

vmm_mam_allocator::min_offset

Minimum address offset under management.

SystemVerilog

```
bit [63:0] min_offset;
```

OpenVera

```
bit [63:0] min_offset;
```

Description

Set by the memory manager before every randomization. Specifies the minimum address offset in the memory under management. The `vmm_mam_allocator_valid` constraint block constrains the "vmm_mam_allocator::start_offset" class property to fall within the range defined by this property and the "vmm_mam_allocator::max_offset" property—minus the required number of memory locations.

Example

Example B-13

```
class my_mam_allocator extends vmm_mam_allocator;
  ...
  constraint my_alloc_cons{
                           len == 100;
                           max_offset == `MAX_OFFSET;
                          }
endclass : my_mam_allocator

class tb_env extends vmm_ral_env;
   ...
```

```
   virtual function void build();
      super.build();
      ...
      //Instantiating MAM
      mam = new("my memory manager",mam_cfg,mem);

      if(!my_alloc.randomize() with {min_offset ==
         `MIN_OFFSET;})
      `vmm_fatal(log,"my_alloc randomization failed, check
         constraints");
      else
        begin
         //Setting my_alloc as default allocator of mam
         mam.default_alloc = my_alloc;
           `vmm_note(log,$psprintf("vmm_mam_allocator::
              min_offset ==> %0h",
              mam.default_alloc.min_offset));
        end
       ...
      mam_reg    = mam.request_region(mam_cfg.n_bytes);
   endfunction
   ...
endclass
```

vmm_mam_allocator::max_offset

Maximum address offset under management.

SystemVerilog

```
bit [63:0] max_offset;
```

OpenVera

```
bit [63:0] max_offset;
```

Description

Set by the memory manager before every randomization. Specifies the maximum address offset in the memory under management. The `vmm_mam_allocator_valid` constraint block constrains the "vmm_mam_allocator::start_offset" class property to fall within the range defined by this property and the "vmm_mam_allocator::min_offset" property—minus the required number of memory locations.

Example

Example B-14

```
class my_mam_allocator extends vmm_mam_allocator;
  ...
  constraint my_alloc_cons{
                           len == 100;
                           min_offset == `MIN_OFFSET;
                          }
endclass : my_mam_allocator

class tb_env extends vmm_ral_env;
   ...
```

```
   virtual function void build();
      super.build();
      ...
      //Instantiating MAM
      mam = new("my memory manager",mam_cfg,mem);

      if(!my_alloc.randomize() with {max_offset ==
         `MAX_OFFSET;})
      `vmm_fatal(log,"my_alloc randomization failed, check
          constraints");
      else
        begin
         //Setting my_alloc as default allocator of mam
          mam.default_alloc = my_alloc;
         `vmm_note(log,$psprintf("vmm_mam_allocator::
            max_offset ==> %0h",
               mam.default_alloc.max_offset));
        end
       ...
      mam_reg    = mam.request_region(mam_cfg.n_bytes);
   endfunction
   ...
endclass
```

vmm_mam_allocator::in_use[$]

Currently allocated memory regions.

SystemVerilog

```
vmm_mam_region in_use[$];
```

OpenVera

```
vmm_mam_region in_use[$];
```

Description

Set by the memory manager before every randomization. Specifies the memory regions currently allocated. The *vmm_mam_allocator_no_overlap* constraint block constrains the "vmm_mam_allocator::start_offset" class property—minus the required number of memory locations—to fall outside of the allocated regions.

Example

Example B-15

```
class tb_env extends vmm_ral_env;
  ...
  int total_rsvd_region;

  virtual function void build();
    ...
    mam = new("my memory manager",mam_cfg,mem);

    if(!my_alloc.randomize() with {start_offset ==
       `START_OFFSET;})
      `vmm_fatal(log,"my_alloc randomization failed, check
        constraints");
```

```
    else
      begin
        //Setting my_alloc as default allocator of mam
        mam.default_alloc = my_alloc;
        mam_reg    = mam.request_region(256);
        ...
        foreach(mam.default_alloc.in_use[i])
           total_rsvd_region ++;
       `vmm_note(log,$psprintf("Total Allocated(Reserved)
           Regions are ==> %0d", total_rsvd_region));
       ...
      end
  endfunction
  ...
endclass
```

vmm_mam_allocator::start_offset

Starting offset of the randomly allocated region.

SystemVerilog

```
rand bit [63:0] start_offset;
```

OpenVera

```
rand bit [63:0] start_offset;
```

Description

After a successful randomization, this class property will contain a valid starting offset for a memory region of the desired length.

Example

Example B-16

```
class tb_env extends vmm_ral_env;
  ...
  int total_rsvd_region;

  virtual function void build();
    ...
    mam = new("my memory manager",mam_cfg,mem);

    if(!my_alloc.randomize() with {start_offset ==
          `START_OFFSET;})
      `vmm_fatal(log,"my_alloc randomization failed, check
      constraints");
    else
      begin
        //Setting my_alloc as default allocator of mam
          mam.default_alloc = my_alloc;
      `vmm_note(log,$psprintf("vmm_mam_allocator::
```

```
        start_offset ==> %0h",
        mam.default_alloc.start_offset));
    end
  ...
endclass
```

vmm_mam_cfg

This class is used to specify the memory managed by an instance of a "vmm_mam" memory allocation manager class.

Summary

- vmm_mam_cfg::n_bytes ..... page B-39
- vmm_mam_cfg::start_offset ..... page B-41
- vmm_mam_cfg::end_offset ..... page B-43
- vmm_mam_cfg::mode ..... page B-45
- vmm_mam_cfg::locality ..... page B-47

vmm_mam_cfg::n_bytes

Number of bytes in each addressable location.

SystemVerilog

```
rand int unsigned n_bytes;
```

OpenVera

```
rand integer n_bytes;
```

Description

Total number of bytes in each memory location. The `vmm_mam_cfg_valid` constraint blocks constrains this class property to the {1:64} range.

Although the memory allocation manager will operate properly with any positive value for this property, it does not make much physical sense to have values that are not powers of two (for example, 1, 2, 4, 8, etc...).

Example

Example B-17

```
class ral_mam_cfg extends vmm_mam_cfg;
   constraint mam_cfg_const {
      start_offset == 64'h0;
      end_offset   == 64'h7f;
      n_bytes      <= 8;
   }
endclass

class tb_env extends vmm_ral_env;
```

```
    ...
    vmm_mam mam;
   ral_mam_cfg mam_cfg;  //Instance of RAL MAM configuration
                         //object
    ...
    virtual function void build();
      super.build();
      if (!this.mam_cfg.randomize() with {n_bytes == 8;})
        begin
          `vmm_fatal(log, "Failed to randomize Memory
             configuration");
        end
      ...
      `vmm_note(log,$psprintf("vmm_mam_cfg::n_bytes ==>
         %0d",mam_cfg.n_bytes));
      ...
    endfunction

endclass
```

vmm_mam_cfg::start_offset

Offset of the first addressable location under management.

SystemVerilog

```
rand bit [63:0] start_offset;
```

OpenVera

```
rand bit [63:0] start_offset;
```

Description

The starting offset of the consecutive memory area managed by the memory manager instance.

The `vmm_mam_cfg_valid` constraint block constrains this class property to be less than "`vmm_mam_cfg::end_offset`".

Example

Example B-18

```
class ral_mam_cfg extends vmm_mam_cfg;
   constraint mam_cfg_const {
      start_offset == 64'h0;
      end_offset   == 64'h7f;
      n_bytes      <= 8;
   }
endclass

class tb_env extends vmm_ral_env;
   ...
   vmm_mam mam;
  ral_mam_cfg mam_cfg;  //Instance of RAL MAM configuration
                        //object
```

```
   ...
   virtual function void build();
     super.build();
     if (!this.mam_cfg.randomize())
       begin
        `vmm_fatal(log, "Failed to randomize Memory
           configuration");
       end
     else
       begin
       `vmm_note(log,$psprintf("vmm_mam_cfg::start_offset
          ==> 64'h%0h", mam_cfg.start_offset));
     end
     ...
   endfunction

   ...
endclass
```

vmm_mam_cfg::end_offset

Offset of the last addressable location under management.

SystemVerilog

```
rand bit [63:0] end_offset;
```

OpenVera

```
rand bit [63:0] end_offset;
```

Description

The ending offset of the consecutive memory area managed by the memory manager instance.

The `vmm_mam_cfg_valid` constraint block constrains this class property to be greater than "`vmm_mam_cfg::start_offset`".

Example

Example B-19

```
class ral_mam_cfg extends vmm_mam_cfg;
   constraint mam_cfg_const {
      start_offset == 64'h0;
      end_offset   == 64'h7f;
      n_bytes      <= 8;
   }
endclass

class tb_env extends vmm_ral_env;
   ...
   vmm_mam mam;
  ral_mam_cfg mam_cfg;  //Instance of RAL MAM configuration
                        //object
```

```
    ...
    virtual function void build();
      super.build();
      if (!this.mam_cfg.randomize())
        begin
          `vmm_fatal(log, "Failed to randomize Memory
            configuration");
        end
      else
        begin
          `vmm_note(log,$psprintf("vmm_mam_cfg::end_offset
            ==> 64'h%0h", mam_cfg.end_offset));
        end
    endfunction
   ...
endclass
```

vmm_mam_cfg::mode

Memory allocation mode.

SystemVerilog

```
rand enum {vmm_mam::GREEDY, vmm_mam::THRIFTY} mode;
```

OpenVera

```
rand enum {vmm_mam::GREEDY, vmm_mam::THRIFTY} mode;
```

Description

Configures how new memory regions are allocated by the "`vmm_mam::request_region()`" method. If set to `vmm_mam::THRIFTY`, memory used in previously allocated, but now freed regions is preferred. If set to `vmm_mam::GREEDY`, previously unused memory is preferred.

Note:The default value of `vmm_mam_cfg::mode` is `THRIFTY`, and the mode, `GREEDY` is not implemented.

Example

Example B-20

```
class ral_mam_cfg extends vmm_mam_cfg;
   constraint mam_cfg_const {
      start_offset == 64'h0;
      end_offset   == 64'h7f;
      n_bytes      == 8;
   }
endclass

class tb_env extends vmm_ral_env;
```

```
   ...
   if (!this.mam_cfg.randomize() with {mode ==
         vmm_mam::THRIFTY})
      begin
      `vmm_fatal(log, "Failed to randomize Memory
          configuration");
      end
   else
      begin
        `vmm_note(log,$psprintf("vmm_mam_region::mode ==>
          %0s", mam_cfg.mode.name));
         mam = new("my memory manager",mam_cfg,mem);
         mam_reg = mam.request_region(10);   //Creating a
                                   //region in THRIFTY mode
      end
   ...

   if(!this.mam_cfg.randomize() with {mode ==
         vmm_mam::GREEDY})
      begin
         `vmm_fatal(log, "Failed to randomize Memory
           configuration");
      end
   else
      begin
        mam_cfg = mam.reconfigure(this.mam_cfg);
        mam_reg = mam.request_region(10); //Creating a
                                     //region in GREEDY mode
      end
 ...
endclass
```

vmm_mam_cfg::locality

Memory allocation locality.

SystemVerilog

```
rand enum {vmm_mam::BROAD, vmm_mam::NEARBY} locality;
```

OpenVera

```
rand enum {vmm_mam::BROAD, vmm_mam::NEARBY} locality;
```

Description

Configures where in memory, in relation to currently allocated regions, new memory is allocated by the "`vmm_mam::request_region()`" method. If set to `vmm_mam::NEARBY`, memory near or adjacent to currently allocated regions is preferred. If set to `vmm_mam::BROAD`, the entire memory space is used without preference.

Note: The default value of `vmm_mam_cfg::locality` is `BROAD`, and the mode, `NEARBY` is not implemented

Example

Example B-21

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize())
      begin
         `vmm_fatal(log, "Failed to randomize Memory
           configuration");
      end
   ...
```

```
   `vmm_note(log,$psprintf("vmm_mam_region::locality ==>
         %0s", mam_cfg.locality.name));
   ...
endclass
```

vmm_mam_region

This class is used by the memory allocation manager to describe allocated memory regions. Instances of this class should not be created directly, therefore, this appendix does not document the constructor. Instances of this class should be created only from within the memory manager, in the "vmm_mam::reserve_region()" and "vmm_mam::request_region()" methods.

Summary

- vmm_mam_region::get_start_offset() ..... page B-50
- vmm_mam_region::get_end_offset() ..... page B-52
- vmm_mam_region::get_len() ..... page B-53
- vmm_mam_region::get_n_bytes() ..... page B-54
- vmm_mam_region::get_memory() ..... page B-55
- vmm_mam_region::get_virtual_registers() ..... page B-56
- vmm_mam_region::psdisplay() ..... page B-57
- vmm_mam_region::read() ..... page B-58
- vmm_mam_region::write() ..... page B-60
- vmm_mam_region::burst_read() ..... page B-62
- vmm_mam_region::burst_write() ..... page B-65
- vmm_mam_region::peek() ..... page B-68
- vmm_mam_region::poke() ..... page B-70

vmm_mam_region::get_start_offset()

Return the starting offset of the allocated region.

SystemVerilog

```
function bit [63:0] get_start_offset();
```

OpenVera

```
function bit [63:0] get_start_offset();
```

Description

Return the starting offset, within the memory, of the allocated region.

Example

Example B-22

```
class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
     super.build();
     mam_region = mam.reserve_region(mam_cfg.start_offset,
          mam_cfg.n_bytes);
     if(mam_region == null)
        begin
             `vmm_fatal(log,$psprintf({"Failed to reserve
                Memory Location from ", "%0h location."},
                 mam_cfg.start_offset));
        end
    `vmm_note(log,$psprintf("vmm_mam_region::
         get_start_offset ==> %h",
         mam_region.get_start_offset()));
   ...
   endfunction
   ...
```

```
endclass
```

vmm_mam_region::get_end_offset()

Return the ending offset of the allocated region.

SystemVerilog

```
function bit [63:0] get_end_offset();
```

OpenVera

```
function bit [63:0] get_end_offset();
```

Description

Return the ending offset, within the memory, of the allocated region.

Example

Example B-23

```
class tb_env extends vmm_ral_env;
   ...
   mam_region = mam.reserve_region(mam_cfg.start_offset,
         mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
             Location from %0h ","location."},
             mam_cfg.start_offset));
     end
  `vmm_note(log,$psprintf("vmm_mam_region::get_end_offset
       ==> %h", mam_region.get_end_offset()));
   ...
endclass
```

vmm_mam_region::get_len()

Return the number of memory locations in the allocated region.

SystemVerilog

```
function int unsigned get_len();
```

OpenVera

```
function integer get_len();
```

Description

Return the number of consecutive memory locations (not necessarily bytes) in the allocated region.

Example

Example B-24

```
class tb_env extends vmm_ral_env;
   ...
   mam_region = mam.reserve_region(mam_cfg.start_offset,
         mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
              Location from %0h ", "location."},
              mam_cfg.start_offset));
     end
     `vmm_note(log,$psprintf("vmm_mam_region::get_len ==>
           %d", mam_region.get_len()));
   ...
endclass
```

vmm_mam_region::get_n_bytes()

Return the number of memory locations in the allocated region.

SystemVerilog

```
function int unsigned get_n_bytes();
```

OpenVera

```
function integer get_n_bytes();
```

Description

Return the number of consecutive bytes in the allocated region. If the managed memory contains more than one byte per address, the number of bytes in an allocated region may be greater than the number of requested or reserved bytes.

Example

Example B-25

```
class tb_env extends vmm_ral_env;
   ...
   mam_region = mam.reserve_region(mam_cfg.start_offset,
         mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
             Location from %0h ", "location."},
             mam_cfg.start_offset));
     end
  `vmm_note(log,$psprintf("vmm_mam_region::get_n_bytes ==>
        %d", mam_region.get_n_bytes()));
   ...
endclass
```

vmm_mam_region::get_memory()

Return the RAL memory abstraction class for the region.

SystemVerilog

```
function vmm_ral_mem get_memory();
```

Description

Return the reference to the RAL memory abstraction class for the memory implementing this allocated memory region. Returns `null` if no memory abstraction class was specified for the allocation manager that allocated this region.

Example

Example B-26

```
vmm_ral_mem::init_e pat;
region_mem = my_region.get_memory(); //Accessing the Memory
                     //reference associated with my_region
...
if(region_mem == null)
  begin
    `vmm_note(log,"no memory abstraction class was specified
       for my_region");
    end
else
  begin  //If memory is associated then, initializing it
      `vmm_note(log,$psprintf("Connected Memory size ==>
          %0d", region_mem.get_size()));
      region_mem.init(is_ok,pat,data); //Initialize the
                                                  //memory
  end
...
```

vmm_mam_region::get_virtual_registers()

Return the RAL abstraction class for the virtual register set.

SystemVerilog

```
function vmm_ral_vreg get_virtual_registers();
```

Description

Return the reference to the RAL virtual register abstraction class for the set of virtual registers implemented in the allocated region. Returns `null` if the memory region is not known to implement virtual registers.

Example

Example B-27

```
vmm_mam_region::get_virtual_registers() [page 208]

  vmm_ral_vreg vregs[];
  ral_model.memory_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_mam_region::psdisplay()

Human-readable description of allocated memory.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

```
function string psdisplay(string prefix = "");
```

Description

Create a human-readable description of the allocated region. Each line of the description is prefixed with the specified prefix.

Example

Example B-28

```
...
mam_region =
    mam.reserve_region(mam_cfg.start_offset,
    mam_cfg.n_bytes);
//Display Allocated Region [Min Location:Max Location]
`vmm_note(log,$psprintf("%s",
      mam_region.psdisplay("Allocated Region")));
...
```

vmm_mam_region::read()

Reads a region location from the design.

SystemVerilog

```
task read(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(
   bit [63:0]       offset,
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory region location from the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `vmm_rw_access` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-29

```
//Assume that vmm_ral_mem has already provided to my_region
...
fork
  ...
   begin
      wait(write_done);
      this.mam_region.read(status,addr,read_data);
      `vmm_note(log,$psprintf({"Read Data ==> %0h and
          Status ==> %0s", read_data,status.name));
      write_done = 1'b0;
   end
join_none
...
```

vmm_mam_region::write()

Writes a region location in the design.

SystemVerilog

```
task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(
   bit [63:0]       offset,
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer          stream_id = -1)
```

Description

Writes the specified value at the specified region location in the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `vmm_rw_access` transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-30

```
...
fork
  begin
     this.mam_region.write(status,addr,data);
     `vmm_note(log,$psprintf("Memory Status %0s and
         offset address %0h" status.name,
         mam_region.get_start_offset);
     write_done = 1'b1;
   end
   ...
join_none
...
```

vmm_mam_region::burst_read()

Perform a burst-read operation on the region.

SystemVerilog

```
task burst_read(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   output bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int               stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer           stream_id = -1)
```

Description

Burst-read the current values of the region locations specified by the burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the `vmm_ral_access::burst_read()` method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-31

```
class my_ral_mem_burst extends vmm_ral_mem_burst;
   ...
   constraint my_cons {
     start_offset == 64'h00;
     incr_offset  == 64'h5;
     max_offset   == 64'h20;
   }
endclass
...
class my_ral_mem_frontdoor extends vmm_ral_mem_frontdoor;
   ...
   virtual task burst_read(...);
      ...
      //Need to overwrite
   endtask
   ...
endclass
...
begin
   ...
   my_ral_mem_frontdoor ftdr = new;
   this.my_mem.set_frontdoor(ftdr);
```

```
   ...
end
...
my_ral_mem_burst br;
bit [63:0] data[];
...
//No.of transfers in a burst opertion as 10
if(br.randomize() with {n_beats == 10;})
  begin
    ...
    //Need to provide memory to my_region
    my_region.burst_read(status,br,data);//Performing a
                                         //burst read on my_region
    if(status == vmm_rw::IS_OK)
      begin
        `vmm_note(log,"Successful burst read operation");
        ...
      end
      ...
  end
```

vmm_mam_region::burst_write()

Perform a burst-write operation on the region.

SystemVerilog

```
task burst_write(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   input  bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int               stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer           stream_id = -1)
```

Description

Burst-write the specified values in the region locations specified by burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the `vmm_ral_access::burst_write()` method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-32

```
...
class my_ral_mem_burst extends vmm_ral_mem_burst;
   ...
   constraint my_cons {
     n_beats == 50;
     start_offset == 64'h00;
     incr_offset  == 64'h5;
     max_offset   == 64'h20;
   }
endclass

class my_ral_mem_frontdoor extends vmm_ral_mem_frontdoor;
   ...
   virtual task burst_write(...);
      ...
      //Need to overwrite
   endtask
   ...
endclass
...
my_ral_mem_burst br;
bit [63:0] data[];
```

```
...  //Provide data in data[] dymanic array

if(br.randomize())  //Randomizing the burst
  begin
    ...
    my_region.burst_write(status,br,data); //Performing a
                                           //burst write on my_region
    if(status == vmm_rw::IS_OK)
      begin
        `vmm_note(log,"Successful burst write operation");
        ...
     end
     ...
  end
...
```

vmm_mam_region::peek()

Peek a region location from the design.

SystemVerilog

```
task peek(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e peek_t(
   bit [63:0]       offset,
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer          stream_id = -1)
```

Description

Reads the current value of the region location from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-33

```
//Assume that vmm_ral_mem has already provided to my_region
...
fork
  ...
  begin
    wait(poke_done);
    this.mam_region.peek(status,addr,peek_data);
    `vmm_note(log,$psprintf({"Peek Data ==> %0h and Status
          ==> %0s", peek_data,status.name));
    poke_done = 1'b0;
  end
join_none
...
```

vmm_mam_region::poke()

Poke a region location in the design.

SystemVerilog

```
task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e poke_t(
   bit [63:0]       offset,
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value at the specified region location in the design using a back-door access. Depending on the design model implementation, it may be possible to modify the content of a read-only memory.

The optional value of the arguments:

`data_id`
`scenario_id`
`stream_id` arguments

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-34

```
...
fork
  begin
    this.mam_region.poke(status,addr,data);
    `vmm_note(log,$psprintf("Memory Status %0s and offset
          address %0h", status.name,
          mam_region.get_start_offset);
    poke_done = 1'b1;
  end
  ...
join_none
...
```

vmm_ral

Utility class for global symbolic values. Each set of symbolic values is specified using enumerated types. The symbolic values are accessed using a fully-qualified name, such as `vmm_ral::RW`.

A separate encapsulating class is used to minimize the length of these identifiers and to make them easier to share across classes.

Summary

- vmm_ral::path_e ..... page B-73
- vmm_ral::access_e ..... page B-75
- vmm_ral::check_e ..... page B-79
- vmm_ral::endianness_e ..... page B-80
- vmm_ral::reset_e ..... page B-82
- vmm_ral::coverage_e ..... page B-84

vmm_ral::path_e

Symbolic values identifying the path used to access a register or memory within the design.

SystemVerilog

```
vmm_ral::BFM
vmm_ral::BACKDOOR
vmm_ral::DEFAULT
```

OpenVera

```
vmm_ral::BFM
vmm_ral::BACKDOOR
vmm_ral::DEFAULT
```

Description

`vmm_ral::BFM`

Accesses the register or memory through the appropriate physical interface transactor. Also known as front-door access.

`vmm_ral::BACKDOOR`

Accesses the register or memory directly through an appropriate back-door mechanism, usually a hierarchical path into the design.

`vmm_ral::DEFAULT`

Accesses the register or memory using the default path specified in the enclosing block or system abstraction class.

Example

Example B-35

```
this.ral_model.status_reg.read(status, value,
                               vmm_ral::BACKDOOR);
```

Example B-36

```
status = uart_ral.status_reg.tx_en.write_t(1,
                                           vmm_ral::BFM);
```

vmm_ral::access_e

Symbolic values identifying the behavior of the bits implementing a field when read or written.

SystemVerilog

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
vmm_ral::A1
vmm_ral::OTHER
vmm_ral::USER0
vmm_ral::USER1
vmm_ral::USER2
vmm_ral::USER3
vmm_ral::DC
```

OpenVera

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
vmm_ral::A1
vmm_ral::OTHER
vmm_ral::USER0
vmm_ral::USER1
vmm_ral::USER2
vmm_ral::USER3
vmm_ral::DC
```

Description

`vmm_ral::RW`

The content of the field can be read and written. The content of the field is never modified by the design.

`vmm_ral::RO`

The content of the field can be read. Writing has no effect on its content. The content of the field is never modified by the design. It may be possible to modify the content of the field using "`vmm_ral_field::poke()`".

`vmm_ral::WO`

The content of the field can be written. The value read is always zero and has no correlation with the content of the field. You can obtain the actual content of the field by using "`vmm_ral_field::peek()`".

`vmm_ral::W1`

The content of the field can be read but can be physically written only once after a reset. The write-once mechanism prevents further modifications of the content of the field after the first "`vmm_ral_field::write()`" operation. The content of the field can be written using "`vmm_ral_field::poke()`" any number of times. The field can only be written again after a reset operation.

`vmm_ral::RU`

The content of the field can be read. Writing has no effect on its content. The content of the field can be modified by the design. It may be possible to modify the content of the field using "`vmm_ral_field::poke()`".

`vmm_ral::RC`

The content of the field can be read. The content of the field is cleared after each read operation. Writing has no effect on its content. The content of the field is modified by the design. The actual content of the field can be obtained without modifying it using "`vmm_ral_field::peek()`".

`vmm_ral::W1C`

The content of the field can be read. Writing to the field clears all bits corresponding to the bits that are set in the value being written. Bits that are cleared have no effect on its content. The content of the field is modified by the design. Using "`vmm_ral_field::poke()`" can force the bits to a specific value.

`vmm_ral::A1`

The content of the field can be read and cleared. Writing a 1 has no effect. The content of the field can also be set by the design. Using "`vmm_ral_field::poke()`" can force the bits to a specific value.

`vmm_ral::A0`

The content of the field can be read and set. Writing a 0 has no effect. The content of the field can also be cleared by the design. Using "`vmm_ral_field::poke()`" can force the bits to a specific value.

`vmm_ral::DC`

The content of the field is never checked for correctness (don't care) against the mirrored value when using the "vmm_ral_field::mirror()" method. The content of the mirror is updated as if the field was a vmm_ral::RW field. Use when the content of the field is not really predictable.

vmm_ral::OTHER

The effect of reading or writing the field on its content is unknown or may have effects on the content of other fields. The mirrored value assumes a rw behavior.

vmm_ral::USER*n*

The effect of read or writing the field on its content is user-defined. The mirrored value assumes a rw behavior.

Example

Example B-37

```
vmm_ral_field fields[];
ral_model.status_reg.get_fields(fields);
foreach (fields[i]) begin
   if (fields[i].get_access() != vmm_ral::RO) begin
      `vmm_error(log, "A status bit is not read-only");
   end
end
```

Example B-38

```
vmm_ral_field fields[*];
reg.get_fields(fields);
foreach (fields, i) {
   if (fields[i].get_access() == vmm_ral::W1C &&
       value[i] == 1'b0) begin
      value[i] = 1'b1;
   end
end
status = reg.write_t(value);
```

vmm_ral::check_e

Symbolic values identifying the behavior when an expected value does not match an actual value.

SystemVerilog

```
vmm_ral::QUIET
vmm_ral::VERB
```

OpenVera

```
vmm_ral::QUIET
vmm_ral::VERB
```

Description

`vmm_ral::QUIET`

No message reporting the discrepancy is reported.

`vmm_ral::VERB`

A message reporting the discrepancy is reported through an appropriate instance of the message service interface.

Example

Example B-39

```
this.ral_model.mirror(status, vmm_ral::QUIET,
                      vmm_ral::BACKDOOR);
```

Example B-40

```
status = this.ral_model.mirror_t(vmm_ral::VERB);
```

vmm_ral::endianness_e

Symbolic values identifying the endianness of values when mapped to a narrower data path.

SystemVerilog

```
vmm_ral::NO_ENDIAN
vmm_ral::LITTLE_ENDIAN
vmm_ral::BIG_ENDIAN
vmm_ral::LITTLE_FIFO
vmm_ral::BIG_FIFO
```

OpenVera

```
vmm_ral::NO_ENDIAN
vmm_ral::LITTLE_ENDIAN
vmm_ral::BIG_ENDIAN
vmm_ral::LITTLE_FIFO
vmm_ral::BIG_FIFO
```

Description

`vmm_ral::NO_ENDIAN`

It is not possible to map wide values onto discrete values on a narrower data path. Used as default value only.

`vmm_ral::LITTLE_ENDIAN`

Values are mapped onto a narrower data path using consecutive addresses, with the least significant bits in the lower addresses.

`vmm_ral::BIG_ENDIAN`

Values are mapped onto a narrower data path using consecutive addresses, with the most significant bits in the lower addresses.

`vmm_ral::LITTLE_FIFO`

Values are mapped onto a narrower data path using consecutive values at the same addresses, with the least significant bits transferred first.

`vmm_ral::BIG_FIFO`

Values are mapped onto a narrower data path using consecutive values at the same addresses, with the most significant bits transferred first.

Example

Example B-41

```
ral_blk_myblk  ral_blk  =  new ( vmm_ral ::  LITTLE_ENDIAN);
```

vmm_ral::reset_e

Symbolic values identifying the type of reset.

SystemVerilog

```
vmm_ral::HARD
vmm_ral::SOFT
```

OpenVera

```
vmm_ral::HARD
vmm_ral::SOFT
```

Description

`vmm_ral::HARD`

A hard reset.

`vmm_ral::VERB`

A soft reset.

Example

Example B-42

```
class tb_vmm_ral extends vmm_ral;
   ...
   reset_e rst_inst;

   function new();
      super.new();
   endfunction
   ...
endclass
```

```
class tb_env extends vmm_ral_env;
   ...
   tb_vmm_ral tb_ral;
   ...
   function new();
     super.new();
     this.tb_ral = new();
   endfunction

   virtual task hw_reset();
     tb_ral.rst_inst = vmm_ral::HARD;
     `vmm_note(log,$psprintf("reset_e =
%0d",tb_ral.rst_inst));
   endtask
   ...
endclass
```

vmm_ral::coverage_e

Symbolic values identifying functional coverage models.

SystemVerilog

```
vmm_ral::NO_COVERAGE
vmm_ral::REG_BITS
vmm_ral::ADDR_MAP
vmm_ral::FIELD_VALS
vmm_ral::ALL_COVERAGE
```

OpenVera

```
vmm_ral::NO_COVERAGE
vmm_ral::REG_BITS
vmm_ral::ADDR_MAP
vmm_ral::FIELD_VALS
vmm_ral::ALL_COVERAGE
```

Description

Symbolic values identifying functional coverage models. In order to use this class, the corresponding functional coverage models must have been previously generated using the `-c` option of `ralgen`. See "Predefined Functional Coverage Models" on page 121 for more details.

`vmm_ral::NO_COVERAGE`

No coverage models.

`vmm_ral::REG_BITS`

The "Register Bits" coverage model.

`vmm_ral::ADDR_MAP`

The “Address Map” coverage model. This model can only be dynamically controlled through a block or system abstraction class.

`vmm_ral::FIELD_VALS`

The “Field Values” coverage model. This model can only be dynamically controlled through a block or system abstraction class.

`vmm_ral::ALL_COVERAGE`

All known coverage models.

Example

Example B-43

```
class ral_reg_SAMPLE extends vmm_ral_reg;
   ...
   local virtual function void domain_coverage(string
domain,bit rights,int idx);
     if (this.can_cover(vmm_ral::REG_BITS)) begin
         ral_cvr_reg_oc_ethernet_INT_MASK cg;
         cg = new(this.get_fullname(), domain);
         this.reg_bits[idx] = cg;
     end
   endfunction
   ...
endclass
```

vmm_ral_access

RAL component managing accesses to registers and memories through physical interfaces. Also provides an access-by-address and access-by-name service to registers and memories.

Summary

- vmm_ral_access::set_model() ..... page B-87
- vmm_ral_access::get_model() ..... page B-89
- vmm_ral_access::add_xactor() ..... page B-90
- vmm_ral_access::read() ..... page B-92
- vmm_ral_access::write() ..... page B-94
- vmm_ral_access::burst_read() ..... page B-96
- vmm_ral_access::burst_write() ..... page B-98
- vmm_ral_access::default_path ..... page B-100
- vmm_ral_access::set_by_name() ..... page B-102
- vmm_ral_access::get_by_name() ..... page B-104
- vmm_ral_access::read_by_name() ..... page B-106
- vmm_ral_access::write_by_name() ..... page B-108
- vmm_ral_access::read_mem_by_name() ..... page B-110
- vmm_ral_access::write_mem_by_name() ..... page B-112

vmm_ral_access::set_model()

Associates a RAL abstraction model with the RAL physical access component.

SystemVerilog

```
function void set_model(vmm_ral_block_or_sys model)
```

OpenVera

```
task set_model(vmm_ral_block_or_sys model)
```

Description

Associates the specified RAL abstraction model with the RAL physical access component instance. Once a model is associated with an access component, registers, fields and memories can be accessed through the RAL.

A model can be associated with only one access component. Similarly, an access component can be associated with only one abstraction model.

It is possible to have multiple instances of the access component associated with their respective abstraction model.

Example

Example B-44

```
class my_env extends vmm_ral_env;
   ral_block_my_block model;

   function new();
      this.model = new;
```

```
      super.ral.set_model(this.model);
   endfunction: new
endclass: my_env
```

Example B-45

```
class my_env extends rvm_env {
   test_cfg cfg;
   vmm_ral_access ral[];
   ral_block_my_block model[];
   ...
   function build();
      this.ral = new [this.cfg.n];
      this.model = new [this.cfg.n];
      foreach (this.ral, i) {
         this.model[i] = new;
         this.ral[i].set_model(this.model[i]);
      end
   }
   ...
}
```

vmm_ral_access::get_model()

Returns the RAL abstraction model associated with the RAL physical access component.

SystemVerilog

```
function vmm_ral_block_or_sys get_model()
```

OpenVera

```
function vmm_ral_block_or_sys get_model()
```

Description

Return the RAL abstraction model that was associated with the RAL physical access component instance using the "vmm_ral_access::set_model()" method.

Example

Example B-46

```
if (this.ral.get_model() == null) begin
   `vmm_fatal(log, "No RAL model was specified");
end
```

vmm_ral_access::add_xactor()

Associates a physical-level transactor with a domain in the RAL abstraction model.

SystemVerilog

```
function void add_xactor(vmm_rw_xactor xact,
                         string       domain = "")
```

OpenVera

```
task add_xactor(vmm_rw_xactor xact,
                string        domain = "")
```

Description

Associates the specified physical-level transactor with the specified domain in the RAL abstraction model. The specified domain must exist in the model and only one transactor can be associated with a particular domain.

The physical-level transactor is implicitly started when added to the RAL abstraction model. This allows the predefined RAL tests to execute and the RAL model to later be used to configure the DUT in the `vmm_env::cfg_dut()` step, before the `vmm_env::start()` step has been executed. The physical-level transactor can be explicitly started in the `vmm_env::start()` step without adverse effects.

If a domain has no physical-level transactor associated with it, it is not possible to perform physical accesses to its registers and memories.

This method must be called after "vmm_ral_access::set_model()" has been called.

Example

Example B-47

```
class my_env extends vmm_ral_env;
   ral_ahb_master ahb_ma;
   ...
   function void build();
      super.build();
      this.ahb_ma = new(...);
      super.ral.add_xactor(this.ahb_ma);
   endfunction: build
   ...
endclass: my_env
```

Example B-48

```
class my_env extends vmm_ral_env {
   ral_pci_master pci_ma;
   ral_pci_config pci_cfg;
   ...
   task build() {
      super.build();
      this.pci_ma = new(...);
      super.ral.add_xactor(this.pci_ma, "PCI");
      this.pci_cfg = new(...);
      super.ral.add_xactor(this.pci_cfg, "CFG");
   }
   ...
}
```

vmm_ral_access::read()

Reads a value from a specified physical address.

SystemVerilog

```
task read(output vmm_rw::status_e status,
          input  bit [63:0]       addr,
          output bit [63:0]       data,
          input  int              n_bits = 64,
          string                  domain = "",
          input  int              data_id = -1,
          input  int              scenario_id = -1,
          input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(bit [63:0] addr,
                                 var bit [63:0] data,
                                 integer   n_bits = 64,
                                 string    domain = "",
                                 integer   data_id = -1,
                                 integer   scenario_id = -1
                                 integer   stream_id = -1)
```

Description

Reads a value from the specified physical address through the physical interface of the specified domain.

Returns an indication of the success or failure of the operation.

The optional value of the arguments:

```
n_bits
data_id
scenario_id
stream_id
```

...are assigned to their corresponding `vmm_data/rvm_data` class property in the `"vmm_rw_access"` transaction descriptor used to execute the read cycle. This allows the read cycle to be traced back to the higher-level transaction that caused the cycle to occur.

The mirrored content of any register or memory located at that address is not updated. This method is provided if low-level read operations are necessary. Reading of fields, registers or memory locations should be done using the `"vmm_ral_field::read()"`, `"vmm_ral_reg::read()"` or `"vmm_ral_mem::read()"` methods, respectively.

Example

Example B-49

```
this.ral.read(status, 'h0000, value);
if (status != vmm_rw::OK) begin
   vmm_error(log, "Error reading from 0x0000");
end
```

Example B-50

```
for (i = 0; i < 'hFFFF; i++) {
   status = this.ral.read_t(i, value, , "AHB");
   if (status != vmm_rw::OK) {
      vmm_error(log, psprintf("Error reading from AHB'h%h",
                                i));
   }
}
```

vmm_ral_access::write()

Writes a value at a specified physical address.

SystemVerilog

```
task write(output vmm_rw::status_e status,
           input  bit [63:0]       addr,
           input  bit [63:0]       data,
           input  int              n_bits,
           input  string           domain = "",
           input  int              data_id = -1,
           input  int              scenario_id = -1,
           input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(bit [63:0] addr,
                                  bit [63:0] data,
                                  integer    n_bits,
                                  string     domain = "",
                                  integer    data_id = -1,
                               integer    scenario_id = -1,
                                 integer    stream_id = -1)
```

Description

Writes the specified value at the specified physical address through the physical interface of the specified domain.

Returns an indication of the success or failure of the operation.

The optional value of the arguments:

```
n_bits
data_id
scenario_id
stream_id
```

...are assigned to their corresponding `vmm_data/rvm_data` class property in the "`vmm_rw_access`" transaction descriptor used to execute the write cycle. This allows the write cycle to be traced back to the higher-level transaction that caused the cycle to occur.

The mirrored content of any register or memory located at that address is not updated. This method is provided if low-level write operations are necessary. Writing of fields, registers or memory locations should be done using the "`vmm_ral_field::write()`", "`vmm_ral_reg::write()`" and "`vmm_ral_mem::write()`" methods, respectively.

Example

Example B-51

```
this.ral.write(status, 'h0000, 'h0003);
if (status != vmm_rw::OK) begin
   vmm_error(log, "Error writing to 0x0000");
end
```

Example B-52

```
for (i = 0; i < 'hFFFF; i++) {
   status = this.ral.write_t(i, random(), , "AHB");
   if (status != vmm_rw::OK) {
      vmm_error(log, psprintf("Error writing to AHB'h%h",
                              i));
   }
}
```

vmm_ral_access::burst_read()

Reads a series value from a specified set of physical addresses.

SystemVerilog

```
task burst_read(output vmm_rw::status_e status,
                input  bit [63:0]         start,
                input  bit [63:0]         incr,
                input  bit [63:0]         max,
                input  int                n_beats,
                output bit [63:0]         data[],
                input  vmm_data           user = null,
                input  int                n_bits = 64,
                string                    domain = "",
                input  int                data_id = -1,
                input  int               scenario_id = -1,
                input  int                stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e burst_read_t(
     input  bit [63:0] start,
                input  bit [63:0] incr,
                input  bit [63:0] max,
                input  integer    n_beats,
                output bit [63:0] data[],
                input  vmm_data   user = null,
                input  integer    n_bits = 64,
                string            domain = "",
                input  integer    data_id = -1,
                input  integer    scenario_id = -1,
                input  integer    stream_id = -1)
```

Description

Reads a set of values using a burst read cycle through the physical interface of the specified domain. The following parameters are used to populate the "vmm_rw_burst" descriptor that will eventually be executed by the "vmm_rw_xactor::execute_burst()" method:

```
start
incr
max
n_beats
user
```

Returns an indication of the success or failure of the operation.

The mirrored content of any register or memory located in the burst area is not updated. This method is provided if low-level burst-read operations are necessary. Burst-read operations should be done using the "vmm_ral_mem::burst_read()" method.

Example

Example B-53

```
...
   this.ral.burst_read(status,'h0000,'h1000, value);
   if (status != vmm_rw::OK) begin
   `vmm_error(log, "Error reading a series value from a
0x1000");
   end
   ...
```

vmm_ral_access::burst_write()

Writes a series of values at a specified set of physical addresses.

SystemVerilog

```
task burst_write(output vmm_rw::status_e status,
                 input  bit [63:0]       start,
                 input  bit [63:0]       incr,
                 input  bit [63:0]       max,
                 input  bit [63:0]       data[],
                 input  vmm_data         user = null,
                 input  int              n_bits = 64,
                 string                  domain = "",
                 input  int              data_id = -1,
                 input  int              scenario_id = -1,
                 input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e burst_write_t(
     input  bit [63:0] start,
                 input  bit [63:0] incr,
                 input  bit [63:0] max,
                 input  bit [63:0] data[],
                 input  vmm_data   user = null,
                 input  integer    n_bits = 64,
                 string            domain = "",
                 input  integer    data_id = -1,
                 input  integer    scenario_id = -1,
                 input  integer    stream_id = -1)
```

Description

Writes a set of value using a burst write cycle through the physical interface of the specified domain. The following parameters are used to populate the "vmm_rw_burst" descriptor that will eventually be executed by the "vmm_rw_xactor::execute_burst()" method:

```
start
incr
max
user
```

The number of beats is assumed to be equal to the number of data values to be written.

Returns an indication of the success or failure of the operation.

The mirrored content of any register or memory located in the burst area is not updated. This method is provided if low-level burst-write operations are necessary. Burst-write operations should be done using the "vmm_ral_mem::burst_write()" method.

Example

Example B-54

```
...
   this.ral.burst_write(status,'h0000,'h1000,'h100F);
   if (status != vmm_rw::OK) begin
  vmm_error(log, "Error writing a series value at 0x1000");
   end
   ...
```

vmm_ral_access::default_path

Specifies the default path to use when accessing registers and memories.

SystemVerilog

```
vmm_ral::path_e default_path
```

OpenVera

```
vmm_ral::path_e default_path
```

Description

Specifies the default path to use when accessing fields, registers or memories in the design. This default path can be superseded by the default access path in the system, sub-system and block abstraction models. This default path can also be superseded on a per-access basis.

The value of this property must be `vmm_ral::BFM` or `vmm_ral::BACKDOOR`. Whereas it is the topmost and ultimate default path specification, it cannot be `vmm_ral::DEFAULT`.

This default path is not used when using the "vmm_ral_access::read()" and "vmm_ral_access::write()" methods.

Example

Example B-55

```
super.ral.default_path = vmm_ral::BACKDOOR;
this.ral_model.soc.blk.default_path = vmm_ral::BFM;
```

```
this.ral_model.mirror();
```

vmm_ral_access::set_by_name()

Sets the mirror value of the specified register.

SystemVerilog

```
virtual function bit set_by_name(
   string      name,
   bit [63:0] value)
```

OpenVera

```
virtual function bit set_by_name(
   string      name,
   bit [63:0] value)
```

Description

Locates the register with the specified name and sets its value mirrored in the RAL abstraction model. The actual register in the design is not written or updated. See "vmm_ral_reg::set()" for more details on the operation of the mirror.

See "vmm_ral_block_or_sys::get_reg_by_name()" for details on how the register is located. Returns TRUE if a unique register of the specified name is found in the RAL model. Returns FALSE otherwise.

It is better to use the "vmm_ral_reg::set()" method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-56

```
Use:

   ral_model.blk.ctrl_reg.set(’h0001);

instead of:

   ral.set_by_name("ctrl_reg", ’h0001);
```

Example B-57

```
string reg_name;
reg_name = compute_name();
if (!this.ral.set_by_name(reg_name, ’hFFFF)) {
   rvm_error(log, {"No such register: ", reg_name};
}
```

vmm_ral_access::get_by_name()

Returns the mirror value of the specified register.

SystemVerilog

```
virtual function bit get_by_name(
   input  string      name,
   output bit [63:0] value)
```

OpenVera

```
virtual function bit get_by_name(
   string          name,
   var bit [63:0] value)
```

Description

Locates the register with the specified name and returns its value mirrored in the RAL abstraction model. The actual register in the design is not read. See "vmm_ral_reg::get()" for more details on the operation of the mirror.

See "vmm_ral_block_or_sys::get_reg_by_name()" for details on how the register is located. Returns TRUE if a unique register of the specified name is found in the RAL model. Returns FALSE otherwise.

It is better to use the "vmm_ral_reg::get()" method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-58

```
Use:

   ral_model.blk.status_reg.get(status);

instead of:

   ral.get_by_name("status_reg", status);
```

Example B-59

```
string reg_name;
reg_name = compute_name();
if (!this.ral.get_by_name(reg_name, val)) {
   rvm_error(log, {"No such register: ", reg_name};
}
printf("%s = %h\n", reg_name, val);
```

vmm_ral_access::read_by_name()

Reads a value from a specified named register.

SystemVerilog

```
task read_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_by_name_t(
   string          name,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the register with the specified name and performs the specified read operation through its RAL abstraction model. The mirror is updated.

See "vmm_ral_block_or_sys::get_reg_by_name()" for details on how the register is located.

It is better to use the "vmm_ral_reg::read()" method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-60

```
Use:

   ral_model.blk.status_reg.read(status, val);

instead of:

   ral.read_by_name(status, "status_reg", val);
```

Example B-61

```
string reg_name;
reg_name = compute_name();
status = this.ral.read_by_name_t(reg_name, val);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such register: ", reg_name};
}
else printf("%s = %h\n", reg_name, val);
```

vmm_ral_access::write_by_name()

Writes a value from a specified named register.

SystemVerilog

```
task write_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_by_name_t(
   string          name,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the register with the specified name and performs the specified write operation through its RAL abstraction model. The mirror is updated.

See "vmm_ral_block_or_sys::get_reg_by_name()" for details on how the register is located.

It is better to use the "vmm_ral_reg::write()" method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-62

```
Use:

   ral_model.blk.ctrl_reg.write(status, 'h0003);

instead of:

   ral.write_by_name(status, "ctrl_reg", 'h0003);
```

Example B-63

```
string reg_name;
reg_name = compute_name();
status = this.ral.write_by_name_t(reg_name, 'hFFFF);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such register: ", reg_name};
}
```

vmm_ral_access::read_mem_by_name()

Reads a value from a specified named memory.

SystemVerilog

```
task read_mem_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        input  bit [63:0]       offset,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_mem_by_name_t(
   string          name,
   bit [63:0]      offset,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the memory with the specified name and performs the specified read operation through its RAL abstraction model. The mirror is updated.

See "vmm_ral_block_or_sys::get_mem_by_name()" for details on how the memory is located.

It is better to use the "vmm_ral_mem::read()" method directly in the RAL abstraction class for a memory in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-64

```
Use:

   ral_model.blk.mem0.read(status, 0, val);

instead of:

   ral.read_mem_by_name(status, "mem0", 0, val);
```

Example B-65

```
string mem_name;
mem_name = compute_name();
status = this.ral.read_mem_by_name_t(mem_name, 0, val);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such memory: ", mem_name};
}
else printf("%s[0] = %h\n", mem_name, val);
```

vmm_ral_access::write_mem_by_name()

Writes a value from a specified named register.

SystemVerilog

```
task write_mem_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        input  bit [63:0]       offset,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_mem_by_name_t(
   string          name,
   bit [63:0]      offset,
   bit [63:0]      data,
   vmm_ral::path_e path = DEFAULT
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the memory with the specified name and performs the specified write operation through its RAL abstraction model. The mirror is updated.

See "vmm_ral_block_or_sys::get_mem_by_name()" for details on how the memory is located.

It is better to use the "vmm_ral_mem::write()" method directly in the RAL abstraction class for a memory in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-66

```
Use:

   ral_model.blk.mem0.write(status, 0, 'h0003);

instead of:

   ral.write_mem_by_name(status, "mem0", 0, 'h0003);
```

Example B-67

```
string mem_name;
mem_name = compute_name();
status = this.ral.write_mem_by_name_t(mem_name, 0, 'hFFFF);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such memory: ", mem_name};
}
```

vmm_ral_block_or_sys

Virtual base class for block and system descriptors. Provides functionality that is identical between blocks and systems.

Summary

- vmm_ral_block_or_sys::log ..... page B-116
- vmm_ral_block_or_sys::get_name() ..... page B-117
- vmm_ral_block_or_sys::get_type() ..... page B-118
- vmm_ral_block_or_sys::get_fullname() ..... page B-120
- vmm_ral_block_or_sys::get_domains() ..... page B-121
- vmm_ral_block_or_sys::get_parent_domain() ..... page B-123
- vmm_ral_block_or_sys::get_external_domain() ..... page B-124
- vmm_ral_block_or_sys::get_n_tops() ..... page B-125
- vmm_ral_block_or_sys::get_parent() ..... page B-126
- vmm_ral_block_or_sys::get_base_addr() ..... page B-127
- vmm_ral_block_or_sys::get_top() ..... page B-128
- vmm_ral_block_or_sys::C_addr_of() ..... page B-129
- vmm_ral_block_or_sys::set_offset() ..... page B-130
- vmm_ral_block_or_sys::get_n_bytes() ..... page B-132
- vmm_ral_block_or_sys::get_endian() ..... page B-134
- vmm_ral_block_or_sys::default_access ..... page B-136
- vmm_ral_block_or_sys::get_default_access() ..... page B-138
- vmm_ral_block_or_sys::display() ..... page B-140
- vmm_ral_block_or_sys::psdisplay() ..... page B-141
- vmm_ral_block_or_sys::psdisplay_domain() ..... page B-143
- vmm_ral_block_or_sys::get_fields() ..... page B-145
- vmm_ral_block_or_sys::get_field_by_name() ..... page B-147
- vmm_ral_block_or_sys::get_registers() ..... page B-149
- vmm_ral_block_or_sys::get_reg_by_name() ..... page B-150
- vmm_ral_block_or_sys::get_reg_by_offset() ..... page B-152
- vmm_ral_block_or_sys::get_virtual_fields() ..... page B-154
- vmm_ral_block_or_sys::get_virtual_field_by_name() page B-155
- vmm_ral_block_or_sys::get_virtual_registers() .... page B-156
- vmm_ral_block_or_sys::get_vreg_by_name() ..... page B-157
- vmm_ral_block_or_sys::get_memories() ..... page B-158
- vmm_ral_block_or_sys::get_mem_by_name() ..... page B-159
- vmm_ral_block_or_sys::get_constraints() ..... page B-160
- vmm_ral_block_or_sys::has_cover() ..... page B-164
- vmm_ral_block_or_sys::set_cover() ..... page B-165
- vmm_ral_block_or_sys::is_cover_on() ..... page B-167
- vmm_ral_block_or_sys::reset() ..... page B-169
- vmm_ral_block_or_sys::needs_update() ..... page B-170
- vmm_ral_block_or_sys::needs_update() ..... page B-170
- vmm_ral_block_or_sys::update() ..... page B-172
- vmm_ral_block_or_sys::mirror() ..... page B-173
- vmm_ral_block_or_sys::readmemh() ..... page B-175
- vmm_ral_block_or_sys::writememh() ..... page B-176
- vmm_ral_block_or_sys::set_attribute() ..... page B-177
- vmm_ral_block_or_sys::get_attribute() ..... page B-178
- vmm_ral_block_or_sys::get_all_attributes() ..... page B-179

- vmm_ral_block_or_sys::ral_power_down() ..... page B-180
- vmm_ral_block_or_sys::ral_power_up() ..... page B-181

vmm_ral_block_or_sys::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the block or system descriptor. A single message service interface instance is shared by all block and system abstraction class instances.

Example

Example B-68

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_name()

Returns the name of the block or system.

SystemVerilog

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

OpenVera

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

Description

Returns the name of the block or system corresponding to the instance of the descriptor.

Example

Example B-69

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_name=>
Gets the name",
                          " of the block or system: %0s"},
                              log.get_name()));
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_type()

Returns the type name of the block or system.

SystemVerilog

```
virtual function string get_type()
```

OpenVera

```
virtual function string get_type()
```

Description

Returns the name of the block or system corresponding to the declaration of the descriptor.

This name is usually the same as the instance name returned by "vmm_ral_block_or_sys::get_name()", except when the instance name has been renamed because of multiple instances of the same block or system, or because a domain in a multiple-domain block or system is instantiated.

Example

Example B-70

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_type=>
```

```
Gets the type",
                        " name of the block or system: %0s"},
                                log.get_type()));
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_fullname()

Returns the fully-qualified name of the block or system.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the block or system corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-71

```
`vmm_note(log, $psprintf("Block Name:
%s\n",ral_model.block_name.get_fullname()));
```

vmm_ral_block_or_sys::get_domains()

Returns the name of the domains in the block or system.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the domains in the block or system. The order of the domain names is not specified.

Example

Example B-72

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys, string domain);
     super.new("VMM RAL System",sys,"");
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   this.get_domains("");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_domain
s=>Gets the name",
                            " name of the domains in the
block or system.",
                     " size of names = %0d"},names.size()));
```

```
    ...
endclass
```

vmm_ral_block_or_sys::get_parent_domain()

Returns the name of the parent-level domain.

SystemVerilog

```
function string get_parent_domain(string domain)
```

OpenVera

```
function string get_parent_domain(string domain)
```

Description

Returns the name of the parent-level domain that instantiates the specified domain of this block or system.

Example

Example B-73

```
string reg_domains[];
vmm_ral_block b = reg.get_parent();
reg.get_domains(domains);
$write("Register %s is in domain %s\n",
       reg.get_fullname(),
b.get_parent_domain(domains[0]));
```

vmm_ral_block_or_sys::get_external_domain()

Returns the name of the top-level domain.

SystemVerilog

```
function string get_external_domain(string domain)
```

OpenVera

```
function string get_external_domain(string domain)
```

Description

Return the name of the top-level domain that instantiates the specified domain of this block or system.

Example

Example B-74

```
string reg_domains[];
vmm_ral_block b = reg.get_parent();
reg.get_domains(domains);
$write("Register %s is in domain %s\n",
       reg.get_fullname(),
b.get_external_domain(domains[0]));
```

vmm_ral_block_or_sys::get_n_tops()

Return the number of top-level block or system RAL models.

SystemVerilog

```
static function int vmm_ral_block_or_sys::get_n_tops();
```

Description

The number of tops depends upon the way it is instantiated in the environment. If a RAL block has no parent then, it is considered to be the top.

Example

```
ral_sys_dut ral_model = new;
vmm_log log = new("prog_blk","test");
vmm_ral_access acc = new();
vmm_ral_reg regs[];

initial begin
acc.set_model(ral_model);

$display("No. of tops are %0d",ral_model.get_n_tops());

if (ral_model.get_n_tops() != 4)
   `vmm_error(log,$psprintf("expected total num of tops is
      4 but got %0d",ral_model.get_n_tops()));

log.report();
```

vmm_ral_block_or_sys::get_parent()

Returns the system that instantiates this block or system.

SystemVerilog

```
virtual function vmm_ral_sys get_parent()
```

OpenVera

```
virtual function ram_ral_sys get_parent()
```

Description

Returns a reference to the descriptor of the system that includes the block or system corresponding to the descriptor instance. If this is the top-level block or system, returns null.

Example

Example B-75

```
vmm_ral_block_or_sys::get_parent() : [Page 263]

  vmm_ral_sys parent;
  parent = ral_model.block_name.get_parent();
  parent.display();
```

vmm_ral_block_or_sys::get_base_addr()

Returns the base address of the block or system.

SystemVerilog

```
virtual function bit [63:0] get_base_addr(string domain = "")
```

OpenVera

```
virtual function bit [63:0] get_base_addr(string domain = "")
```

Description

Returns the base address of the specified domain of the block or system in the address space of the immediately instantiating system.

If this is the top-level block or system, always returns 0.

Example

Example B-76

```
bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
addr = ral_model.block_name.get_base_addr();
```

vmm_ral_block_or_sys::get_top()

Return the specified top-level block or system RAL model.

SystemVerilog

```
static function vmm_ral_block_or_sys vmm_ral_block_or_sys::
   get_top( int n = 0);
```

Description

This returns the specified top-level block or system RAL model.

Example

```
ral_sys_dut ral_model = new;
vmm_log log = new("prog_blk","test");
vmm_ral_access acc = new();
vmm_ral_reg regs[];
vmm_ral_block_or_sys my_ral;
initial begin
   acc.set_model(ral_model);
   my_ral = ral_model.get_top(2);
   $display("2nd top is %s",my_ral.get_name());
```

vmm_ral_block_or_sys::C_addr_of()

Gets the base address of the block or system for the C API.

SystemVerilog

```
function int C_addr_of()
```

OpenVera

```
function integer C_addr_of()
```

Description

Returns the address of the block or system in the C API address space. The returned address is designed to be passed to the RAL C API and is not designed to be used as the actual physical base address of the block or system.

vmm_ral_block_or_sys::set_offset()

Modify the base address of the block or system.

SystemVerilog

```
virtual function bit set_offset(bit [63:0] offset,
    string domain = "")
```

OpenVera

```
virtual function bit set_offset(
   bit [63:0] offset,
   string domain = "");
```

Description

Dynamically relocate the base address of the specified domain in the block or subsystem in the address space of the immediately instantiating system. The new address range for the block or subsystem must not be occupied by another block or subsystem. Note that after using this method, the behavior of the RAL model will be different from the RALF specification.

Returns TRUE of the relocation was succesful. Returns FALSE if the specified domain does not exist in the immediately enclosing system or the new base address creates an overlap between this block or subsystem address range and another block or subsystem.

It is not possible to relocate the base address of the top-level system because is it not instantiated anywhere.

Example

Example B-77

```
ral_model.block_name.set_offset('h1000);
```

vmm_ral_block_or_sys::get_n_bytes()

Returns the width of the physical interface on this block or system.

SystemVerilog

```
virtual function int unsigned get_n_bytes(
   string domain = "")
```

OpenVera

```
virtual function integer get_n_bytes(
   string domain = "")
```

Description

Returns the width, in number of bytes, of the physical interface of the block or system for the specified domain.

Example

Example B-78

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys string domain);
     super.new("VMM RAL System",sys,"", 24);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   `vmm_note(log,$psprintf({"
vmm_ral_block_or_sys::get_n_bytes => Gets the",
                          " width of the physical interface
on this block",
                         " or system. Width (bytes) = %0d"},
                             this.get_n_bytes("")));
```

```
    ...
endclass
```

vmm_ral_block_or_sys::get_endian()

Returns the endianness of the physical interface on this block or system.

SystemVerilog

```
virtual function vmm_ral::endianness_e get_endian(
   string domain = "")
```

OpenVera

```
virtual function vmm_ral::endianness_e get_endian(
   string domain = "")
```

Description

Returns the endianness of the physical interface of the block or system for the specified domain.

Example

Example B-79

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   `vmm_note(log,$psprintf({"
vmm_ral_block_or_sys::get_endian => Gets the",
                            " endianness of the physical
interface on this",
```

```
                                  " block or system ENDIAN = %0d"},
                                     this.get_endian("")));
   ...
endclass
```

vmm_ral_block_or_sys::default_access

Specifies the default access path for this block or system.

SystemVerilog

```
vmm_ral::path_e default_access
```

OpenVera

```
vmm_ral::path_e default_access
```

Description

Specifies the default access path when reading and writing registers and memories in this block or system.

If set to `vmm_ral::DEFAULT`, uses the default access path of the parent system. If set to `vmm_ral::DEFAULT` and this is the top-level block or system, uses the default access path of the RAL access interface. See "vmm_ral_access::default_path" for further details.

Example

Example B-80

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral::path_e default_access = vmm_ral::DEFAULT;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
```

```
   endfunction
   ...
endclass
...
this.ral.write(status, 'h0000, 'h000F);
if(status != vmm_rw::OK) begin
  `vmm_error(log, "Error reading from 0x0000");
end
...
```

vmm_ral_block_or_sys::get_default_access()

Returns the default access path for this block or system.

SystemVerilog

```
virtual function vmm_ral::path_e get_default_access()
```

OpenVera

```
virtual function vmm_ral::path_e get_default_access()
```

Description

Determines the default access path for this block or system, based on the value specified in its `default_access` property and its parent systems. See "vmm_ral_block_or_sys::default_access" for more details.

Example

Example B-81

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral::path_e default_access = vmm_ral::DEFAULT;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_defaul
```

```
t_access => Gets",
                              " the default access path for
block or system ",
                              "Default Access =
%0d"},this.get_default_access()));
   ...
endclass
```

vmm_ral_block_or_sys::display()

Displays a description of the block or system to `stdout`.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the "`vmm_ral_block_or_sys::psdisplay()`" method to the standard output.

Example

Example B-82

```
ral_model.block_name.display();
```

vmm_ral_block_or_sys::psdisplay()

Creates a human-readable description of the block or system.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the block or system, including the registers and memories it contains. Each line of the description is prefixed with the specified prefix.

A description of all domains within the block or system is created.

In a multi-domain block or system, if a description is required only for a specific domain, then you use the `vmm_ral_block_or_sys::psdisplay_domain()` function instead.

Example

Example B-83

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
```

```
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::psdisplay=
>Creates a human-",
                           " readable description of the
block or system: %s",
                       this.psdisplay("RAL_BLK_SYS","")));
   ...
endclass
```

vmm_ral_block_or_sys::psdisplay_domain()

Creates a human-readable description of the block or system.

SystemVerilog

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

OpenVera

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

Description

Creates a human-readable description of the block or system, including the registers and memories it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, only a description of that domain is created. Otherwise, a description of all domains within the block or system is created.

Example

Example B-84

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
...
vmm_log log;
...
function new(vmm_ral_sys sys,string domain);
super.new("VMM RAL System",sys,"",vmm_ral::LITTLE_ENDIAN);
log = new("VMM RAL BLOCK/SYS","Log");
endfunction
...
```

```
`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::psdisplay_
domain=>Creates a human-",
" readable description of the block or system: %s",
this.psdisplay_domain("RAL_BLK_SYS","")));
...
endclass
```

vmm_ral_block_or_sys::get_fields()

Returns all fields in this block or system.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_field fields[],
   input string     domain = "")
```

OpenVera

```
virtual task get_fields(
   var vmm_ral_field fields[*],
   string           domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the fields contained in the block or system. If a domain is specified, only the fields accessible through the specified domain are returned. The order in which the fields are located in the array is not specified.

Example

Example B-85

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral_field SAMPLE;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.SAMPLE = new(this,"SAMPLE","");
   endfunction
```

```
    ...
    this.get_fields(this.SAMPLE,"");
    ...
endclass
```

vmm_ral_block_or_sys::get_field_by_name()

Returns the field with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

Description

Finds a field with the specified name in the block or system and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within registers. Therefore, if used on a system or block with more than one field having the same name, this method returns the first field found.

Example

Example B-86

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral_field SAMPLE1;
   vmm_ral_field SAMPLE2;
   vmm_ral_field temp;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
```

```
      log = new("VMM RAL BLOCK/SYS","Log");
      this.SAMPLE1 = new(this,"SAMPLE1","");
      this.SAMPLE2 = new(this,"SAMPLE2","");
    endfunction
    ...
    temp = this.get_field_by_name("SAMPLE");
    ...
endclass
```

vmm_ral_block_or_sys::get_registers()

Returns all registers in this block or system.

SystemVerilog

```
virtual function void get_registers(
   ref vmm_ral_reg regs[],
   input string   domain = "")
```

OpenVera

```
virtual task get_registers(
   var vmm_ral_reg regs[*],
   string         domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the registers contained in the block or system. If a domain is specified, only the registers accessible by the specified domain are returned. The order in which the registers are located in the array is not specified.

Example

Example B-87

```
vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      regs[i].display();
  end
```

vmm_ral_block_or_sys::get_reg_by_name()

Returns the register with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_reg get_reg_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_reg get_reg_by_name(
   string name)
```

Description

Finds a register with the specified name in the block or system and return its descriptor. If no registers are found, returns null.

Register name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one register having the same name, this method returns the first register found.

Example

Example B-88

```
class ral_reg_REG_SAMPLE extends vmm_ral_reg;
  ...
  rand vmm_ral_field RX_M;
  rand vmm_ral_field TX_M;
  ...
  function new(vmm_ral_block blk);
     super.new(parent, name);
     this.RX_M = new(this, "RX_M");
     this.TX_M = new(this, "TX_M");
  endfunction
```

```
  ...
endclass

class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ral_reg_REG_SAMPLE REG_SAMPLE;
   vmm_ral_reg temp;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.REG_SAMPLE = = new(this, "REG_SAMPLE");
   endfunction
   ...
   temp = this.get_reg_by_name("REG_SAMPLE");
   ...
endclass
```

vmm_ral_block_or_sys::get_reg_by_offset()

Gets the register at the specified offset in this block or system.

SystemVerilog

```
virtual function vmm_ral_reg
   vmm_ral_sys::get_reg_by_offset(
   bit [63:0] offset, string domain = "")
virtual function vmm_ral_reg
   vmm_ral_block::get_reg_by_offset(
   bit [63:0] offset, string domain = "")
```

OpenVera

```
virtual function vmm_ral_reg
   vmm_ral_sys::get_reg_by_offset(
   bit [63:0] offset, string domain = "")
virtual function vmm_ral_reg
   vmm_ral_block::get_reg_by_offset(
   bit [63:0] offset, string domain = "")
```

Description

Finds the register located at the specified offset within the block or system address space in the specified domain and returns its descriptor. If no register is found at the specified offset, returns NULL.

The entire register may occupy more than one offset within the address space of the block or system if it is wider than the physical interface. In such cases, this function looks for the start (lowest) address of the register's address space.

This function has two implementation versions – a default version and a faster version, which takes more memory than the default version.

In the default (slower) version, all registers of the underlying blocks and subsystems are searched, short-circuiting the search, wherever possible, to check if a register exists at the specified block/system level offset/address.

The faster version uses associative arrays to cache block level register offsets, which helps to speed up the search, given an offset. Memory consumption is higher in the faster version, due to caching. You can activate the faster version by defining the runtime macro VMM_RAL_FAST_SRCH.

Example

Example B-89

```
vmm_ral_reg register;
  register = ral_model.block_name.get_reg_by_offset('h10);
  if (register == null) `vmm_error(log, "wrong offset");
```

vmm_ral_block_or_sys::get_virtual_fields()

Returns all virtual fields in this block or system.

SystemVerilog

```
virtual function void get_virtual_fields(
   ref vmm_ral_vfield fields[],
   input string      domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields contained in the block or system. If a domain is specified, only the fields implemented in memories accessible through the specified domain are returned. The order in which the fields are located in the array is not specified.

Example

Example B-90

```
vmm_ral_block_or_sys::get_virtual_fields() [page 279]

  vmm_ral_vfield vfields[];
  ral_model.block_name.get_virtual_fields(vfields);
  foreach (vfields[i]) begin
     vfields[i].display();
  end
```

vmm_ral_block_or_sys::get_virtual_field_by_name()

Returns the virtual field with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_vfield get_virtual_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name in the block or system and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within virtual registers. Therefore, if used on a system or block with more than one field having the same name, this method returns the first field found.

Example

Example B-91

```
vmm_ral_block_or_sys::get_virtual_field_by_name() [page
280]

  vmm_ral_vfield vfield;
  vfield =
ral_model.block_name.get_virtual_field_by_name("virtual_fi
eld_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_block_or_sys::get_virtual_registers()

Returns all virtual registers in this block or system.

SystemVerilog

```
virtual function void get_virtual_registers(
   ref vmm_ral_vreg regs[],
   input string    domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual registers contained in the block or system. If a domain is specified, only the registers implemented in memories accessible by the specified domain are returned. The order in which the registers are located in the array is not specified.

Example

Example B-92

```
vmm_ral_block_or_sys::get_virtual_registers() [page 281]

  vmm_ral_vreg vregs[];
  ral_model.block_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_ral_block_or_sys::get_vreg_by_name()

Returns the virtual register with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_vreg get_vreg_by_name(
   string name)
```

Description

Finds a virtual register with the specified name in the block or system and return its descriptor. If no registers are found, returns null.

Register name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one register having the same name, this method returns the first register found.

Example

Example B-93

```
vmm_ral_block_or_sys::get_vreg_by_name() [page 282]

  vmm_ral_vreg vreg;
  vreg =
ral_model.block_name.get_vreg_by_name("virtual_reg_name");
  if (vreg == null) `vmm_error(log, "specified virtual
register doesn't exists");
```

vmm_ral_block_or_sys::get_memories()

Returns all memories in this block or system.

SystemVerilog

```
virtual function void get_memories(
   ref vmm_ral_mem mems[],
   input string   domain = "")
```

OpenVera

```
virtual task get_memories(
   var vmm_ral_mem mems[*],
   string         domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the memories contained in the block or system. If a domain is specified, only those memories accessible in the specified domain are returned. The order in which the memories are located in the array is not specified.

Example

Example B-94

```
vmm_ral_mem memories[];
  ral_model.block_name.get_memories(memories);
  foreach (memories[i]) begin
      memories[i].display();
  end
```

vmm_ral_block_or_sys::get_mem_by_name()

Returns the memory with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_mem get_mem_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_mem get_mem_by_name(
   string name)
```

Description

Finds a memory with the specified name in the block or system and returns its descriptor. If no memories are found, returns null.

Memory name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one memory having the same name, this method returns the first memory found.

Example

Example B-95

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   rand my_ral_mem my_mem;
   vmm_ral_mem temp;
   ...
   temp = this.get_mem_by_name("my_mem");
   ...
endclass
```

vmm_ral_block_or_sys::get_constraints()

Returns the constraint blocks in this block or system.

SystemVerilog

```
virtual function void get_constraints(
   ref string names[])
```

OpenVera

```
virtual task get_constraints(
   var string names[*])
```

Description

Fills the specified dynamic array with the names of the constraint blocks in this block or system. Does not include the constraint blocks in the registers or fields in this block or system. The location of each constraint block name in the array is not defined.

Example

Example B-96

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   rand vmm_ral_field MINFL;
   ...
   constraint MINFL_spec {
      MINFL.value == 'h40;
   }
   ...
endclass
...
my_ral_block_or_sys my_blk_sys;
...
```

```
string str[];
...
this.ral_model.my_blk_sys.get_constraints(str);
foreach (str[i])
   `vmm_note(log,$psprintf("Constraint Name is
%0s",str[i]));
...
```

vmm_ral_block_or_sys::get_n_tops();

Return the number of top-level block or system RAL models.

SystemVerilog

```
static function int vmm_ral_block_or_sys::get_n_tops();
```

OpenVera

```
Not Supported.
```

Description

The number of tops depends upon the way it is instantiated in the env. If a ral block has no parent then it is considered as top.

Example

Example B-97

```
ral_sys_dut ral_model = new;
vmm_log log = new("prog_blk","test");
vmm_ral_access acc = new();
vmm_ral_reg regs[];

initial begin
acc.set_model(ral_model);
$display("No. of tops are %0d",ral_model.get_n_tops());

if (ral_model.get_n_tops() != 4)
  `vmm_error(log,$psprintf("expected total num of tops is 4
but got %0d",ral_model.get_n_tops()));

log.report();
```

vmm_ral_block_or_sys::get_top();

Return the specified top?level block or system RAL model.

SystemVerilog

```
static function vmm_ral_block_or_sys
vmm_ral_block_or_sys::get_top(int n = 0);
```

OpenVera

Not Supported.

Description

This returns the specified top?level block or system RAL model.

Example

Example B-98

```
ral_sys_dut ral_model = new;
vmm_log log = new("prog_blk","test");
vmm_ral_access acc = new();
vmm_ral_reg regs[];
vmm_ral_block_or_sys my_ral;

initial begin

acc.set_model(ral_model);
my_ral = ral_model.get_top(2);

$display("2nd top is %s",my_ral.get_name());
.............

.............
```

vmm_ral_block_or_sys::has_cover()

Query available functional coverage models.

SystemVerilog

```
virtual function int has_cover(vmm_ral::coverage_e models)
```

OpenVera

```
virtual function integer has_cover(vmm_ral::coverage_e
models)
```

Description

Queries which of the specified functional coverage models are available. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

Returns the sum of the available functional coverage models.

A functional coverage model is available only if it has been generated by `ralgen` (see "Predefined Functional Coverage Models" on page 121) then enabled when calling "vmm_ral_block::new()" or "vmm_ral_sys::new()".

Example

Example B-99

```
ral_model.set_cover(ral_model.has_cover(vmm_all::ALL_CO
VERAGE));
```

vmm_ral_block_or_sys::set_cover()

Turns functional coverage measurement on or off.

SystemVerilog

```
virtual function int set_cover(vmm_ral::coverage_e is_on)
```

OpenVera

```
virtual function integer set_cover(vmm_ral::coverage_e
is_on)
```

Description

Turns the collection of functional coverage measurements on or off for this block or system and all subsystems, blocks, registers, fields and memories within it. The functional coverage measurement is turned on for every coverage model specified. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off.

Returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that have been generated by `ralgen` (see "Predefined Functional Coverage Models" on page 121) then enabled when calling "vmm_ral_block::new()" or "vmm_ral_sys::new()".

See the "vmm_ral_block_or_sys::has_cover()" method to identify the available functional coverage models.

Example

Example B-100

```
ral_model.set_cover(vmm_all::NO_COVERAGE));

ral_model.set_cover(vmm_all::REG_BITS +
vmm_all::ADDR_MAP);
```

vmm_ral_block_or_sys::is_cover_on()

Queries if functional coverage measurement is on or off.

SystemVerilog

```
virtual function bit is_cover_on(vmm_ral::coverage_e is_on)
```

OpenVera

```
virtual function bit is_cover_on(vmm_ral::coverage_e is_on)
```

Description

Returns TRUE of measurement for all of the specified functional coverage models that are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. See "vmm_ral_block_or_sys::set_cover()" for more details.

Example

Example B-101

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   function new(vmm_ral_sys sys,int cover_on);
     super.new("VMM RAL System",sys,vmm_ral::NO_COVERAGE);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::is_cover_o
n=>Queries if",
                         "functional coverage measurement
is on or off.",
                                  " =
%b"},this.is_cover_on(vmm_ral::REG_BITS));
   endfunction
```

```
    ...
endclass
```

vmm_ral_block_or_sys::reset()

Resets the mirror values in this block or system.

SystemVerilog

```
virtual function void reset(string           domain = "",
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

OpenVera

```
virtual task reset(string           domain = "",
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of all registers in the block or system to the specified hard or soft reset value. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding descriptor in the RAL model.

If a domain is specified, only those registers accessible through the specified domain are reset.

The mirror values of memories is not modified.

Example

Example B-102

```
ral_model.block_name.reset();
```

vmm_ral_block_or_sys::needs_update()

Queries if a mirrored value in this block or system has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If a mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the registers are outdated. This method returns TRUE if the state of the registers needs to be updated to match the mirrored values (or vice-versa).

The mirror values, or actual content of registers, are not modified. For additional information, see "vmm_ral_block_or_sys::update()" or "vmm_ral_block_or_sys::mirror()".

Example

Example B-103

```
class ral_reg_REG_SAMPLE extends vmm_ral_reg;
  ...
  rand vmm_ral_field RX_M;
  rand vmm_ral_field TX_M;
  ...
  function new(vmm_ral_block blk);
     super.new(parent, name);
     this.RX_M = new(this, "RX_M");
```

```
     this.TX_M = new(this, "TX_M");
  endfunction
  ...
endclass

class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ral_reg_REG_SAMPLE REG_SAMPLE;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.REG_SAMPLE = = new(this, "REG_SAMPLE");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::needs_upda
te=>Queries if a"
                            "mirrored value in this block
or system has been",
                       " set. = %b",this.needs_update()));
   ...
endclass
```

vmm_ral_block_or_sys::update()

Updates registers to match mirrored values in this block or system.

SystemVerilog

```
virtual task update(
   output vmm_rw::status_e status,
   input vmm_ral::path_e  path = vmm_ral::DEFAULT)
```

OpenVera

```
virtual function vmm_rw::status_e update_t(
   vmm_ral::path_e path = vmm_ral::DEFAULT)
```

Description

Using the minimum number of write operations, updates the content of the registers in the design to match the mirrored values. The update can be performed using the physical interfaces (front-door access) or back-door (zero-time) access.

This method performs the reverse operation of "vmm_ral_block_or_sys::mirror()".

Example

Example B-104

```
bit update;
vmm_rw::status_e status;
update = ral_model.block_name.needs_update();
if (update == 1) begin
   ral_model.block_name.update(status);
end
```

vmm_ral_block_or_sys::mirror()

Updates the mirrored value of registers in this block or system to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e status,
   input  vmm_ral::check_e  check = vmm_ral::QUIET,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT)
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e  path  = vmm_ral::DEFAULT)
```

Description

Updates the content of the registers mirror values to match their corresponding values in the design. The mirroring can be performed using the physical interfaces (front-door access) or back-door (zero-time) access. If the `check` argument is specified as `vmm_ral::VERB`, an error message is issued if the current mirrored value does not match the actual value in the design.

This method performs the reverse operation of "vmm_ral_block_or_sys::update()".

Example

Example B-105

```
vmm_rw::status_e status;
ral_model.block_name.mirror(status);
```

vmm_ral_block_or_sys::readmemh()

Initializes the registers and memories in the block or system.

SystemVerilog

```
virtual task readmemh(string filename)
```

OpenVera

```
virtual task readmemh_t(string filename)
```

Description

Not yet implemented.

Initializes the content of all registers and memories in the design using the values in the specified file. The values are updated using the default access path. See "vmm_ral_block_or_sys::writememh()" for details.

The format of the file is not specified.

vmm_ral_block_or_sys::writememh()

Dumps the value of all registers and memories in the block or system.

SystemVerilog

```
virtual task writememh(string filename)
```

OpenVera

```
virtual task writememh_t(string filename)
```

Description

Not yet implemented.

Dumps the content of all registers and memories in the design to the specified file. The file can then be used as an input for the "vmm_ral_block_or_sys::readmemh()" method. The values are obtained using the default access path.

The format of the file is not specified.

vmm_ral_block_or_sys::set_attribute()

Set an attribute for a block or system.

SystemVerilog

```
virtual function void set_attribute(string name,
    string value);
```

OpenVera

```
Not supported
```

Description

Set the specified attribute to the specified value for this block or system. If the value is specified as "", the specified attribute is deleted.

A warning is issued if an existing attribute is modified.

Attribute names are case sensitive.

vmm_ral_block_or_sys::get_attribute()

Get an attribute from a block or system.

SystemVerilog

```
virtual function string get_attribute(string name,
    bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Get the value of the specified attribute for this block or system. If the attribute does not exists, "" is returned.

If the "inherited" argument is specifed as TRUE, the value of the attribute is inherited from the nearest enclosing system if it is not specified for this block or system. If it is specified as FALSE, the value "" is returned if it does not exists in the this block or system.

Attribute names are case sensitive.

vmm_ral_block_or_sys::get_all_attributes()

Get all attributes for a block or system.

SystemVerilog

```
virtual function void get_all_attributes(
     ref string names[],
     input bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Return an array filled with the name of the attributes defined for this block or system.

If the "inherited" argument is specifed as TRUE, the value of all attributes inherited from the enclosing system(s) is included. If the argument is specified as FALSE, only the attributed defined for this block or system are returned.

The order in which attribute names are returned is not specified.

vmm_ral_block_or_sys::ral_power_down()

Specify that a block or system is powered down.

SystemVerilog

```
virtual function void ral_power_down(bit retain = 0);
```

OpenVera

```
Not supported
```

Description

Specify that this block or all blocks in this system has been put in a power-saving state. A read or write access to any register or memory inside the powered-down block will result in a run-time error message and a vmm_ral::ERROR status code.

If the "retain" argument is TRUE, the mirrored value of registers with an inherited non-zero RETAIN attribute value will be maintained and restored when the block is powered back up using the vmm_ral_block_or_sys::ral_power_up() method. If the retain argument is FALSE, the mirrored value of registers will be set to the reset value when the block or system is powered back up.

A powered-down block with retention enabled can be further powered down with retention disabled.

This method does not physically power down the block or system. It only indicates to the RAL model that the block or system has been powered down.

vmm_ral_block_or_sys::ral_power_up()

Specify that blocks are powered back up.

SystemVerilog

```
virtual function void ral_power_up(
    string power_domains = "");
```

OpenVera

```
Not supported
```

Description

Specify that the block or blocks in the system and the memories within them with an inherited POWER_DOMAIN attribute value that matches the specified power domain regular expression have been restored to a powered-up state. If the power domain is specified as "", then the block or blocks in the system and any memory within them are powered up regardless of the POWER_DOMAIN attribute value.

If a block was as powered down using the vmm_ral_block_or_sys::ral_power_down() method with a "retain" argument specified as TRUE, the mirrored value of registers with an inherited non-zero RETAIN attribute value is restored. Otherwise, the mirrored value of registers is set to the specified reset value. By default, a block or system is powered-up.

This method does not physically power up the block or system. It only indicates to the RAL model that the block or system has been powered up.

vmm_ral_block

Block descriptor class derived from "vmm_ral_block_or_sys".

Summary

- vmm_ral_block::new() ..... page B-183
- vmm_ral_block::sample_field_values() ..... page B-185

vmm_ral_block::new()

Creates an instance of a RAL model.

SystemVerilog

```
function new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

OpenVera

```
task new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

Description

Creates an instance of a RAL model with the corresponding block as the top-level structural element.

The `cover_on` argument specifies the functional coverage models to be enabled in the RAL model. Multiple functional coverage models may be specified by adding their symbolic names. Only functional coverage models that were generated by `ralgen` using the `-c` option can be enabled. Because the functional coverage models affect the memory footprint and runtime performance of a RAL model, they should be enabled only when relevant.

It is not possible to enable a functional coverage model at a later time, but it is possible to turn the measurement of a functional coverage model off then back on using the "vmm_ral_block_or_sys::set_cover()" method.

Example

Example B-106

```
ral_sys_mysys ral_model = new(vmm_ral::REG_BITS +
                              vmm_ral::ADDR_MAP);
```

vmm_ral_block::sample_field_values()

Samples the field values within this block.

SystemVerilog

```
function void sample_field_values()
```

OpenVera

```
function void sample_field_values()
```

Description

By using this function, you can sample the field value coverage within the RAL registers.

In the generated code, it will be extended to call `field_values.sample()` for all the registers for which you have the "field value" coverage enabled.

Example

Example B-107

```
class top_ral_env extends vmm_ral_env;
  ral_block_top ral_model; //where ral_block_top is an
instance of vmm_ral_block  ...
  task cfg_dut();
    ...
    //writes to different registers
    ral_model.sample_field_values();
  endtask
endclass
```

vmm_ral_env

Base class, derived from `vmm_env`, to be used when creating RAL-based verification environments. This section documents only the differences or additions in this class in comparison to the base class. You can find the documentation for the properties and methods inherited as-is in the *Reference Verification Methodology User's Guide* and the *Verification Methodology Manual for SystemVerilog*.

Summary

- `vmm_ral_env::new()` ..... page B-187
- `vmm_ral_env::ral` ..... page B-188
- `vmm_ral_env::hw_reset` ..... page B-190
- `vmm_ral_env::sw_reset` ..... page B-192
- `vmm_ral_env::reset_dut` ..... page B-193

vmm_ral_env::new()

Creates a RAL-based environment.

SystemVerilog

```
function new(string name = "RAL-Based Verif Env")
```

OpenVera

```
task new(string name = "RAL-Based Verif Env")
```

Description

Creates a new instance of a RAL-based environment base class. This method is usually called by the constructor of a user-defined extension of this class using `super.new()`.

Example

Example B-108

```
class tb_env extends vmm_ral_env;
   ...
   function new();
      super.new("RAL Based RTL Env");
      this.cfg = new;
   endfunction
   ...
endclass
```

vmm_ral_env::ral

RAL access interface object instance.

SystemVerilog

```
vmm_ral_access ral
```

OpenVera

```
vmm_ral_access ral
```

Description

Instance of the RAL access interface used to access registers and memories in the design verified by this environment. The instance is allocated in the constructor for this class and should not be modified, replaced, nulled or reallocated. See "vmm_ral_access" for additional information.

Example

Example B-109

```
class tb_env extends vmm_ral_env;
   ...
   if(this.ral != null)
     begin
      `vmm_note(log,{"Instance of vmm_ral_access class has
been created. ",
                  "Instance name is ral."});
     end
   else
     begin
     `vmm_error(log,"Creation of instance of vmm_ral_access
class is failed.");
     end
```

```
    ...
endclass
```

vmm_ral_env::hw_reset

Performs a hardware reset operation.

SystemVerilog

```
virtual task hw_reset()
```

OpenVera

```
virtual task hw_reset_t()
```

Description

This method must be overloaded in a user-extension of this base class. It performs a complete hardware reset of the design. The design must be in its reset state when the task returns.

This method is called by the default implementation of the "vmm_ral_env::reset_dut" method.

Example

Example B-110

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
      tb_top.reset <= 1;
      `vmm_note(log,"Entire System Hardware is reseted.");
      @ (posedge tb_top.clk);
      tb_top.reset <= 0;
   endtask
   ...
endclass
...
initial
```

```
begin
   tb_env env = new();
   env.reset_dut();
end
```

vmm_ral_env::sw_reset

Performs a software reset operation.

SystemVerilog

```
virtual task sw_reset()
```

OpenVera

```
virtual task sw_reset_t()
```

Description

This method must be overloaded in a user-extension of this base class. It performs a complete software reset of the design. The design must be in its reset state when the task returns.

Example

Example B-111

```
class tb_env extends vmm_ral_env;
   ...
   virtual task sw_reset();
      tb_top.reset <= 1;
      `vmm_note(log,"Entire System Software is reseted.");
      @ (posedge tb_top.clk);
      tb_top.reset <= 0;
   endtask

   virtual task reset_dut();
     super.reset_dut();
     this.sw_reset();
   endtask
   ...
endclass
```

vmm_ral_env::reset_dut

Hardware reset step in a simulation sequence.

SystemVerilog

```
virtual task reset_dut()
```

OpenVera

```
virtual task reset_dut_t()
```

Description

This method calls the "vmm_ral_env::hw_reset" method to reset the design. It should not need to be overloaded in a user-defined extension of this class.

It is important that the design under verification be as idle and inactive as possible after the completion of this method. The predefined tests built on top of the verification environment expect that the value of registers will remain unchanged between accesses.

Example

Example B-112

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
   ...
   ...
   endtask
   ...
endclass
...
```

```
program test_ral_env;
  ...
  initial
    begin
       tb_env env = new();
       env.reset_dut();
    end
  ...
endprogram
```

vmm_ral_field

Field descriptors.

Summary

- vmm_ral_field::log ..... page B-196
- vmm_ral_field::get_name() ..... page B-197
- vmm_ral_field::get_fullname() ..... page B-198
- vmm_ral_field::get_register() ..... page B-199
- vmm_ral_field::get_lsb_pos_in_register() ..... page B-200
- vmm_ral_field::get_n_bits() ..... page B-201
- vmm_ral_field::get_access() ..... page B-202
- vmm_ral_field::set_access() ..... page B-204
- vmm_ral_field::display() ..... page B-206
- vmm_ral_field::psdisplay() ..... page B-207
- vmm_ral_field::set() ..... page B-208
- vmm_ral_field::predict() ..... page B-210
- vmm_ral_field::get() ..... page B-212
- vmm_ral_field::reset() ..... page B-213
- vmm_ral_field::set_reset() ..... page B-214
- vmm_ral_field::needs_update() ..... page B-216
- vmm_ral_field::read() ..... page B-217
- vmm_ral_field::write() ..... page B-219
- vmm_ral_field::peek() ..... page B-223
- vmm_ral_field::poke() ..... page B-225
- vmm_ral_field::mirror() ..... page B-227
- vmm_ral_field::append_callback() ..... page B-229
- vmm_ral_field::prepend_callback() ..... page B-231
- vmm_ral_field::unregister_callback() ..... page B-233

vmm_ral_field::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the field descriptor. A single message service interface instance is shared by all field abstraction class instances.

Example

Example B-113

```
class my_ral_field extends vmm_ral_field;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_reg rg);
      super.new("VMM RAL REGISTER",rg);
      log = new("VMM RAL REGISTER","Log");
     `vmm_note(log,"vmm_ral_field::log instance created.");
      ...
   endfunction
   ...
endclass
```

vmm_ral_field::get_name()

Returns the name of the field.

SystemVerilog

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

OpenVera

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

Description

Returns the name of the field corresponding to the instance of the descriptor.

Example

Example B-114

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(mems);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Field Name: %s\n",
fields[i].get_name()));
  end
```

vmm_ral_field::get_fullname()

Returns the fully-qualified name of the field.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the field corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-115

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(mems);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Field Name: %s\n",
fields[i].get_fullname()));
  end
```

vmm_ral_field::get_register()

Returns the register that instantiates this field.

SystemVerilog

```
virtual function vmm_ral_reg get_register()
```

OpenVera

```
virtual function ram_ral_reg get_register()
```

Description

Returns a reference to the descriptor of the register that includes the field corresponding to the descriptor instance.

Example

Example B-116

```
  vmm_ral_reg register;
  register =
ral_model.block_name.reg1.field_1.get_register();
  register.display();
```

vmm_ral_field::get_lsb_pos_in_register()

Returns the offset of the least-significant bit of the field.

SystemVerilog

```
virtual function int unsigned get_lsb_pos_in_register()
```

OpenVera

```
virtual function integer get_lsb_pos_in_register()
```

Description

Returns the index of the least significant bit of the field in the register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register.

Example

Example B-117

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(fields);
  foreach (fields[i]) begin
     `vmm_note(log, $psprintf("Offset of LSB of the field:
%s\n", fields[i].get_lsb_pos_in_register()));
  end
```

vmm_ral_field::get_n_bits()

Returns the width of the field.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

OpenVera

```
virtual function integer get_n_bits()
```

Description

Returns the width, in number of bits, of the field.

Example

Example B-118

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(fields);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Width of the field: %d\n",
fields[i].get_n_bits()));
  end
```

vmm_ral_field::get_access()

Returns the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the field when written and read through the optionally specified domain.

If the register containing the field is shared across multiple domains, a domain must be specified. The access mode of a field in a specific domain may be restricted. For example, a RW field may only be writable through one of the domains and read-only through all of the other domains.

Example

Example B-119

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand my_ral_field my_field;
   ...
   `vmm_note(log,$psprintf({"vmm_ral_field::get_access =>
Gets the access mode",
             " of the field.=
%0d"},this.my_field.get_access("")));
```

```
    ...
endclass
```

vmm_ral_field::set_access()

Set the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e
set_access(vmm_ral::access_e mode)
```

OpenVera

```
virtual function vmm_ral::access_e
set_access(vmm_ral::access_e mode)
```

Description

Set the access mode of the field to the specified mode and return the previous access mode.

WARNING! Using this method will modify the behavior of the RAL model from the behavior specified in the original specification.

Example

Example B-120

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand vmm_ral_field my_ral_field;
   ...
   function new(vmm_ral_block blk);
      super.new("VMM RAL BLOCK",blk);
      log = new("VMM RAL BLOCK","Log");
   endfunction
   ...
   my_ral_field my;
   ...
```

```
   this.my.set_access(vmm_ral::RW);
   ...
   `vmm_note(log,$psprintf({"After Setting the access mode
of the field.",
            " access mode = %0d"},this.my.get_access("")));
   ...
endclass
```

vmm_ral_field::display()

Displays a description of the field to stdout.

SystemVerilog

```
virtual function void display(string prefix = "")
```

OpenVera

```
virtual task display(string prefix = "")
```

Description

Displays the image created by the "vmm_ral_field::psdisplay()" method on the standard output.

Example

Example B-121

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
foreach (fields[i]) begin
    fields[i].display();
end
```

vmm_ral_field::psdisplay()

Creates a human-readable description of the field.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the field and its current mirrored value. Each line of the description is prefixed with the specified prefix.

Example

Example B-122

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
foreach (fields[i]) begin
   `vmm_note(log, $psprintf("psdisplay of the field: %s\n",
fields[i].psdisplay()));
end
```

vmm_ral_field::set()

Sets the mirror value of the field.

SystemVerilog

```
virtual function void set(bit [63:0] value)
```

OpenVera

```
virtual task set(bit [63:0] value)
```

Description

Sets the mirror value of the field to the specified value. Does not actually set the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the "vmm_ral_reg::update()" method to update the actual register with the mirrored value or the "vmm_ral_field::write()" method to set the actual field and its mirrored value.

The final value in the mirror is a function of the field access mode and the set value, just like a normal physical write operation to the corresponding bits in the hardware. As such, this method (when eventually followed by a call to "vmm_ral_reg::update()") is a zero-time functional replacement for the "vmm_ral_field::write()" method. For example, the mirrored value of a read-only field is not modified by this method, and the mirrored value of a write-once field can only be set if the field has not yet been written to using a physical (for example, front-door) write operation.

To modify the mirrored value to a specific value, regardless of the access mode, and thus use the RAL mirror as a scoreboard for the register values in the DUT, use the "vmm_ral_field::predict()" method.

Example

Example B-123

```
$write("Setting the mirrored value to the field to specified
value");
ral_model.fld.set(8'hff);
```

vmm_ral_field::predict()

Force the mirror value of the field.

SystemVerilog

```
virtual function bit predict(bit [63:0] value,
                             string fname = "",
                             int lineno = 0,
                             bit force_predict = 0)
```

OpenVera

```
virtual function bit predict(bit [63:0] value)
```

Description

Force the mirror value of the field to the specified value. Does not actually force the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the "vmm_ral_reg::update()" method to update the actual register with the mirrored value or the "vmm_ral_field::write()" method to set the actual field and its mirrored value.

The final value in the mirror is the specified value, regardless of the access mode. For example, the mirrored value of a read-only field is modified by this method, and the mirrored value of a read-update field can be updated to any value predicted to correspond to the value in the corresponding physical bits in the design. By default, `predict` does not allow any update of the mirror, when RAL is busy executing a transaction on this field. However, if need be, that can be overridden, by setting the `force_predict` argument to 1.

Returns FALSE if this method is called while the register containing the field is being read or written, therefore, rendering the prediction unreliable. Returns TRUE otherwise.

Example

Example B-124

```
...
this.ral_model.my_reg.my_field.predict(data);
`vmm_note(log,$psprintf("Forced Value ==> %0h",data));
`vmm_note(log,$psprintf("Previous Value ==> %h",
          this.ral_model.my_reg.my_field.get()));

this.ral_model.my_reg.write(status,data);
`vmm_note(log,$psprintf("Written Value ==> %h",
          this.ral_model.my_reg.my_field.get()));
...
```

vmm_ral_field::get()

Returns the mirror value of the field.

SystemVerilog

```
virtual function bit [63:0] get()
```

OpenVera

```
virtual function bit [63:0] get()
```

Description

Returns the mirror value of the field. Does not actually read the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model.

The mirrored value of a write-only field is the value that was set or written and assumed to be stored in the bits implementing the field. Even though a physical read operation of a write-only field returns zeroes, this method returns the assumed content of the field.

Use the "vmm_ral_field::read()" method to read the actual field and update its mirrored value.

Example

Example B-125

```
bit [63:0] data;
data = ral_model.block_name.reg1.field_1.get();
```

vmm_ral_field::reset()

Resets the mirror values in the field.

SystemVerilog

```
virtual function void reset(vmm_ral::reset_e kind =
vmm_ral::HARD)
```

OpenVera

```
virtual task reset(vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of the field to the specified reset value. Does not actually reset the value of the field in the design, only the value mirrored in the descriptor in the RAL model.

The value of a write-once (`vmm_ral::W1`) field can be subsequently modified each time a hard reset is applied.

Example

Example B-126

```
ral_model.block_name.reg1.field_1.reset();
```

vmm_ral_field::set_reset()

Modify the reset value.

SystemVerilog

```
virtual function logic [63:0] set_reset(
     logic [63:0]       value,
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

OpenVera

```
virtual function bit [63:0] set_reset(
     bit [63:0]        value,
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Modify the reset value of the field to the specified value and return the previously-defined reset value. A soft-reset value specified as all X's indicates no soft-reset value.

WARNING! Using this method will modify the behavior of the RAL model from the behavior specified in the original register specification.

Example

Example B-127

```
   ...
   bit [63:0] usr_rst;
   ...
   this.ral_model.my_reg.my_field.reset();
   `vmm_note(log,$psprintf("Default Field Reset Value ==>
%0h",
             this.ral_model.my_reg.my_field.get()));
```

```
   this.ral_model.my_reg.my_field.set_reset(usr_rst);
   this.ral_model.my_reg.my_field.reset();
   `vmm_note(log,$psprintf("User-defined Field Reset Value
==> %0h",
             this.ral_model.my_reg.my_field.get()));
   ...
```

vmm_ral_field::needs_update()

Queries if the mirrored value for this field has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If the mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the registers are outdated. This method returns TRUE if the state of the field needs to be updated to match the mirrored values (or vice-versa).

The mirror value or actual content of the field are not modified. See "vmm_ral_reg::update()" or "vmm_ral_reg::mirror()".

Example

Example B-128

```
  bit update;
  vmm_rw::status_e status;
  update =
ral_model.block_name.reg1.field_1.needs_update();
```

vmm_ral_field::read()

Reads a field value from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the field from the design using the specified access path. If a back-door access path is used, the effect of reading the field through a physical access is mimicked. For example, a write-only field will return zeroes.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the field, and all other fields located in the same register, is updated with the value read from the design. The mirror value of write-only fields are never updated during a read operation.

Example

Example B-129

```
...
fork
  begin
    wait(written);
    this.ral_model.my_reg.my_field.read(status,rd_data);
    `vmm_note(log,$psprintf("Read Value:: %0h with status
%0s", rd_data,status.name));
      written = 1'b0;
    end
join_none
   ...
```

vmm_ral_field::write()

Sets a field value in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified field value in the design using the specified access path. If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, a read-only field will not be written.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the register, where this field is physically located, contains other fields, the following values are used for the other bits in the register, based on each field access mode:

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::OTHER
vmm_ral::USERn
```

The value for the other field is written using the mirrored content of the field.

```
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
```

The value for the other field is written as all zeroes.

```
vmm_ral::A1
```

The value for the other field is written as all ones.

If the register containing this field contains other write-once fields, these other fields are written at the same time as this field. If a physical (for example, front-door) access is used, it is not possible to modify their content without first resetting the model. It may be preferable to use a `set()/update()` approach. Once write-once fields are written through a physical (for example, front-door) access, their value can no longer be modified.

The mirrored value of the fields are updated based on the written value and the specified behavior of the field contents after a write operation.

Example

Example B-130

```
class my_ral_reg extends vmm_ral_reg;
   rand vmm_ral_field my_field_1;
   rand vmm_ral_field my_field_2;

   function new(...);
      super.new(...);
     //Here vmm_ral::A1 is value for the access_e enum. You
can change it
      //to get the required value for perticular field.
      this.my_field_1 = new(this, "my_field_1", 1,
vmm_ral::RW, ...);
      this.my_field_1 = new(this, "my_field_1", 1,
vmm_ral::A1, ...);
      ...
   endfunction
   ...
endclass
...
fork
  begin

this.ral_model.my_reg.my_field_1.write(status,data,vmm_ral
::BACKDOOR);
```

```
   `vmm_note(log,$psprintf(" Written Value:: %0h with status
%0s",
              data,status.name);
    written = 1'b1;
  end
  begin
    wait(written);
    this.ral_model.my_reg.my_field_1.read(status,rd_data);
    `vmm_note(log,$psprintf("my_field_1 Value:: %0h with
status %0s",
              rd_data,status.name);

    this.ral_model.my_reg.my_field_2.read(status,rd_data);
    `vmm_note(log,$psprintf("my_field_2 Value:: %0h with
status %0s",
              rd_data,status.name);
    written = 1'b0;
  end
join_none
...
```

vmm_ral_field::peek()

Peek a field value from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Peek the current value of the field from the design using a back-door access. The value of the field in the design is not modified, regardless of the access mode.

The optional value of the `data_id`, `scenario_id` and `stream_id` arguments are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction which caused the access to occur.

The mirrored value of the field, and all other fields located in the same register, is updated with the value peeked from the design.

Example

Example B-131

```
vmm_ral_block  b;
bit [63:0]  value;
begin
b.register.fld.peek(status,value);
 `vmm_note(log, $psprintf("b.register.fld   %h", value));
end
```

vmm_ral_field::poke()

Poke a field value in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e poke_t(
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified field value in the design using a back-door access. The value of the field is updated, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the register, where this field is physically located, contains other fields, the current value of the other fields are peeked first then poked back in.

Example

Example B-132

```
vmm_ral_block  b;
b.register.fld.poke(status, 8'hAB);
if (status != vmm_rw::IS_OK)  begin
`vmm_error(log, $psprintf("Update status was  %s",
status.name()));
end
```

vmm_ral_field::mirror()

Updates the mirrored value of this field to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e status,
   input vmm_ral::check_e  check = vmm_ral::QUIET,
   input vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input string           domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e   path  = vmm_ral::DEFAULT,
   string           domain = "")
```

Description

Updates the content of the field mirror value for all the fields in the same register to match the current values in the design. The mirroring can be performed using the physical interfaces (frontdoor) or "vmm_ral_field::peek()" (backdoor). If the `check` argument is specified as `vmm_ral::VERB`, an error message is issued if the current mirrored value of the entire register does not match the actual value in the design.

The content of a write-only field is mirrored and optionally checked only if a `vmm_ral::BACKDOOR` access path is used to read the register containing the field.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

Example

Example B-133

```
   ...
   this.ral_model.my_reg.my_field.set(data);
   fork
     begin
       this.ral_model.my_reg.my_field.mirror(status);
       `vmm_note(log,$psprintf(" Mirrored Value:: %0h with
status %0s",
                  data,status.name);
       mirrored = 1'b1;
     end
   join_none
   ...
```

vmm_ral_field::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(vmm_ral_field callbacks cbs)
```

OpenVera

```
task append_callback(vmm_ral_field_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the order of registration.

Note that field callback methods will be invoked before their corresponding "vmm_ral_reg" callback methods.

Example

Example B-134

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();
      env.ral_model.my_field.append_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_field::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(vmm_ral_field_callbacks cbs)
```

OpenVera

```
task prepend_callback(vmm_ral_field_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the reverse order of registration.

Note that field callback methods will be invoked before their corresponding "vmm_ral_reg" callback methods.

Example

Example B-135

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();
      env.ral_model.my_field.prepend_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_field::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(vmm_ral_field_callbacks
cbs)
```

OpenVera

```
task unregister_callback(vmm_ral_field_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this field descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-136

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_field.append_callback(cb);
```

```
      //Can't call second time for same instance.
     env.ral_model.my_field.append_callback(cb);  //Wrong
Way
      ...
      env.ral_model.my_field.unregister_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_field_callbacks

Field descriptors.

Summary

- vmm_ral_field_callbacks::pre_write() ..... page B-236
- vmm_ral_field_callbacks::post_write() ..... page B-238
- vmm_ral_field_callbacks::pre_read() ..... page B-240
- vmm_ral_field_callbacks::post_read() ..... page B-242

vmm_ral_field_callbacks::pre_write()

OOP callback invoked before writing a field.

SystemVerilog

```
virtual task pre_write(vmm_ral_field       field,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                      ref string         domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_field       field,
                         var bit [63:0]      wdat,
                         var vmm_ral::path_e path,
                        var string         domain)
```

Description

This callback method is invoked before a value is written to a field in the DUT. The written value, if modified, modifies the actual value that will be written. The path and domain used to write to the field can also be modified.

This callback method is only invoked when the "vmm_ral_field::write()" or the "vmm_ral_reg::write()" method is used to write to the field inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "vmm_ral_field::set()" method.

Because writing a field causes the register to be written, and therefore all of the other fields it contains to also be written, all registered "vmm_ral_field_callbacks::pre_write()"

methods with the fields contained in the register will also be invoked, then all registered "vmm_ral_reg_callbacks::pre_write()" methods with the register containing the field will also be invoked.

Example

Example B-137

```
...
fork
  //It will call pre_write() and post_write() callback
  //methods.
  this.ral_model.my_reg.my_field.write(status,data);
join_none
...
program test;
...
class my_ral_field_callbacks extends
      vmm_ral_field_callbacks;
   ...
   virtual task pre_write(vmm_ral_field              field,
                    ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                     ref vmm_ral::path_e              path,
                    ref string                     domain);
      ...
     if(field.get_name() == "TX_R" && path == vmm_ral::BFM)
       wdat = `VMM_RAL_DATA_WIDTH'h0;
       ...
      endtask: pre_write
     ...
   endclass
   ...
   this.ral_model.my_reg.my_field.append_callback(cb);
   ...
endprogram
```

vmm_ral_field_callbacks::post_write()

OOP callback invoked after writing a field.

SystemVerilog

```
virtual task post_write(vmm_ral_field          field,
                        bit [63:0]             wdat,
                        vmm_ral::path_e        path,
                        string                 domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_field          field,
                          bit [63:0]             wdat,
                          vmm_ral::path_e        path,
                          string                 domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is written to a field in the DUT. The `wdat` value is the final mirrored value in the register as reported by the "`vmm_ral_field::get()`" method.

This callback method is only invoked when the "`vmm_ral_field::write()`" or the "`vmm_ral_reg::write()`" method is used to write to the field inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "`vmm_ral_field::set()`" method.

Because writing a field causes the register to be written and, therefore, all of the other fields it contains to also be written, all registered "`vmm_ral_field_callbacks::post_write()`"

methods with the fields contained in the register will also be invoked. At this point, all registered "vmm_ral_reg_callbacks::post_write()" methods with the register containing the field will also be invoked.

Example

Example B-138

```
program test;
  ...
  class my_ral_field_callbacks extends
       vmm_ral_field_callbacks;
    ...
     virtual task post_write(vmm_ral_field           field,
                        bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                          vmm_ral::path_e             path,
                          string                    domain,
                       ref vmm_rw::status_e        status);
        if(status == vmm_rw::ERROR)
         `vmm_warning(log,$psprintf("Write operation
               failed on field %0s", field.get_name()));
    endtask
    ...
  endclass
  ...
  env.ral_model.my_reg.my_field.append_callback(cb);
  ...
endprogram
```

vmm_ral_field_callbacks::pre_read()

OOP callback invoked before reading a field.

SystemVerilog

```
virtual task pre_read(vmm_ral_field       field,
                      ref vmm_ral::path_e path,
                     ref string         domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_field       field,
                        var vmm_ral::path_e path,
                       var string         domain)
```

Description

This callback method is invoked before a value is read from a field in the DUT. The path and domain used to read from the field can be modified.

This callback method is only invoked when the "vmm_ral_field::read()" or the "vmm_ral_reg::read()" method is used to read from the field inside the DUT. This callback method is not invoked when only the mirrored value is read using the "vmm_ral_field::get()" method.

Because reading a field causes the register to be read and, therefore, all of the other fields it contains to also be read, all registered "vmm_ral_field_callbacks::pre_read()" methods with the fields contained in the register will also be invoked. At this point, all registered "vmm_ral_reg_callbacks::pre_read()" methods with the register containing the field will also be invoked.

Example

Example B-139

```
...
fork
  begin
    //It will call pre_write() and post_write() callback
    //methods.
    this.ral_model.my_reg.my_field.write(status,data);
    write_done = 1'b1;
  end

  begin
    wait(write_done);
    //It will call pre_read() and post_read() callback
    //methods.
    this.ral_model.my_reg.my_field.read(status,read_data);
    `vmm_note(log,$psprintf({"Read Data ==> %0h and Status
          ==> %0s", read_data,status.name));
    write_done = 1'b0;
  end
join_none
...
program test;
...
  class my_ral_field_callbacks extends
        vmm_ral_field_callbacks;
    ...
      virtual task pre_read(vmm_ral_field         field,
                            ref vmm_ral::path_e   path,
                            ref string            domain);
        ...
        path = vmm_ral::BFM;  //Keeping all the read
                                    //operations on BFM.
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.my_field.append_callback(cb);
   ...
endprogram
```

vmm_ral_field_callbacks::post_read()

OOP callback invoked after reading a field.

SystemVerilog

```
virtual task post_read(input vmm_ral_field     field,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_field         field,
                         var bit [63:0]        rdat,
                         vmm_ral::path_e       path,
                         string                domain
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is read from a field in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the "`vmm_ral_field::read()`" method and can be modified.

This callback method is only invoked when the "`vmm_ral_field::read()`" or the "`vmm_ral_reg::read()`" method is used to read from the field inside the DUT. This callback method is not invoked when only the mirrored value is read from using the "`vmm_ral_field::get()`" method.

Because reading a field causes the register to be read and, therefore, all of the other fields it contains to also be read, all registered "`vmm_ral_field_callbacks::post_read()`"

methods with the fields contained in the register will also be invoked. At this point, all registered "vmm_ral_reg_callbacks::post_read()" methods with the register containing the field will also be invoked.

Example

Example B-140

```
program test;
...
  class my_ral_field_callbacks extends
        vmm_ral_field_callbacks;
    ...
    virtual task post_read(vmm_ral_field              field,
                           ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                              vmm_ral::path_e                path,
                              string                       domain,
                           ref vmm_rw::status_e          status);
         if(status == vmm_rw::ERROR)
           `vmm_warning(log,$psprintf("Read operation
                 failed on field %0s", field.get_name()));
    endtask
    ...
  endclass
  ...
  env.ral_model.my_reg.my_field.append_callback(cb);
  ...
endprogram
```

vmm_ral_mem

Memory descriptors.

Summary

- vmm_ral_mem::log ..... page B-245
- vmm_ral_mem::mam ..... page B-246
- vmm_ral_mem::get_name() ..... page B-247
- vmm_ral_mem::get_fullname() ..... page B-248
- vmm_ral_mem::get_domains() ..... page B-249
- vmm_ral_mem::get_block() ..... page B-250
- vmm_ral_mem::get_offset_in_block() ..... page B-251
- vmm_ral_mem::get_address_in_system() ..... page B-252
- vmm_ral_mem::get_size() ..... page B-253
- vmm_ral_mem::get_n_bits() ..... page B-254
- vmm_ral_mem::get_n_bytes() ..... page B-255
- vmm_ral_mem::get_access() ..... page B-256
- vmm_ral_mem::get_rights() ..... page B-258
- vmm_ral_mem::get_virtual_fields() ..... page B-259
- vmm_ral_mem::get_virtual_field_by_name() ..... page B-260
- vmm_ral_mem::get_virtual_registers() ..... page B-261
- vmm_ral_mem::get_vreg_by_name() ..... page B-262
- vmm_ral_mem::display() ..... page B-263
- vmm_ral_mem::psdisplay() ..... page B-264
- vmm_ral_mem::psdisplay_domain() ..... page B-265
- vmm_ral_mem::set_frontdoor() ..... page B-266
- vmm_ral_mem::get_frontdoor() ..... page B-267
- vmm_ral_mem::set_backdoor() ..... page B-268
- vmm_ral_mem::get_backdoor() ..... page B-270
- vmm_ral_mem::init() ..... page B-271
- vmm_ral_mem::init_e ..... page B-272
- vmm_ral_mem::read() ..... page B-274
- vmm_ral_mem::write() ..... page B-276
- vmm_ral_mem::burst_read() ..... page B-278
- vmm_ral_mem::burst_write() ..... page B-281
- vmm_ral_mem::peek() ..... page B-284
- vmm_ral_mem::poke() ..... page B-286
- vmm_ral_mem::readmemh() ..... page B-288
- vmm_ral_mem::writememh() ..... page B-289
- vmm_ral_mem::append_callback() ..... page B-290
- vmm_ral_mem::prepend_callback() ..... page B-292
- vmm_ral_mem::unregister_callback() ..... page B-294
- vmm_ral_mem::set_attribute() ..... page B-296
- vmm_ral_mem::get_attribute() ..... page B-297
- vmm_ral_mem::get_all_attributes() ..... page B-298
- vmm_ral_mem::ral_power_down() ..... page B-299
- vmm_ral_mem::ral_power_up() ..... page B-300

vmm_ral_mem::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the memory descriptor. A single message service interface instance is shared by all memory abstraction class instances.

Example

Example B-141

```
class my_mem extends vmm_ral_mem;
   vmm_log log;
   ...
   function new(...);
      super.new(...);
      log = new("my_ral_mem","log");
      `vmm_note(log,"Log instance is successfully created
for vmm_ral_mem.");
      ...
   endfunction
   ...
endclass
```

vmm_ral_mem::mam

Default allocation manager.

SystemVerilog

```
const vmm_mam mam
```

Description

Default memory allocation manager that can be used to dynamically allocate or reserve regions of consecutive locations. All regions allocated by this allocation manager instance are associated with this memory. By default, the entire memory address space is available for allocation.

This allocation manager must not be replaced with another allocation manager. Instead, it should be reconfigured to match the application requirements.

Example

Example B-142

```
class my_mem extends vmm_ral_mem;
   ...
   if(this.mam == null)
     `vmm_error(log,"Failed to create vmm_mam class instant
in vmm_ral_mem.");
   else
      `vmm_note(log,"vmm_mam class instant is created
successfully.");
   ...
endclass
```

vmm_ral_mem::get_name()

Returns the name of the memory.

SystemVerilog

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

OpenVera

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

Description

Returns the name of the memory corresponding to the instance of the descriptor.

Example

Example B-143

```
  vmm_ral_mem mems[];
  ral_model.block_name.get_memories(mems);
  foreach (mems[i]) begin
      `vmm_note(log, $psprintf("Memory Name: %s\n",
mems[i].get_name()));
  end
```

vmm_ral_mem::get_fullname()

Returns the fully-qualified name of the memory.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the memory corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-144

```
  vmm_ral_mem mems[];
  ral_model.block_name.get_memories(mems);
  foreach (mems[i]) begin
      `vmm_note(log, $psprintf("Memory Name: %s\n",
mems[i].get_fullname()));
  end
```

vmm_ral_mem::get_domains()

Returns the name of the domains sharing this memory.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access this memory. The order of the domain names is not specified.

Example

Example B-145

```
string domains[];
ral_model.block_name.mem1.get_domains(domains)
foreach (domains[i]) begin
    $display("Domain Name: %s\n", domains[i]);
end
```

vmm_ral_mem::get_block()

Returns the block that instantiates this memory.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

OpenVera

```
virtual function ram_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the memory corresponding to the descriptor instance.

Example

Example B-146

```
vmm_ral_block blk;
blk = ral_model.block_name.mem1.get_block();
blk.display();
```

vmm_ral_mem::get_offset_in_block()

Returns the address of a memory location within the block address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_block(
   input bit [63:0] mem_addr = 0,
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_offset_in_block(
   bit [63:0] mem_addr = 0,
   string     domain = "")
```

Description

Returns the address of the specified location in the memory within the overall address space of the block that instantiates it. By default, returns the base address of the memory. If the memory is shared between multiple physical interfaces, a domain must be specified.

If the memory location is wider than the physical interface of the block, the lowest address value is returned.

Example

Example B-147

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.mem1.get_offset_in_block();
  addr =
ral_model.block_name_shared.mem2.get_offset_in_block("d1")
; //multiple domain block
```

vmm_ral_mem::get_address_in_system()

Returns the address of a memory location within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
   input bit [63:0] mem_addr = 0,
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_address_in_system(
   bit [63:0] mem_addr = 0,
   string     domain = "")
```

Description

Returns the address of the specified location in the memory within the overall address space of the design. By default, returns the base address of the memory. If the memory is shared between multiple physical interfaces, a domain must be specified.

If the memory location is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-148

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.mem1.get_address_in_system();
  addr =
ral_model.block_name_shared.mem2.get_address_in_system("d1
"); //multiple domain block
```

vmm_ral_mem::get_size()

Returns the number of unique locations in this memory.

SystemVerilog

```
virtual function int unsigned get_size()
```

OpenVera

```
virtual function integer get_size()
```

Description

Returns the number of unique memory locations in this memory.

Example

Example B-149

```
int size;
size = ral_model.block_name.mem1.get_size();
```

vmm_ral_mem::get_n_bits()

Returns the number of bits in each memory location.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

OpenVera

```
virtual function integer get_n_bits()
```

Description

Returns the width, in number of bits, of each memory location in the memory.

Example

Example B-150

```
int width;
width = ral_model.block_name.mem1.get_n_bits();
```

vmm_ral_mem::get_n_bytes()

Returns the number of bits in each memory location.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

Description

Returns the width, in number of bits, of each memory location within the memory. If the number of bits is not a multiple of eight, some bits in the most significant byte are not implemented.

Example

Example B-151

```
int size;
size = ral_model.block_name.mem1.get_n_bytes();
```

vmm_ral_mem::get_access()

Returns the access mode of the memory.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_access(strong domain
= "")
```

Description

Returns the specification of the behavior of the memory when written and read. If the memory is shared in more than one domain, a domain name must be specified.

If access restrictions are present when accessing a memory through the specified domain, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through a domain with read-only restrictions would return `vmm_ral::RO`.

Example

Example B-152

```
class my_mem extends vmm_ral_mem;
   ...
   vmm_ral::access_e access;
   access = this.get_access();
   `vmm_log(log,$psprintf("Memory Access is
%0s",access.name);
```

```
    ...
endclass
```

vmm_ral_mem::get_rights()

Returns the access rights of the memory.

SystemVerilog

```
virtual function vmm_ral::access_e get_rights(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_rights(string domain
= "")
```

Description

Returns the access rights of a memory. Returns `vmm_ral::RW`, `vmm_ral::RO`, or `vmm_ral::WO`. The access rights of a memory is always `vmm_ral::RW`, unless it is a shared memory with access restrictions in a particular domain.

If the memory is shared in more than one domain, a domain name must be specified. If the memory is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-153

```
  vmm_ral::access_e rights;
  rights = ral_model.block_name.mem1.get_rights();
  rights =
ral_model.block_name_shared.mem2.get_rights("d1"); //
rights of shared memory mem1 in domain "d1"
```

vmm_ral_mem::get_virtual_fields()

Returns all virtual fields implemented in this memory.

SystemVerilog

```
virtual function void get_virtual_fields(
   ref vmm_ral_vfield fields[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields implemented in this memory. The order in which the fields are located in the array is not specified.

Example

Example B-154

```
vmm_ral_mem::get_virtual_fields() [page 351]

  vmm_ral_vfield vfields[];
  ral_model.memory_name.get_virtual_fields(vfields);
  foreach (vfields[i]) begin
     vfields[i].display();
  end
```

vmm_ral_mem::get_virtual_field_by_name()

Returns the virtual field with the specified name in this memory.

SystemVerilog

```
virtual function vmm_ral_vfield get_virtual_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name implemented in this memory and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within virtual registers. Therefore, if used on a memory implementing more than one field having the same name, this method returns the first field found.

Example

Example B-155

```
vmm_ral_mem::get_virtual_field_by_name() [page 352]

  vmm_ral_vfield vfield;
  vfield =
ral_model.memory_name.get_virtual_field_by_name("virtual_f
ield_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_mem::get_virtual_registers()

Returns all virtual registers in this memory.

SystemVerilog

```
virtual function void get_virtual_registers(
   ref vmm_ral_vreg regs[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual registers implemented in this memory. The order in which the registers are located in the array is not specified.

Example

Example B-156

```
vmm_ral_mem::get_virtual_registers() [page 353]

  vmm_ral_vreg vregs[];
  ral_model.memory_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_ral_mem::get_vreg_by_name()

Returns the virtual register with the specified name in this memory.

SystemVerilog

```
virtual function vmm_ral_vreg get_vreg_by_name(
   string name)
```

Description

Finds a virtual register with the specified name implemented in this memory and returns its descriptor. If no registers are found, returns null.

Example

Example B-157

```
vmm_ral_mem::get_vreg_by_name() [page 354]

  vmm_ral_vreg vreg;
  vreg =
ral_model.memory_name.get_vreg_by_name("virtual_reg_name")
;
  if (vreg == null) `vmm_error(log, "specified virtual
register doesn't exists");
```

vmm_ral_mem::display()

Displays a description of the memory to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the "vmm_ral_mem::psdisplay()" method to the standard output.

Example

Example B-158

```
ral_model.block_name.mem1.display();
```

vmm_ral_mem::psdisplay()

Creates a human-readable description of the memory.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the memory. Each line of the description is prefixed with the specified prefix.

The content of the memory is not displayed.

Example

Example B-159

```
 `vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.mem1.psdisplay()));
```

vmm_ral_mem::psdisplay_domain()

Creates a human-readable description of the memory.

SystemVerilog

```
virtualfunctionstringpsdisplay_domain(stringprefix="",
                                      string domain = "")
```

OpenVera

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

Description

Creates a human-readable description of the memory. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the base address of the memory within that domain is used.

The content of the memory is not displayed.

Example

Example B-160

```
`vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.mem1.psdisplay_domain()));
```

vmm_ral_mem::set_frontdoor()

Defines a user-defined access mechanism for this memory.

SystemVerilog

```
function void set_frontdoor(vmm_ral_mem_frontdoor ftdr,
                            string domain = "")
```

OpenVera

```
task set_frontdoor(vmm_ral_mem_frontdoor ftdr,
                   string                domain = "")
```

Description

By default, memories are mapped linearly into the address space of the block that instantiates them. If memories are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding memory abstraction class.

See "User-Defined Memory Access" on page 116 for an example.

vmm_ral_mem::get_frontdoor()

Returns the user-defined access mechanism for this memory.

SystemVerilog

```
function vmm_ral_mem_frontdoor get_frontdoor(
   string domain = "")
```

OpenVera

```
function vmm_ral_mem_frontdoor get_frontdoor(
   string domain = "")
```

Description

Returns the current user-defined mechanism for this memory for the specified domain. If `null`, no user-defined mechanism has been defined. A user-defined mechanism is defined by using the "`vmm_ral_mem::set_frontdoor()`" method.

Example

Example B-161

```
class my_mem extends vmm_ral_mem;
   ...
   if(this.get_frontdoor == null);
     `vmm_note(log,"set_frontdoor method is not called for
this memory.");
   ...
endclass
```

vmm_ral_mem::set_backdoor()

Defines the back-door access mechanism for this memory.

SystemVerilog

```
function void set_backdoor(vmm_ral_mem_backdoor bkdr)
```

OpenVera

```
task set_backdoor(vmm_ral_mem_backdoor bkdr)
```

Description

Memories implemented using SystemVerilog unpacked arrays can be accessed using a hierarchical path. This direct back-door access is automatically generated if the necessary `hdl_path` properties are specified in the RALF description.

However, memories can be modeled using other methods, such as using DesignWare models. These memory models come with their own back-door mechanism. This method is used to associate a back-door access mechanism with a memory descriptor to enable back-door access.

A class extension implementing the back-door mechanism for DesignWare memory models has already been defined and is included in RAL.

Example

Example B-162

```
class my_block extends vmm_ral_block;
   ...
```

```
   rand my_mem mem;
   function new(...);
      super.new(...);
      mem.new(...);
      begin
        ral_mem_bkdr bkdr = new;
        this.mem.set_backdoor(bkdr);
      end
   endfunction
   ...
endclass
```

vmm_ral_mem::get_backdoor()

Returns the back-door access mechanism for this memory.

SystemVerilog

```
function vmm_ral_mem_backdoor get_backdoor()
```

OpenVera

```
function vmm_ral_mem_backdoor get_backdoor()
```

Description

Returns the current back-door mechanism for this memory. If `null`, no back-door mechanism has been defined. A back-door mechanism can be automatically defined by using the `hdl_path` properties in the RALF definition or user-defined using the "`vmm_ral_mem::set_backdoor()`" method.

Example

Example B-163

```
my_mem.write(0, 16'hABCD,
             (my_mem.get_backdoor() == null) ?
                vmm_rw::BFM : vmm_rw::BACKDOOR);
```

vmm_ral_mem::init()

Initializes the memory.

SystemVerilog

```
virtual task init(
   output bit                  is_ok,
   input  vmm_ral_mem::init_e pattern,
   input  bit [63:0]          data = 0)
```

OpenVera

```
virtual function bit init_t(
   vmm_ral_mem::init_e pattern,
   bit [63:0]          data = 0)
```

Description

Not yet implemented.

Initializes the memory and its mirrored content with the specified pattern. Requires that a back-door access to the memory be available. See "vmm_ral_mem::init_e" for a description of the available initialization patterns.

Returns TRUE if the initialization was successful. Returns FALSE otherwise.

vmm_ral_mem::init_e

Symbolic values identifying the pattern with which to initialize a memory.

SystemVerilog

```
vmm_ral_mem::UNKNOWNS
vmm_ral_mem::ZEROES
vmm_ral_mem::ONES
vmm_ral_mem::VALUE
vmm_ral_mem::INCR
vmm_ral_mem::DECR
```

OpenVera

```
vmm_ral_mem::UNKNOWNS
vmm_ral_mem::ZEROES
vmm_ral_mem::ONES
vmm_ral_mem::VALUE
vmm_ral_mem::INCR
vmm_ral_mem::DECR
```

Description

See "`vmm_ral_mem::init()`" for the method that uses these symbolic values. Each symbolic value is used to define a specific memory initialization pattern as follows:

`vmm_ral_mem::UNKNOWNS`

Initializes the memory with all unknowns ('bx).

`vmm_ral_mem::ZEROES`

Initializes the memory with all zeroes ('b0).

`vmm_ral_mem::ONES`

Initializes the memory with all ones (’b1).

`vmm_ral_mem::VALUE`

Initializes the memory with a specified value.

`vmm_ral_mem::INCR`

Initializes the memory, starting with a specified value and increasing it for each location toward higher addresses.

`vmm_ral_mem::DECR`

Initializes the memory, starting with a specified value and decreasing it for each location toward lower addresses.

vmm_ral_mem::read()

Reads a memory location from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]       mem_addr,
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory location from the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

Memories are not mirrored. Instead, use the "`vmm_ral_mem::peek()`" method.

Example

Example B-164

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.mem1.read(status,0,data);
```

vmm_ral_mem::write()

Sets a memory location in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       mem_addr,
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified value at the specified memory location in the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

Memories are not mirrored. Instead, use the "`vmm_ral_mem::poke()`" method.

Example

Example B-165

```
vmm_rw::status_e status;
ral_model.block_name.mem1.write(status,0,'hABCD);
```

vmm_ral_mem::burst_read()

Perform a burst-read operation on the memory.

SystemVerilog

```
virtual task burst_read(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   output bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer          stream_id = -1)
```

Description

Burst-read the current values of the memory locations specified by the burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the "vmm_ral_access::burst_read()" method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

It is not possible to perform a burst-read operation on a memory instantiated in a block or system with a narrower data path.

Memories are not mirrored. Instead, use the "vmm_ral_mem::peek()" method.

Example

Example B-166

```
class my_ral_mem_burst extends vmm_ral_mem_burst;
   ...
   constraint my_cons {
     n_beats      == 5;
     start_offset == 64'h00;
     incr_offset  == 64'h10;
     max_offset   == 64'h79;
   }
   ...
endclass
...
class my_ral_mem_frontdoor extends vmm_ral_mem_frontdoor;
   ...
   virtual task burst_read(...);
      ...
      //Needs to overwrite
   endtask
   ...
endclass
```

```
...
begin
   ...
   my_ral_mem_frontdoor ftdr = new;
   this.my_mem.set_frontdoor(ftdr);
   ...
end
...
my_ral_mem_burst br;
...
fork
  ral_model.my_mem.burst_read(status,br,data);
join_none
```

vmm_ral_mem::burst_write()

Perform a burst-write operation on the memory.

SystemVerilog

```
virtual task burst_write(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   input  bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer          stream_id = -1)
```

Description

Burst-write the specified values in the memory locations specified by burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the "vmm_ral_access::burst_write()" method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

It is not possible to perform a burst-write operation on a memory instantiated in a block or system with a narrower data path.

Memories are not mirrored. Instead, use the "vmm_ral_mem::poke()" method.

Example

Example B-167

```
class my_ral_mem_frontdoor extends vmm_ral_mem_frontdoor;
   ...
   virtual task burst_write(...);
      ...
      //Needs to overwrite
   endtask
   ...
endclass
...
my_ral_mem_burst br;
...
data  =  new[br.n_beats];
data[0] = 64'h54;
...
fork
  ral_model.my_mem.burst_write(status,br,data);
join_none
...
```

Note that while using the object model for burst operations, addressability is limited to offsets of memory words.Therefore, a random address cannot be chosen as the start of the burst. In a scenario where a burst has to be initiated from a random address, use `"vmm_ral_access::burst_write()"` method or `"vmm_ral_access::burst_read()"` method.

vmm_ral_mem::peek()

Peek a memory location from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   bit [63:0]       mem_addr,
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory location from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-168

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.mem1.peek(status,0,data);
```

vmm_ral_mem::poke()

Poke a memory location in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e poke_t(
   bit [63:0]       mem_addr,
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value at the specified memory location in the design using a back-door access. Depending on the design model implementation, it may be possible to modify the content of a read-only memory.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-169

```
vmm_rw::status_e status;
ral_model.block_name.mem1.poke(status,0,'hABCD);
```

vmm_ral_mem::readmemh()

Initializes the memory.

SystemVerilog

```
virtual task readmemh(string filename)
```

OpenVera

```
virtual task readmemh_t(string filename)
```

Description

Not yet implemented.

Initializes the content of all memory locations using the values in the specified file. The values are updated using the default access path. See "vmm_ral_mem::writememh()" for details.

The format of the file is the same as the one used by the `$readmemh` task.

vmm_ral_mem::writememh()

Dumps the value of the memory.

SystemVerilog

```
virtual task writememh(string filename)
```

OpenVera

```
virtual task writememh_t(string filename)
```

Description

Not yet implemented.

Dumps the content of all memory locations to the specified file. The file can then be used as an input for the "vmm_ral_mem::readmemh()" method. The values are obtained using the default access path.

The format of the file is the same as the one used by the `$readmemh` task.

vmm_ral_mem::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task append_callback(
   vmm_ral_mem_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this memory descriptor. Callbacks are invoked in the order of registration.

Example

Example B-170

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.append_callback(cb);
```

```
      ...
   end
...
endprogram
```

vmm_ral_mem::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task prepend_callback(
   vmm_ral_mem_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this memory descriptor. Callbacks are invoked in the reverse order of registration.

Example

Example B-171

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.prepend_callback(cb);
```

```
      ...
   end
...
endprogram
```

vmm_ral_mem::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task unregister_callback(
   vmm_ral_mem_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this memory descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-172

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.append_callback(cb);
```

```
      //Can't call second time for same instance.
     env.ral_model.my_mem.append_callback(cb);  //Wrong Way
      ...
      env.ral_model.my_mem.unregister_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_mem::set_attribute()

Set an attribute for a memory.

SystemVerilog

```
virtual function void set_attribute(string name,
    string value);
```

OpenVera

```
Not supported
```

Description

Set the specified attribute to the specified value for this memory. If the value is specified as "", the specified attribute is deleted.

A warning is issued if an existing attribute is modified.

Attribute names are case sensitive.

vmm_ral_mem::get_attribute()

Get an attribute for a memory.

SystemVerilog

```
virtual function string get_attribute(string name,
    bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Get the value of the specified attribute for this memory. If the attribute does not exists, "" is returned.

If the "inherited" argument is specifed as TRUE, the value of the attribute is inherited from the nearest enclosing block or system if it is not specified for this memory. If it is specified as FALSE, the value "" is returned if it does not exists in the this memory.

Attribute names are case sensitive.

vmm_ral_mem::get_all_attributes()

Get all attributes for a memory.

SystemVerilog

```
virtual function void get_all_attributes(
     ref string names[],
     input bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Return an array filled with the name of the attributes defined for this memory.

If the "inherited" argument is specifed as TRUE, the value of all attributes inherited from the enclosing block and system(s) is included. If the argument is specified as FALSE, only the attributed defined for this memory are returned.

The order in which attribute names are returned is not specified.

vmm_ral_mem::ral_power_down()

Specify that a memory is powered down.

SystemVerilog

```
virtual function void ral_power_down();
```

OpenVera

```
Not supported
```

Description

Specify that this memory has been put in a power-saving state. A read or write access to any location inside the memory will result in a run-time error message and a vmm_ral::ERROR status code.

This method does not physically power down the memory.Iit only indicates to the RAL model that the memory has been powered down.

vmm_ral_mem::ral_power_up()

Specify that a memory is powered back up.

SystemVerilog

```
virtual function void ral_power_up();
```

OpenVera

```
Not supported
```

Description

Specify that this memory has been a restored to a powered-up state.

This method does not physically power up the memory.Iit only indicates to the RAL model that the memory has been powered up.

vmm_ral_mem_backdoor

Virtual base class for back-door access to memories. Extensions of this class are automatically generated by RAL if full hierarchical paths to memories are specified through the `hdl_path` properties in RALF descriptions.

Can be extended by users to provide user-specific back-door access to memories that are not implemented in pure SystemVerilog.

Summary

- vmm_ral_mem_backdoor::read() ..... page B-302
- vmm_ral_mem_backdoor::write() ..... page B-304
- vmm_ral_mem_backdoor::pre_write() ..... page B-306
- vmm_ral_mem_backdoor::post_write() ..... page B-308
- vmm_ral_mem_backdoor::pre_read() ..... page B-309
- vmm_ral_mem_backdoor::post_read() ..... page B-310

vmm_ral_mem_backdoor::read()

Peek a memory location.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]     offset,
   var bit [63:0] data,
   integer        data_id,
   integer        scenario_id,
   integer        stream_id)
```

Description

Peek the current value at the specified offset in the memory corresponding to the instance of this class. Returns the content of the memory location and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_mem::read()" method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

Ideally, the execution of this method should be non-blocking.

See "Implementing a Memory Backdoor in SystemVerilog" on page 100 for an example.

vmm_ral_mem_backdoor::write()

Poke a memory location.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  offset,
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Deposit the specified value at the specified offset of the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_mem::write()" method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

Ideally, the execution of this method should be non-blocking.

If it is not possible to deposit the specified value, for example, a ROM is implemented using constants, a vmm_rw::ERROR status should be returned.

See "Implementing a Memory Backdoor in SystemVerilog" on page 100 for an example.

vmm_ral_mem_backdoor::pre_write()

Invokes all the registered `vmm_ral_mem_backdoor_callbacks::pre_write(…)` and `vmm_ral_mem_backdoor_callbacks::encode(…)` callback methods.

SystemVerilog

```
virtual function void pre_write(inout bit
[`VMM_RAL_ADDR_WIDTH-1:0] offset,
                 inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                 input int data_id = -1,
                 input int scenario_id = -1,
                 input int stream_id = -1);
```

OpenVera

```
virtual task pre_write_t(var bit [VMM_RAL_ADDR_WIDTH-1:0]
offset,
              var bit [VMM_RAL_DATA_WIDTH-1:0] data,
              integer data_id = -1,
              integer scenario_id = -1,
              integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_mem_backdoor_callbacks::pre_write()` methods in the order of their registration. And then, it invokes all the registered `vmm_ral_mem_backdoor_callbacks::encode()` methods in the order of their registration.

This method should be invoked just before executing any backdoor write operation. `ralgen` generated auto backdoor read/write code takes care of this by invoking this method just before writing any data

through backdoor. User defined backdoors should take care of this. The `data` to be written, if modified by this method modifies the actual value that is written. The `offset` if modified by this method modifies the actual memory location i.e. the `offset` that is written.

vmm_ral_mem_backdoor::post_write()

Invokes all the registered `vmm_ral_mem_backdoor_callbacks::post_write(…)` callback methods.

SystemVerilog

```
virtual function void post_write(inout vmm_rw::status_e
status,
              inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
              input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
              input int data_id = -1,
              input int scenario_id = -1,
              input int stream_id = -1);
```

OpenVera

```
virtual task post_write_t(var vmm_rw::status_e status,
              var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
              bit [VMM_RAL_DATA_WIDTH-1:0] data,
              integer data_id = -1,
              integer scenario_id = -1,
              integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_mem_backdoor_callbacks::post_write()` methods in the order of their registration.

This method should be invoked just after executing any backdoor write operation. `ralgen` generated auto backdoor read/write code takes care of this by invoking this method just after writing any data through backdoor. User defined backdoors should take care of this.

vmm_ral_mem_backdoor::pre_read()

Invokes all the registered `vmm_ral_mem_backdoor_callbacks::pre_read(…)` methods.

SystemVerilog

```
virtual task pre_read(input vmm_ral_mem mem,
               inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
               input int data_id = -1,
               input int scenario_id = -1,
               input int stream_id = -1);
```

OpenVera

```
virtual task pre_read_t(var bit [VMM_RAL_ADDR_WIDTH-1:0]
offset,
                    integer data_id = -1,
                    integer scenario_id = -1,
                    integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_mem_backdoor_callbacks::pre_read()` methods in the order of their registration.

This method should be invoked just before executing any backdoor read operation. `ralgen` generated auto backdoor read/write code takes care of this by invoking this method just before reading any data through backdoor. User defined backdoors should take care of this. The `offset`, if modified by this method modifies the actual memory location i.e. the offset that is read.

vmm_ral_mem_backdoor::post_read()

Invokes all the registered `vmm_ral_mem_backdoor_callbacks::post_read(…)` and `vmm_ral_mem_backdoor_callbacks::decode(…)` methods.

SystemVerilog

```
virtual task post_read(input vmm_ral_mem mem,
               inout vmm_rw::status_e status,
               inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
               inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
               input int data_id = -1,
               input int scenario_id = -1,
               input int stream_id = -1);
```

OpenVera

```
virtual task post_read_t(var vmm_rw::status_e status,
                var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
                var bit [VMM_RAL_DATA_WIDTH-1:0] data,
                integer data_id = -1,
                integer scenario_id = -1,
                integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_mem_backdoor_callbacks::decode()` methods in the reverse order of their registration. And then, it invokes all the registered `vmm_ral_mem_backdoor_callbacks::post_read()` methods in the order of their registration.

This method should be invoked just after executing any backdoor read operation. `ralgen` generated auto backdoor read/write code takes care of this by invoking this method just after reading any data through backdoor. User defined backdoors should take care of this.

vmm_ral_mem_burst

Descriptor for memory burst read/write operation.

Summary

- vmm_ral_mem_burst::n_beats ..... page B-313
- vmm_ral_mem_burst::start_offset ..... page B-315
- vmm_ral_mem_burst::incr_offset ..... page B-316
- vmm_ral_mem_burst::max_offset ..... page B-317
- vmm_ral_mem_burst::user_data ..... page B-318

vmm_ral_mem_burst::n_beats

Length of the burst.

SystemVerilog

```
rand int n_beats
```

OpenVera

```
rand integer n_beats
```

Description

Specifies the number of beats or transfers in a memory burst operation.

Example

Example B-173

```
class tb_env extends vmm_ral_env;
   ...
   vmm_ral_mem_burst cfg;
   ...
   function new();
      super.new();
      this.cfg = new();
      ...
   endfunction

   virtual function void gen_cfg();
      super.gen_cfg();
      if (!this.cfg.randomize()) begin
         `vmm_fatal(log, "Failed to randomize Memory Burst
configuration");
      end
      ...
```

```
   endfunction
      `vmm_note(log,$psprintf("vmm_ral_mem_burst::n_beats
==> %0d",cfg.n_beats));
   ...
endclass
```

vmm_ral_mem_burst::start_offset

Starting memory offset.

SystemVerilog

```
rand bit [63:0] start_offset
```

OpenVera

```
rand bit [63:0] start_offset
```

Description

Specifies the first offset that is the target of a memory burst read or write operation.

Example

Example B-174

```
 ...
 `vmm_note(log,$psprintf("vmm_ral_mem_burst::start_offset
==> %0d",
            cfg.start_offset));
   ...
```

vmm_ral_mem_burst::incr_offset

Offset increment between beats.

SystemVerilog

```
rand bit [63:0] incr_offset
```

OpenVera

```
rand bit [63:0] incr_offset
```

Description

Memory location offset increment between individual beats of a burst read or write operation. The first beat reads or writes the memory location specified by "vmm_ral_mem_burst::start_offset". The nth beat reads or writes the memory location specified by "vmm_ral_mem_burst::start_offset" + (n-1) "vmm_ral_mem_burst::incr_offset".

A value of 0 implies a burst operation that repeatedly accesses the same memory location.

Example

Example B-175

```
...
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::incr_offset
==> %0d",
            cfg.incr_offset));
   ...
```

vmm_ral_mem_burst::max_offset

Maximum offset for the burst.

SystemVerilog

```
rand bit [63:0] max_offset
```

OpenVera

```
rand bit [63:0] max_offset
```

Description

Limits the memory location offset range of a burst operation. Causes wrapping if subsequent beats would access a location past the specified offset.

The specified maximum offset must be a valid memory offset.

Example

Example B-176

```
   ...
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::max_offset
==> %0d",
            cfg.max_offset));
   ...
```

Note that all the offsets correspond to offsets of memory words and not the address offset.

vmm_ral_mem_burst::user_data

Additional burst configuration information.

SystemVerilog

```
vmm_data user_data
```

OpenVera

```
rvm_data user_data
```

Description

Provides a mechanism for passing additional burst configuration information to the "vmm_rw_xactor::execute_burst()" method that will ultimately execute the physical burst operation. Any reference to additional user information is passed through transparently through "vmm_rw_burst::user_data". The additional information can be recovered by using `$cast()` or `cast_assign()`.

Example

Example B-177

```
  class my_ral_mem_burst extends vmm_ral_mem_burst;
   ...
   vmm_data user_data;
   ...
endclass
...
my_ral_mem_burst br;
...
```

vmm_ral_mem_callbacks

Memory descriptors.

Summary

- vmm_ral_mem_callbacks::pre_write() ..... page B-320
- vmm_ral_mem_callbacks::post_write() ..... page B-322
- vmm_ral_mem_callbacks::pre_read() ..... page B-324
- vmm_ral_mem_callbacks::post_read() ..... page B-326
- vmm_ral_mem_callbacks::pre_burst() ..... page B-328
- vmm_ral_mem_callbacks::post_burst() ..... page B-329

vmm_ral_mem_callbacks::pre_write()

OOP callback invoked before writing a memory location.

SystemVerilog

```
virtual task pre_write(vmm_ral_mem         mem,
                       ref bit [63:0]      offset,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                      ref string         domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_mem         mem,
                         var bit [63:0]      offset,
                         var bit [63:0]      wdat,
                         var vmm_ral::path_e path,
                        var string         domain)
```

Description

This callback method is invoked before a value is written to a memory location in the DUT. The written value, if modified, modifies the actual value that will be written. The address of the location within the memory, as well as the path and domain used to write to it can also be modified.

Example

Example B-178

```
...
fork
  //It will call pre_write() and post_write() callback
methods if status IS_OK.
  this.ral_model.my_mem.write(status,addr,data);
join_none
```

```
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
     virtual task pre_write(vmm_ral_mem                mem,
                   ref bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                     ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                       ref vmm_ral::path_e             path,
                       ref string                     domain
         `vmm_note(log,{"pre_write method is called for
vmm_ral_mem_callbacks",
                    " class"});
      endtask: pre_write
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::post_write()

OOP callback invoked after writing a memory location.

SystemVerilog

```
virtual task post_write(vmm_ral_mem           mem,
                        bit [63:0]            offset,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_mem           mem,
                          bit [63:0]            offset,
                          bit [63:0]            wdat,
                          vmm_ral::path_e       path,
                          string                domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully written to a memory location in the DUT. The `wdat` value is the value that was attempted to be written to the memory location, not necessarily the current value of that memory location. If a physical write access did not return `vmm_rw::IS_OK`, this method is not called.

Example

Example B-179

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
```

```
     virtual task post_write(vmm_ral_mem                mem,
                        bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                          bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                            vmm_ral::path_e             path,
                            string                    domain,
                         ref vmm_rw::status_e        status);
         `vmm_note(log,{"post_write method is called for
vmm_ral_mem_callbacks",
                    " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::pre_read()

OOP callback invoked before reading a memory location.

SystemVerilog

```
virtual task pre_read(vmm_ral_mem         mem,
                      ref bit [63:0]      offset,
                      ref vmm_ral::path_e path,
                     ref string         domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_field       field,
                        var bit [63:0]      offset,
                        var vmm_ral::path_e path,
                       var string         domain)
```

Description

This callback method is invoked before a value is read from a memory location in the DUT. The address of the location in the memory, as well as the path and domain used to read from it, can be modified.

Example

Example B-180

```
...
fork
  begin
    //It will call pre_write() and post_write() callback
methods if status IS_OK.
    this.ral_model.my_mem.write(status,addr,data);
    write_done = 1'b1;
  end
```

```
  begin
    wait(write_done);
   //It will call pre_read() and post_read() callback methods
if status IS_OK.
    this.ral_model.my_mem.read(status,addr,read_data);
    `vmm_note(log,$psprintf({"Read Data ==> %0h and Status
==> %0s",
              read_data,status.name));
    write_done = 1'b0;
  end
join_none
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
     virtual task pre_read(vmm_ral_mem                 mem,
                   ref bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                       ref vmm_ral::path_e            path,
                         ref string                     dom
         `vmm_note(log,{"pre_read method is called for
vmm_mem_reg_callbacks",
                    " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::post_read()

OOP callback invoked after reading a memory location.

SystemVerilog

```
virtual task post_read(input vmm_ral_mem       mem,
                       input bit [63:0]        offset,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain,
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_mem           mem,
                         bit [63:0]            offset,
                         var bit [63:0]        rdat,
                         vmm_ral::path_e       path,
                         string                domain,
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully read from a memory location in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the "`vmm_ral_mem::read()`" method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

Example

Example B-181

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
```

```
      ...
    virtual task post_read(input vmm_ral_mem            mem,
                 input bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                   ref  bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                     input vmm_ral::path_e           path,
                     input string                  domain,
                    ref  vmm_rw::status_e         status);
         `vmm_note(log,{"post_read method is called for
vmm_ral_mem_callbacks",
                    " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::pre_burst()

OOP callback invoked before a burst operation.

SystemVerilog

```
virtual task pre_burst(vmm_ral_mem          mem,
                       vmm_rw::kind_e       kind,
                       vmm_ral_mem_burst    burst,
                       ref bit [63:0]       wdata[],
                       ref vmm_ral::path_e  path,
                      ref string          domain)
```

OpenVera

```
virtual task pre_burst_t(vmm_ral_field        field,
                         vmm_rw::kind_e       kind,
                         vmm_ral_mem_burst    burst,
                         var bit [63:0]       wdata[*],
                         var vmm_ral::path_e  path,
                        var string          domain)
```

Description

This callback method is invoked before a burst operation is performed. The description of the burst area, any value to be written (if it is a burst-write operation), as well as the path and domain used to perform the operation, can be modified.

vmm_ral_mem_callbacks::post_burst()

OOP callback invoked after a burst operation.

SystemVerilog

```
virtual task post_burst(input vmm_ral_mem       mem,
                        input vmm_rw::kind_e    kind,
                        input vmm_ral_mem_burst burst,
                        ref   bit [63:0]        data[],
                        input vmm_ral::path_e   path,
                        input string            domain,
                       ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_burst_t(    vmm_ral_field     field,
                              vmm_rw::kind_e    kind,
                              vmm_ral_mem_burst burst,
                          var bit [63:0]        data[*],
                          var vmm_ral::path_e   path,
                              string            domain,
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a burst operation is performed. The values read (if it is a burst-read operation), as well as the status the operation, can be modified.

vmm_ral_mem_backdoor::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void
prepend_callback(vmm_ral_mem_backdoor_callbacks cb,
                        string fname = "",
                        int lineno = 0);
```

OpenVera

```
task prepend_callback(vmm_ral_mem_backdoor_callbacks cb);
```

Description

Prepends the specified callback extension instance to the registered callbacks for this memory backdoor access class. Callbacks are invoked in the reverse order of registration.

Example

```
program test;
...

class my_ral_mem_backdoor_callbacks extends
vmm_ral_mem_backdoor_callbacks;
...
endclass

my_ral_mem_backdoor_callbacks cb;
...
initial
   begin
      ...
      cb = new();
```

```
env.ral_model.my_mem.get_backdoor().prepend_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_mem_backdoor::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void
unregister_callback(vmm_ral_mem_backdoor_callbacks cb,
                            string fname = "",
                            int lineno = 0);
```

OpenVera

```
task unregister_callback(vmm_ral_mem_backdoor_callbacks
cb);
```

Description

Removes the specified callback extension instance from the registered callbacks for this memory backdoor access class. A warning message is issued if the callback instance has not been previously stored.

Example

```
program test;
...

class my_ral_mem_backdoor_callbacks extends
vmm_ral_mem_backdoor_callbacks;
...
endclass

my_ral_mem_backdoor_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();

env.ral_model.my_mem.get_backdoor().append_callback(cb);
      //Can't call second time for same instance.

env.ral_model.my_mem.get_backdoor().append_callback(cb);
//Wrong Way
      ...

env.ral_model.my_mem.get_backdoor().unregister_callback(cb
);
      ...
   end
...
endprogram
```

vmm_ral_mem_backdoor_callbacks

Façade class for memory backdoor access callback methods.

Summary

- vmm_ral_mem_backdoor_callbacks::pre_write() ..... page B-335
- vmm_ral_mem_backdoor_callbacks::post_write() ..... page B-336
- vmm_ral_mem_backdoor_callbacks::pre_read() ..... page B-337
- vmm_ral_mem_backdoor_callbacks::post_read() ..... page B-338
- vmm_ral_mem_backdoor_callbacks::encode() ..... page B-339
- vmm_ral_mem_backdoor_callbacks::decode() ..... page B-340

vmm_ral_mem_backdoor_callbacks::pre_write()

OOP callback invoked before writing a memory through backdoor.

SystemVerilog

```
virtual task pre_write(input vmm_ral_mem mem,
             inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
             inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
             input int data_id = -1,
             input int scenario_id = -1,
             input int stream_id= -1);
```

OpenVera

```
v irtual task pre_write_t(vmm_ral_mem mem,
             var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
             var bit [VMM_RAL_DATA_WIDTH-1:0] data,
             integer data_id = -1,
             integer scenario_id = -1,
             integer stream_id= -1);
```

Description

This callback method is invoked before a value is written to a memory through backdoor. The written value, if modified, modifies the actual value that will be written.

This callback method is invoked when the `vmm_ral_mem_backdoor::pre_write()` is invoked.

vmm_ral_mem_backdoor_callbacks::post_write()

OOP callback invoked after writing a memory through backdoor.

SystemVerilog

```
virtual task post_write(input vmm_ral_mem mem,
                   inout vmm_rw::status_e status,
                inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                  input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                   input int data_id = -1,
                   input int scenario_id = -1,
                   input int stream_id = -1)
```

OpenVera

```
virtual task post_write_t(vmm_ral_mem mem,
                       var vmm_rw::status_e status,
                    var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
                       bit [VMM_RAL_DATA_WIDTH-1:0] data,
                       integer data_id = -1,
                       integer scenario_id = -1,
                       integer stream_id = -1);
```

Description

This callback method is invoked after a value is written to a memory through backdoor.

This callback method is invoked when the `vmm_ral_mem_backdoor::post_write()` method is invoked.

vmm_ral_mem_backdoor_callbacks::pre_read()

OOP callback invoked before reading a memory through backdoor.

SystemVerilog

```
virtual task pre_read(input vmm_ral_mem mem,
               inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
               input int data_id = -1,
               input int scenario_id = -1,
               input int stream_id = -1);
```

OpenVera

```
virtual task pre_read_t(vmm_ral_mem mem,
                   var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This callback method is invoked before a value is read from a memory through backdoor.

This callback method is invoked when the `vmm_ral_mem_backdoor::pre_read()` method is invoked.

vmm_ral_mem_backdoor_callbacks::post_read()

OOP callback invoked after reading a memory through backdoor.

SystemVerilog

```
virtual task post_read(input vmm_ral_mem mem,
                            inout vmm_rw::status_e status,
                 inout bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                   inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                            input int data_id = -1,
                            input int scenario_id = -1,
                            input int stream_id = -1);;
```

OpenVera

```
virtual task post_read_t(vmm_ral_mem mem,
                      var vmm_rw::status_e status,
                   var bit [VMM_RAL_ADDR_WIDTH-1:0] offset,
                     var bit [VMM_RAL_DATA_WIDTH-1:0] data,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This callback method is invoked after a value is read from a memory through backdoor.

This callback method is invoked when the `vmm_ral_mem_backdoor::post_read()` method is invoked.

vmm_ral_mem_backdoor_callbacks::encode()

OOP callback method invoked before writing a memory through backdoor, but after all the registered `vmm_ral_mem_backdoor_callbacks::pre_write()` methods have been called.

SystemVerilog

```
virtual function bit [`VMM_RAL_DATA_WIDTH-1:0]
encode(bit [`VMM_RAL_DATA_WIDTH-1:0] data);
```

OpenVera

```
function bit [VMM_RAL_DATA_WIDTH-1:0]
encode(bit [VMM_RAL_DATA_WIDTH-1:0] data);
```

Description

For a `vmm_ral_mem_backdoor` class instance, all the registered `vmm_ral_mem_backdoor_callbacks::encode()` methods will be invoked in the order of registration after all the registered `vmm_ral_mem_backdoor_callbacks::pre_write()` methods have been called.

This callback method is invoked when the `vmm_ral_mem_backdoor::pre_write()` method is invoked.

vmm_ral_mem_backdoor_callbacks::decode()

OOP callback invoked after reading a memory through backdoor, but before any of the registered `vmm_ral_mem_backdoor_callbacks::post_read()` method is called.

SystemVerilog

```
virtual function bit [`VMM_RAL_DATA_WIDTH-1:0]
decode(bit [`VMM_RAL_DATA_WIDTH-1:0] data);
```

OpenVera

```
virtual function bit [VMM_RAL_DATA_WIDTH-1:0]
decode(bit [VMM_RAL_DATA_WIDTH-1:0] data);
```

Description

For a `vmm_ral_mem_backdoor` class instance, all the registered `vmm_ral_mem_backdoor_callbacks::decode()` methods will be invoked in the reverse order of registration before any of the registered `vmm_ral_mem_backdoor_callbacks::post_read()` method is called.

This callback method is invoked when the `vmm_ral_mem_backdoor::post_read()` method is invoked.

vmm_ral_mem_frontdoor

Virtual base class for user-defined access to memories through a physical interface.

By default, the entire address space of a memory is mapped within the address space of the block instantiating it. Different memory locations are accessed using different addresses. If the memory is physically accessed using a non-linear, non-mapped mechanism, this base class must be user-extended to provide the physical access to the memory. See "User-Defined Register Access" on page 114 for an example.

Summary

- vmm_ral_mem_frontdoor::read() ..... page B-342
- vmm_ral_mem_frontdoor::write() ..... page B-344
- vmm_ral_mem_frontdoor::burst_read() ..... page B-346
- vmm_ral_mem_frontdoor::burst_write() ..... page B-348

vmm_ral_mem_frontdoor::read()

Performs a physical read to a memory location.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]      offset,
   var bit [63:0]  data,
   integer         data_id,
   integer         scenario_id,
   integer        stream_id)
```

Description

Performs a physical read access to the specified offset in the memory corresponding to the instance of this class. Returns the content of the memory location and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_mem::read()" method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_mem_frontdoor::write()

Performs a physical write to a memory location.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  offset,
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Performs a physical write access to the specified offset of the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_mem::write()" method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_mem_frontdoor::burst_read()

Performs a physical burst-read operation to a memory.

SystemVerilog

```
virtual task burst_read(output vmm_rw::status_e  status,
                        input  vmm_ral_mem_burst burst,
                        output bit [63:0]        data[],
                        input  int               data_id,
                       input  int              scenario_id,
                        input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e burst_read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    data[*],
   integer           data_id,
   integer           scenario_id,
   integer          stream_id)
```

Description

Performs a physical burst-read access on the specified locations defined by the burst descriptor in the memory corresponding to the instance of this class. Returns the content of the memory locations and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "`vmm_ral_mem::burst_read()`" method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

vmm_ral_mem_frontdoor::burst_write()

Performs a physical burst-write operation to a memory.

SystemVerilog

```
virtual task burst_write(output vmm_rw::status_e  status,
                         input  vmm_ral_mem_burst burst,
                         input  bit [63:0]        data[],
                         input  int               data_id,
                       input  int              scenario_id,
                        input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e burst_write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        data[*],
   integer           data_id,
   integer           scenario_id,
   integer          stream_id)
```

Description

Performs a physical burst-write access on the specified locations defined by the burst descriptor in the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_mem::burst_write()" method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_reg

Base class for register descriptors.

Summary

- vmm_ral_reg::log ..... page B-351
- vmm_ral_reg::get_name() ..... page B-352
- vmm_ral_reg::get_fullname() ..... page B-353
- vmm_ral_reg::get_n_domains() ..... page B-354
- vmm_ral_reg::get_domains() ..... page B-355
- vmm_ral_reg::get_rights() ..... page B-356
- vmm_ral_reg::get_block() ..... page B-357
- vmm_ral_reg::get_offset_in_block() ..... page B-358
- vmm_ral_reg::get_address_in_system() ..... page B-359
- vmm_ral_reg::get_n_bytes() ..... page B-360
- vmm_ral_reg::get_constraints() ..... page B-361
- vmm_ral_reg::display() ..... page B-363
- vmm_ral_reg::psdisplay() ..... page B-364
- vmm_ral_reg::psdisplay_domain() ..... page B-365
- vmm_ral_reg::get_fields() ..... page B-366
- vmm_ral_reg::get_field_by_name() ..... page B-367
- vmm_ral_reg::set_frontdoor() ..... page B-368
- vmm_ral_reg::get_frontdoor() ..... page B-369
- vmm_ral_reg::set_backdoor() ..... page B-370
- vmm_ral_reg::get_backdoor() ..... page B-371
- vmm_ral_reg::set() ..... page B-372
- vmm_ral_reg::predict() ..... page B-374
- vmm_ral_reg::get() ..... page B-376
- vmm_ral_reg::reset() ..... page B-377
- vmm_ral_reg::sample_field_values() ..... page B-378
- vmm_ral_reg::needs_update() ..... page B-379
- vmm_ral_reg::update() ..... page B-380
- vmm_ral_reg::mirror() ..... page B-382
- vmm_ral_reg::read() ..... page B-384
- vmm_ral_reg::write() ..... page B-386
- vmm_ral_reg::peek() ..... page B-388
- vmm_ral_reg::poke() ..... page B-390
- vmm_ral_reg::append_callback() ..... page B-392
- vmm_ral_reg::prepend_callback() ..... page B-394
- vmm_ral_reg::unregister_callback() ..... page B-396
- vmm_ral_reg::get_reset() ..... page B-398
- vmm_ral_reg::set_attribute() ..... page B-399
- vmm_ral_reg::get_attribute() ..... page B-400
- vmm_ral_reg::get_all_attributes() ..... page B-401

vmm_ral_reg::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the register descriptor. A single message service interface instance is shared by all register abstraction class instances.

Example

Example B-182

```
class my_ral_reg extends vmm_ral_reg;
   ...
   vmm_log log;
   function new(...);
      ...
      log = new("ral_reg","log");
    `vmm_note(log,"vmm_log instance is successfully created
for vmm_ral_reg");
   endfunction
   ...
endclass
```

vmm_ral_reg::get_name()

Returns the name of the register.

SystemVerilog

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

OpenVera

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

Description

Returns the name of the register corresponding to the instance of the descriptor.

Example

Example B-183

```
  vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      `vmm_note(log, $psprintf("Register Name: %s\n",
regs[i].get_name()));
  end
```

vmm_ral_reg::get_fullname()

Returns the fully-qualified name of the register.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the register corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-184

```
  vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      `vmm_note(log, $psprintf("Register Name: %s\n",
regs[i].get_fullname()));
  end
```

vmm_ral_reg::get_n_domains()

Returns the number of domains sharing this register.

SystemVerilog

```
function int get_n_domains()
```

OpenVera

```
function integer get_n_domains()
```

Description

Returns the number of domains that share this register. You can obtain the name of the domains with the "vmm_ral_reg::get_domains()" method.

Example

Example B-185

```
  int domain_number;
  domain_number =
ral_model.block_name.reg1.get_n_domains();
```

vmm_ral_reg::get_domains()

Returns the name of the domains sharing this register.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access this register. The order of the domain names is not specified.

Example

Example B-186

```
string domains[];
  ral_model.block_name.reg1.get_domains(domains)
  foreach (domains[i]) begin
      $display("Domain Name: %s\n", domains[i]);
  end
```

vmm_ral_reg::get_rights()

Returns the access rights of this register.

SystemVerilog

```
function vmm_ral::access_e get_rights(string domain = "")
```

OpenVera

```
function vmm_ral::access_e get_rights(string domain = "")
```

Description

Returns the access rights of a register. Returns `vmm_ral::RW`, `vmm_ral::RO`, or `vmm_ral::WO`. The access rights of a register is always `vmm_ral::RW`, unless it is a shared register with access restriction in a particular domain.

If the register is shared in more than one domain, a domain name must be specified. If the register is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-187

```
  vmm_ral::access_e rights;
  rights = ral_model.block_name.reg1.get_rights();
  rights =
ral_model.block_name_shared.reg2.get_rights("d1"); //
rights of shared register reg1 in domain "d1"
```

vmm_ral_reg::get_block()

Returns the block that instantiates this register.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

OpenVera

```
virtual function vmm_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the register corresponding to the descriptor instance.

Example

Example B-188

```
vmm_ral_block blk;
blk = ral_model.block_name.reg1.get_block();
blk.display();
```

vmm_ral_reg::get_offset_in_block()

Returns the address of the register within the block address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_block(
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_offset_in_block(
   string     domain = "")
```

Description

Returns the address of the register in the overall address space of the block that instantiates it. If the register is shared between multiple physical interfaces, a domain must be specified.

If the register is wider than the physical interface of the block, the lowest address value is returned.

Example

Example B-189

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.reg1.get_offset_in_block();
  addr =
ral_model.block_name_shared.reg2.get_offset_in_block("d1")
; //multiple domain block
```

vmm_ral_reg::get_address_in_system()

Returns the address of the register within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_address_in_system(
   string     domain = "")
```

Description

Returns the address of the register in the overall address space of the design. If the register is shared between multiple physical interfaces, you must specify a domain.

If the register is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-190

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.reg1.get_address_in_system();
  addr =
ral_model.block_name_shared.reg2.get_address_in_system("d1
"); //multiple domain block
```

vmm_ral_reg::get_n_bytes()

Returns the width of the register.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

OpenVera

```
virtual function integer get_n_bytes()
```

Description

Returns the width, in number of bytes, of the register.

Example

Example B-191

```
integer bytes;
bytes = ral_model.block_name.reg1.get_n_bytes();
```

vmm_ral_reg::get_constraints()

Returns the constraint blocks in this register.

SystemVerilog

```
virtual function void get_constraints(
   ref string names[])
```

OpenVera

```
virtual task get_constraints(
   var string names[*])
```

Description

Fills the specified dynamic array with the names of the constraint blocks in this register. The constraint blocks in the fields within this register are specified in constraint blocks with the same name as the field name. The location of each constraint block name in the array is not defined.

Example

Example B-192

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand vmm_ral_field MINFL;
   ...
   constraint MINFL_spec {
      MINFL.value == 'h40;
   }
   ..
endclass
...
my_ral_reg my_reg;
```

```
...
string str[];
...
this.ral_model.my_reg.get_constraints(str);
foreach (str[i])
 `vmm_note(log,$psprintf("Constraint Name is %0s",str[i]));
...
```

vmm_ral_reg::display()

Displays a description of the register to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the "vmm_ral_reg::psdisplay()" method to the standard output.

Example

Example B-193

```
ral_model.block_name.reg1.display();
```

vmm_ral_reg::psdisplay()

Creates a human-readable description of the register.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

Example

Example B-194

```
 `vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.reg1.psdisplay()));
```

vmm_ral_reg::psdisplay_domain()

Creates a human-readable description of the register.

SystemVerilog

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

OpenVera

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the address of the register within that domain is used.

Example

Example B-195

```
`vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.reg1.psdisplay_domain()));
```

vmm_ral_reg::get_fields()

Returns all fields in this register.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_field fields[])
```

OpenVera

```
virtual task get_fields(
   var vmm_ral_field fields[*])
```

Description

Fills the specified dynamic array with the descriptor for all of the fields contained in the register. Fields are ordered from least-significant position to most-significant position within the register.

Example

Example B-196

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
```

vmm_ral_reg::get_field_by_name()

Returns the field with the specified name in this register.

SystemVerilog

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

Description

Finds a field with the specified name in the register and returns its descriptor. If no fields are found, returns null.

Example

Example B-197

```
  vmm_ral_field field;
  field =
ral_model.block_name.reg1.get_field_by_name("field_f1");
 if (field == null) `vmm_error(log, "specified field doesn't
exists");
```

vmm_ral_reg::set_frontdoor()

Defines a user-defined access mechanism for this register.

SystemVerilog

```
function void set_frontdoor(vmm_ral_reg_frontdoor ftdr,
                            string domain = "")
```

OpenVera

```
task set_frontdoor(vmm_ral_reg_frontdoor ftdr,
                   string                domain = "")
```

Description

By default, registers are mapped linearly into the address space of the block that instantiates them. If registers are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding register abstraction class.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_reg::get_frontdoor()

Returns the user-defined access mechanism for this register.

SystemVerilog

```
function vmm_ral_reg_frontdoor get_frontdoor(
   string domain = "")
```

OpenVera

```
function vmm_ral_reg_frontdoor get_frontdoor(
   string domain = "")
```

Description

Returns the current user-defined mechanism for this register for the specified domain. If `null`, no user-defined mechanism has been defined. A user-defined mechanism is defined by using the "vmm_ral_reg::set_frontdoor()" method.

Example

Example B-198

```
...
my_ral_reg my_reg;
...
vmm_ral::path_e path;
...
my_ral_reg_frontdoor ftdr = new();
this.my_reg.set_frontdoor(ftdr);
...
if(my_reg.get_frontdoor() == null)
  `vmm_note(log,"Register FrontDoor is not set.");
...
```

vmm_ral_reg::set_backdoor()

Defines the back-door access mechanism for this register.

SystemVerilog

```
virtual function void set_backdoor(
   vmm_ral_reg_backdoor bkdr)
```

OpenVera

```
virtual task set_backdoor(
   vmm_ral_reg_backdoor bkdr)
```

Description

Registers implemented using SystemVerilog variables can be accessed using a hierarchical path. This direct back-door access is automatically generated if the necessary `hdl_path` properties are specified in the RALF description.

However, registers can be modeled using other methods, or be included in imported models written in different languages. This method is used to associate a back-door access mechanism with a register descriptor to enable back-door accesses.

vmm_ral_reg::get_backdoor()

Returns the back-door access mechanism for this register.

SystemVerilog

```
virtual function vmm_ral_reg_backdoor get_backdoor()
```

OpenVera

```
virtual function vmm_ral_reg_backdoor get_backdoor()
```

Description

Returns the current back-door mechanism for this register. If `null`, no back-door mechanism has been defined. A back-door mechanism can be automatically defined by using the `hdl_path` properties in the RALF definition or user-defined using the "`vmm_ral_reg::set_backdoor()`" method.

Example

Example B-199

```
my_reg.write(16'hABCD,
             (my_reg.get_backdoor() == null) ?
                vmm_rw::BFM : vmm_rw::BACKDOOR);
```

vmm_ral_reg::set()

Sets the mirror value of a register.

SystemVerilog

```
virtual function void set(
   bit [63:0] value)
```

OpenVera

```
virtual task set(
   bit [63:0] value)
```

Description

Sets the mirror value of the fields in the register to the specified value. Does not actually set the value of the register in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the "vmm_ral_reg::update()" method to update the actual register with the mirrored value or the "vmm_ral_reg::write()" method to set the actual register and its mirrored value.

See "vmm_ral_field::set()" on page 144 for more information on the effect of setting mirror values on fields with different access modes.

To modify the mirrored field values to a specific value, regardless of the access modes—and thus use the RAL mirror as a scoreboard for the register values in the DUT—use the "vmm_ral_reg::predict()" method.

Example

Example B-200

```
ral_model.block_name.reg1.set(0);
```

vmm_ral_reg::predict()

Force the mirror value of the register.

SystemVerilog

```
virtual function bit predict (bit [63:0] value,
                              string fname = "",
                              int lineno = 0,
                              bit force_predict = 0)
```

OpenVera

```
virtual function bit predict(bit [63:0] value)
```

Description

Force the mirror value of the fields in the register to the specified value. Does not actually force the value of the fields in the design, only the value mirrored in their corresponding descriptor in the RAL model. Use the "vmm_ral_reg::update()" method to update the actual register with the mirrored value or the "vmm_ral_reg::write()" method to set the register and its mirrored value.

The final value in the mirror is the specified value, regardless of the access mode of the fields in the register. For example, the mirrored value of a read-only field **is** modified by this method, and the mirrored value of a read-update field can be updated to any value predicted to correspond to the value in the corresponding physical bits in the design. By default, `predict` does not allow any update of the mirror, when RAL is busy executing a transaction on this register. However, if need be, that can be overridden by setting the `force_predict` argument to 1.

Returns FALSE if this method is called while the register is being read or written, thus rendering the prediction unreliable. Returns TRUE otherwise.

Example

Example B-201

```
...
   this.ral_model.my_reg.predict(data);
   `vmm_note(log,$psprintf("Forced Value ==> %0h",data));
   `vmm_note(log,$psprintf("Previous Value==>
%h",this.ral_model.my_reg.get()));

   this.ral_model.my_reg.update();
   `vmm_note(log,$psprintf("Updated Value==>
%h",this.ral_model.my_reg.get()));
   ...
```

vmm_ral_reg::get()

Returns the mirror value of a register.

SystemVerilog

```
virtual function bit [63:0] get()
```

OpenVera

```
virtual function bit [63:0] get()
```

Description

Returns the mirror value of the fields in the register. Does not actually read the value of the register in the design, only the value mirrored in its corresponding descriptor in the RAL model.

If the register contains write-only fields, the mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would return zeroes for these fields, the returned mirrored value is the actual content.

Use the "vmm_ral_reg::read()" method to get the actual register value.

Example

Example B-202

```
bit [63:0] data;
data = ral_model.block_name.reg1.get();
```

vmm_ral_reg::reset()

Resets the mirror value of the register.

SystemVerilog

```
virtual function void reset(vmm_ral::reset_e kind =
vmm_ral::HARD)
```

OpenVera

```
virtual task reset(vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of the fields in the register to the specified reset value. Does not actually reset the value of the register in the design, only the value mirrored in the descriptor in the RAL model.

Write-once fields in the register can be modified after a hard reset operation.

Example

Example B-203

```
ral_model.block_name.reg1.reset();
```

vmm_ral_reg::sample_field_values()

Samples the field values within this register.

SystemVerilog

```
virtual function sample_field_values()
```

OpenVera

```
virtual function sample_field_values()
```

Description

By using this function, you can sample the field value coverage within the RAL registers.

With this method, you will be able to sample field values within the RAL register itself which would sample field coverage for all the fields within the register by calling `field_values.sample()` for the register.

Example

Example B-204

```
class my_reg_cb extends vmm_ral_reg_callbacks;
  task post_write(vmm_ral_reg rg, bit [`VMM_RAL_DATA_WIDTH-
1:0] wdat, vmm_ral::path_e path, string domain, ref
vmm_rw::status_e status);
            rg.sample_field_values();
   endtask
endclass
```

vmm_ral_reg::needs_update()

Queries if a mirrored value in this register has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If a mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the register is outdated. This method returns TRUE if the state of the register needs to be updated to match the mirrored values (or vice-versa).

The mirror values or actual content of registers are not modified. See "vmm_ral_reg::update()" on page 271 or "vmm_ral_reg::mirror()" on page 272 for more details.

Example

Example B-205

```
bit update;
vmm_rw::status_e status;
update = ral_model.block_name.reg1.needs_update();
if (update == 1) begin
   ral_model.block_name.reg1.update(status);
end
```

vmm_ral_reg::update()

Updates the physical register to match mirrored values in this register descriptor.

SystemVerilog

```
virtual task update(
   output vmm_rw::status_e status,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string          domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e update_t(
   vmm_ral::path_e path = vmm_ral::DEFAULT,
   string         domain = "")
```

Description

Updates the content of the register in the design to match the mirrored value, if it has been modified using one of the `set()` methods to a different value. The update can be performed using the physical interfaces (frontdoor) or "vmm_ral_reg::poke()" (backdoor). If the register is shared across multiple physical interfaces and physical access is used (front-door access), a domain must be specified.

This method performs the reverse operation of "vmm_ral_reg::mirror()" on page 272.

Example

Example B-206

```
bit update;
```

```
vmm_rw::status_e status;
update = ral_model.block_name.reg1.needs_update();
if (update == 1) begin
   ral_model.block_name.reg1.update(status);
end
```

vmm_ral_reg::mirror()

Updates the mirrored value of the register descriptor to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e  status,
   input  vmm_ral::check_e  check = vmm_ral::QUIET,
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string           domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e   path  = vmm_ral::DEFAULT,
   string           domain = "")
```

Description

Updates the content of the register mirror value to match the corresponding value in the design. The mirroring can be performed using the physical interfaces (frontdoor) or "vmm_ral_reg::peek()" (backdoor). If the `check` argument is specified as `vmm_ral::VERB`, an error message is issued if the current mirrored value does not match the actual value in the design. If the register is shared across multiple physical interfaces and physical access is used (front-door access), a domain must be specified.

If the register contains write-only fields, their content is mirrored and optionally checked only if a `vmm_ral::BACKDOOR` access path is used to read the register.

This method performs the reverse operation of "vmm_ral_reg::update()".

Example

Example B-207

```
vmm_rw::status_e status;
ral_model.block_name.reg1.mirror(status);
```

vmm_ral_reg::read()

Reads a register from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the register from the design using the specified access path. If the register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the register through a physical access is mimicked. For example, clear-on-read bits in the registers will be cleared.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register is updated with the value read from the design. The mirrored value of any write-only field in the register is updated only if an `vmm_ral::BACKDOOR` access path is used.

Example

Example B-208

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.reg1.read(status,data);
```

vmm_ral_reg::write()

Sets a register in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified value in the register in the design using the specified access path. If the register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the registers will not be written.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

If the register is written using a physical (front-door) path and it contains write-once fields, it is not possible to modify the content of these fields, either through a physical or back-door access.

The mirrored value of the register location is updated based on the written value and the specified behavior of the various fields after a write operation.

Example

Example B-209

```
vmm_rw::status_e status;
ral_model.block_name.reg1.write(status,'hABCD);
```

vmm_ral_reg::peek()

Peek a register from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the register from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register is updated with the value read from the design.

Example

Example B-210

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.reg1.peek(status,data);
```

vmm_ral_reg::poke()

Poke a register in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value in the register in the design, as-is, using a back-door access. See "vmm_ral_field::poke()" for a description of the effect on field values.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register location is updated based on the written value.

Example

Example B-211

```
vmm_rw::status_e status;
ral_model.block_name.reg1.poke(status,'hABCD);
```

vmm_ral_reg::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task append_callback(
   vmm_ral_reg_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the order of registration.

Note that the corresponding "vmm_ral_field" callback methods will be invoked before the register callback methods.

Example

Example B-212

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
```

```
   begin
      ...
      cb = new();
      env.ral_model.my_reg.append_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task prepend_callback(
   vmm_ral_reg_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the reverse order of registration.

Note that the corresponding "vmm_ral_field" callback methods will be invoked before the register callback methods.

Example

Example B-213

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
```

```
   begin
      ...
      cb = new();
      env.ral_model.my_reg.prepend_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task unregister_callback(
   vmm_ral_reg_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this register descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-214

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_reg.append_callback(cb);
```

```
      //Can't call second time for same instance.
     env.ral_model.my_reg.append_callback(cb);  //Wrong Way
      ...
      env.ral_model.my_reg.unregister_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg::get_reset()

Get the reset value for a register.

SystemVerilog

```
virtual function bit [63:0] get_reset(
     vmm_ral::reset_e kind = vmm_ral::HARD);
```

OpenVera

```
Not supported
```

Description

Return the specified reset value for the register.

vmm_ral_reg::set_attribute()

Set an attribute for a register.

SystemVerilog

```
virtual function void set_attribute(string name,
    string value);
```

OpenVera

```
Not supported
```

Description

Set the specified attribute to the specified value for this register. If the value is specified as "", the specified attribute is deleted.

A warning is issued if an existing attribute is modified.

Attribute names are case sensitive.

vmm_ral_reg::get_attribute()

Get an attribute for a register.

SystemVerilog

```
virtual function string get_attribute(string name,
    bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Get the value of the specified attribute for this register. If the attribute does not exists, "" is returned.

If the "inherited" argument is specifed as TRUE, the value of the attribute is inherited from the nearest enclosing block or system if it is not specified for this register. If it is specified as FALSE, the value "" is returned if it does not exists in the this register.

Attribute names are case sensitive.

vmm_ral_reg::get_all_attributes()

Get all attributes for a register.

SystemVerilog

```
virtual function void get_all_attributes(
     ref string names[],
     input bit inherited = 1);
```

OpenVera

```
Not supported
```

Description

Return an array filled with the name of the attributes defined for this register.

If the "inherited" argument is specifed as TRUE, the value of all attributes inherited from the enclosing block and system(s) is included. If the argument is specified as FALSE, only the attributed defined for this register are returned.

The order in which attribute names are returned is not specified.

vmm_ral_reg_backdoor

Virtual base class for back-door access to registers. Extensions of this class are automatically generated by RAL if full hierarchical paths to registers are specified through the `hdl_path` properties in RALF descriptions.

Can be extended by users to provide user-specific back-door access to registers that are not implemented in pure SystemVerilog.

Summary

- vmm_ral_reg_backdoor::read() ..... page B-403
- vmm_ral_reg_backdoor::write() ..... page B-405
- vmm_ral_reg_backdoor::pre_write() ..... page B-407
- vmm_ral_reg_backdoor::post_write() ..... page B-409
- vmm_ral_reg_backdoor::pre_read() ..... page B-410
- vmm_ral_reg_backdoor::post_read() ..... page B-411

vmm_ral_reg_backdoor::read()

Peek a register.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0] data,
   integer    data_id,
   integer    scenario_id,
   integer    stream_id)
```

Description

Peek the current value of the register corresponding to the instance of this class. Returns the content of the register and an indication of the success of the operation.

The value of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_reg::read()" or "vmm_ral_field::read()" method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

The execution of this method should ideally be non-blocking.

See "Implementing a Register Backdoor in SystemVerilog" on page 91 for an example.

vmm_ral_reg_backdoor::write()

Poke a register.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0] data,
   integer    data_id,
   integer    scenario_id,
   integer   stream_id)
```

Description

Deposit the specified value in the register corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_reg::write()" or "vmm_ral_field::write()" method call that requires the back-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

The execution of this method should ideally be non-blocking.

If the bits in the register cannot be forced to the specified value, for example, read-only bits are implemented as constants, a vmm_rw::ERROR status should be returned.

See "Implementing a Register Backdoor in SystemVerilog" on page 91 for an example.

vmm_ral_reg_backdoor::pre_write()

Invokes all the registered `vmm_ral_reg_backdoor_callbacks::pre_write(…)` and `vmm_ral_reg_backdoor_callbacks::encode(…)` callback methods.

SystemVerilog

```
virtual function void pre_write(inout bit
[`VMM_RAL_DATA_WIDTH-1:0] data,
                 input int data_id = -1,
                 input int scenario_id = -1,
                 input int stream_id = -1);
```

OpenVera

```
   virtual task pre_write_t(
                  var bit [VMM_RAL_DATA_WIDTH-1:0] data,
                  integer data_id = -1,
                  integer scenario_id = -1,
                  integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_reg_backdoor_callbacks::pre_write()` methods in the order of their registration. And then, it invokes all the registered `vmm_ral_reg_backdoor_callbacks::encode()` methods in the order of their registration.

Ideally, you should invoke this method before executing any backdoor write operation. `ralgen` generated auto backdoor read/write code takes care of this by invoking this method just before

writing any data through backdoor. User defined backdoors should take care of this. The 'data' to be written, if modified by this method, modifies the actual value that is written.

vmm_ral_reg_backdoor::post_write()

Invokes all the registered `vmm_ral_reg_backdoor_callbacks::post_write(…)` callback methods.

SystemVerilog

```
virtual function void post_write(inout vmm_rw::status_e
status,
                  input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                        input int data_id = -1,
                        input int scenario_id = -1,
                        input int stream_id = -1);
```

OpenVera

```
virtual task post_write_t(var vmm_rw::status_e status,
                      bit [VMM_RAL_DATA_WIDTH-1:0] data,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_reg_backdoor_callbacks::post_write()` methods in the order of their registration.

This method should be invoked just after executing any backdoor write operation. ralgen generated auto backdoor read/write code takes care of this, by invoking this method just after writing any data through backdoor. User defined backdoors will need to take care of this.

vmm_ral_reg_backdoor::pre_read()

Invokes all the registered `vmm_ral_reg_backdoor_callbacks::pre_read(…)` callback methods.

SystemVerilog

```
virtual function void pre_read(input int data_id = -1,
                      input int scenario_id = -1,
                              input int stream_id = -1);
```

OpenVera

```
virtual task pre_read_t(integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This method invokes all the registered `vmm_ral_reg_backdoor_callbacks::pre_read()` methods in the order of their registration.

This method should be invoked just before executing any backdoor read operation. `ralgen` generated auto backdoor read/write code takes care of this, by invoking this method just before reading any data through backdoor. User defined backdoors will need to take care of this.

vmm_ral_reg_backdoor::post_read()

Invokes all registered `vmm_ral_reg_backdoor_callbacks::post_read(…)` and `vmm_ral_reg_backdoor_callbacks::decode(…)` callback methods.

SystemVerilog

```
virtual function void post_read(inout vmm_rw::status_e
status,
                  inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                      input int data_id = -1,
                      input int scenario_id = -1,
                      input int stream_id = -1);
```

OpenVera

```
virtual task post_read_t(var vmm_rw::status_e status,
                     var bit [VMM_RAL_DATA_WIDTH-1:0] data,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This method invokes all registered `vmm_ral_reg_backdoor_callbacks::decode()` methods in the reverse order of their registration. And after that, it invokes all the registered `vmm_ral_reg_backdoor_callbacks::post_read()` methods in the order of their registration.

This method should be invoked just after executing any backdoor read operation. `ralgen` generated auto backdoor read/write code takes care of this, by invoking this method just after reading any data through backdoor. User defined backdoors will need to take care of this.

vmm_ral_reg_callbacks

Base class for register descriptors.

Summary

- vmm_ral_reg_callbacks::pre_write() ..... page B-414
- vmm_ral_reg_callbacks::post_write() ..... page B-416
- vmm_ral_reg_callbacks::pre_read() ..... page B-418
- vmm_ral_reg_callbacks::post_read() ..... page B-421

vmm_ral_reg_callbacks::pre_write()

OOP callback invoked before writing a register.

SystemVerilog

```
virtual task pre_write(vmm_ral_reg          rg,
                       ref bit [63:0]       wdat,
                       ref vmm_ral::path_e  path,
                      ref string          domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_reg          rg,
                         var bit [63:0]       wdat,
                         var vmm_ral::path_e  path,
                        var string          domain)
```

Description

This callback method is invoked before a value is written to a register in the DUT. The written value, if modified, changes the actual value that is written. The path and domain used to write to the register can also be modified.

This callback method is only invoked when the "vmm_ral_reg::write()" or "vmm_ral_field::write()" method is used to write to the register inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "vmm_ral_reg::set()" method.

Because writing a register causes all of the fields it contains to be written, all registered "vmm_ral_field_callbacks::pre_write()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-215

```
...
fork
  //It will call pre_write() and post_write() callback
methods.
  this.ral_model.my_reg.write(status,data);
join_none
...
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
     virtual task pre_write(vmm_ral_reg                 rg,
                     ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                        ref vmm_ral::path_e            path,
                       ref string                  domain);
         `vmm_note(log,{"pre_write method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask: pre_write
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::post_write()

OOP callback invoked after writing a register.

SystemVerilog

```
virtual task post_write(vmm_ral_reg           rg,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_reg           rg,
                          bit [63:0]            wdat,
                          vmm_ral::path_e       path,
                          string                domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully written to a register in the DUT. The `wdat` value is the final mirrored value in the register as reported by the "`vmm_ral_reg::get()`" method. If a physical write access did not return vmm_rw::IS_OK, this method is not called.

This callback method is only invoked when the "`vmm_ral_reg::write()`" or "`vmm_ral_field::write()`" method is used to write to the register inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "`vmm_ral_reg::set()`" method.

Because writing a register causes all of the fields it contains to be written, all registered "vmm_ral_field_callbacks::post_write()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-216

```
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
      virtual task post_write(vmm_ral_reg                rg,
                              bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                               vmm_ral::path_e              path,
                              string                     domain,
                             ref vmm_rw::status_e        status);
         `vmm_note(log,{"post_write method is called for
vmm_ral_reg_callbacks",
                    " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::pre_read()

OOP callback invoked before reading a register.

SystemVerilog

```
virtual task pre_read(vmm_ral_reg         rg,
                      ref vmm_ral::path_e path,
                     ref string         domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_reg         rg,
                        var vmm_ral::path_e path,
                       var string         domain)
```

Description

This callback method is invoked before a value is read from a register in the DUT. You can modify the path and domain used to read the register.

This callback method is only invoked when the "vmm_ral_reg::read()" or the "vmm_ral_field::read()" method is used to read from the register inside the DUT. This callback method is not invoked when only the mirrored value is read using the "vmm_ral_reg::get()" method.

Because reading a register causes all of the fields it contains to be read, all registered "vmm_ral_field_callbacks::pre_read()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-217

```
...
fork
  begin
    //It will call pre_write() and post_write() callback
methods.
    this.ral_model.my_reg.write(status,data);
    write_done = 1'b1;
  end

  begin
    wait(write_done);
    //It will call pre_read() and post_read() callback
methods.
    this.ral_model.my_reg.read(status,read_data);
    `vmm_note(log,$psprintf({"Read Data ==> %0h and Status
==> %0s",
              read_data,status.name));
    write_done = 1'b0;
  end
join_none
...
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
      virtual task pre_read(vmm_ral_reg       rg,
                            ref vmm_ral::path_e  path,
                            ref string           domain);
         `vmm_note(log,{"pre_read method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::post_read()

OOP callback invoked after reading a register.

SystemVerilog

```
virtual task post_read(input vmm_ral_reg        rg,
                       ref   bit [63:0]         rdat,
                       input vmm_ral::path_e    path,
                       input string             domain
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_reg            rg,
                         var bit [63:0]         rdat,
                         vmm_ral::path_e        path,
                         string                 domain
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully read from a register in the DUT. The `rdat` and `status` values are the values that will be ultimately returned by the "`vmm_ral_reg::read()`" method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is invoked only when the "`vmm_ral_reg::read()`" or "`vmm_ral_field::read()`" method is used to read from the register inside the DUT. This callback method is not invoked when only the mirrored value is read from using the "`vmm_ral_reg::get()`" method.

Because reading a register causes all of the fields it contains to be read, all registered "vmm_ral_field_callbacks::post_read()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-218

```
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
     virtual task post_read(vmm_ral_reg                rg,
                     ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                       input vmm_ral::path_e           path,
                       input string                  domain,
                     ref vmm_rw::status_e           status);
         `vmm_note(log,{"post_read method is called for
vmm_ral_reg_callbacks",
                    " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_backdoor::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void
append_callback(vmm_ral_reg_backdoor_callbacks cb,
                string fname = "",
                int lineno = 0);
```

OpenVera

```
task append_callback(vmm_ral_reg_backdoor_callbacks cb);
```

Description

Appends the specified callback extension instance to the list of registered callbacks of this register backdoor access class. Unless you specify explicitly, all the callback methods are invoked in the order of the registration of their corresponding callback class extension instance.

Example

```
program test;
...

class my_ral_reg_backdoor_callbacks extends
vmm_ral_reg_backdoor_callbacks;
...
endclass

my_ral_reg_backdoor_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();

env.ral_model.my_reg.get_backdoor().append_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg_backdoor::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void
prepend_callback(vmm_ral_reg_backdoor_callbacks cb,
                        string fname = "",
                        int lineno = 0);
```

OpenVera

```
task prepend_callback(vmm_ral_reg_backdoor_callbacks cb);
```

Description

Prepends the specified callback extension instance to the list of registered callbacks of this register backdoor access class. Callbacks are invoked in the reverse order of registration.

Example

```
program test;
...

class my_ral_reg_backdoor_callbacks extends
vmm_ral_reg_backdoor_callbacks;
...
endclass

my_ral_reg_backdoor_callbacks cb;
...
initial
   begin
      ...
      cb = new();
```

```
env.ral_model.my_reg.get_backdoor().prepend_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg_backdoor::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void
unregister_callback(vmm_ral_reg_backdoor_callbacks cb,
                            string fname = "",
                            int lineno = 0);
```

OpenVera

```
task unregister_callback(vmm_ral_reg_backdoor_callbacks
cb);
```

Description

Removes the specified callback extension instance from the registered callbacks for this register backdoor access class. A warning message is issued if the callback instance has not been previously registered.

Example

```
program test;
...

class my_ral_reg_backdoor_callbacks extends
vmm_ral_reg_backdoor_callbacks;
...
endclass

my_ral_reg_backdoor_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();

env.ral_model.my_reg.get_backdoor().append_callback(cb);
      //Can't call second time for same instance.

env.ral_model.my_reg.get_backdoor().append_callback(cb);
//Wrong Way
      ...

env.ral_model.my_reg.get_backdoor().unregister_callback(cb
);
      ...
   end
...
endprogram
```

vmm_ral_reg_backdoor_callbacks

Façade class for register backdoor access callback methods.

Summary

- vmm_ral_reg_backdoor_callbacks::pre_write() ..... page B-430
- vmm_ral_reg_backdoor_callbacks::post_write() ..... page B-431
- vmm_ral_reg_backdoor_callbacks::pre_read() ..... page B-432
- vmm_ral_reg_backdoor_callbacks::post_read() ..... page B-433
- vmm_ral_reg_backdoor_callbacks::encode() ..... page B-434
- vmm_ral_reg_backdoor_callbacks::decode() ..... page B-435

vmm_ral_reg_backdoor_callbacks::pre_write()

OOP callback invoked before writing a register through backdoor.

SystemVerilog

```
virtual task
pre_write(input vmm_ral_reg rg,
          inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
          input int data_id = -1,
          input int scenario_id = -1,
          input int stream_id = -1);
```

OpenVera

```
virtual task
pre_write_t(vmm_ral_reg rg,
            var bit [VMM_RAL_DATA_WIDTH-1:0] data,
            integer data_id = -1,
            integer scenario_id = -1,
            integer stream_id = -1);
```

Description

This callback method is invoked before a value is written to a register through backdoor. The written value, if modified, modifies the actual value that will be written.

This callback method is invoked when the `vmm_ral_reg_backdoor::pre_write()` is invoked.

vmm_ral_reg_backdoor_callbacks::post_write()

OOP callback invoked after writing a register through backdoor.

SystemVerilog

```
virtual task post_write(input vmm_ral_reg rg,
                inout vmm_rw::status_e status,
                input bit [`VMM_RAL_DATA_WIDTH-1:0] data,
                input int data_id = -1,
                input int scenario_id = -1,
                input int stream_id = -1);
```

OpenVera

```
virtual task post_write_t(vmm_ral_reg rg,
                      var vmm_rw::status_e status,
                      bit [VMM_RAL_DATA_WIDTH-1:0] data,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This callback method is invoked after a value is written to a register through backdoor.

This callback method is invoked when the `vmm_ral_reg_backdoor::post_write()` method is invoked.

vmm_ral_reg_backdoor_callbacks::pre_read()

OOP callback invoked before reading a register through backdoor.

SystemVerilog

```
virtual task pre_read(input vmm_ral_reg rg,
                      input int data_id = -1,
                      input int scenario_id = -1,
                      input int stream_id = -1);
```

OpenVera

```
virtual task pre_read_t(vmm_ral_reg rg,
                     integer data_id = -1,
                     integer scenario_id = -1,
                     integer stream_id = -1);
```

Description

This callback method is invoked before a value is read from a register through backdoor.

This callback method is invoked when the `vmm_ral_reg_backdoor::pre_read()` method is invoked.

vmm_ral_reg_backdoor_callbacks::post_read()

OOP callback invoked after reading a register through backdoor.

SystemVerilog

```
virtual task post_read(input vmm_ral_reg rg,
             inout vmm_rw::status_e status,
             inout bit [`VMM_RAL_DATA_WIDTH-1:0] data,
             input int data_id = -1,
             input int scenario_id = -1,
             input int stream_id = -1);
```

OpenVera

```
virtual task post_read_t(vmm_ral_reg rg,
                      var vmm_rw::status_e status,
                    var bit [VMM_RAL_DATA_WIDTH-1:0] data,
                      integer data_id = -1,
                      integer scenario_id = -1,
                      integer stream_id = -1);
```

Description

This callback method is invoked after a value is read from a register through backdoor.

This callback method is invoked when the `vmm_ral_reg_backdoor::post_read()` method is invoked.

vmm_ral_reg_backdoor_callbacks::encode()

OOP callback method invoked before writing a register through backdoor, but after all the registered `vmm_ral_reg_backdoor_callbacks::pre_write()` methods have been called.

SystemVerilog

```
virtual function bit [`VMM_RAL_DATA_WIDTH-1:0]
encode(bit [`VMM_RAL_DATA_WIDTH-1:0] data);
```

OpenVera

```
function bit [VMM_RAL_DATA_WIDTH-1:0]
encode(bit [VMM_RAL_DATA_WIDTH-1:0] data);
```

Description

For a `vmm_ral_reg_backdoor` class instance, all the registered `vmm_ral_reg_backdoor_callbacks::encode()` methods will be invoked in the order of registration after all the registered `vmm_ral_reg_backdoor_callbacks::pre_write()` methods have been called.

This callback method is invoked when the `vmm_ral_reg_backdoor::pre_write()` method is invoked.

vmm_ral_reg_backdoor_callbacks::decode()

OOP callback invoked after reading a register through backdoor, but before any of the registered `vmm_ral_reg_backdoor_callbacks::post_read()` method is called.

SystemVerilog

```
virtual function bit [`VMM_RAL_DATA_WIDTH-1:0]
decode(bit [`VMM_RAL_DATA_WIDTH-1:0] data);
```

OpenVera

```
virtual function bit [VMM_RAL_DATA_WIDTH-1:0]
decode(bit [VMM_RAL_DATA_WIDTH-1:0] data);
```

Description

For a `vmm_ral_reg_backdoor` class instance, all the registered `vmm_ral_reg_backdoor_callbacks::decode()` methods will be invoked in the reverse order of registration before any of the `registered vmm_ral_reg_backdoor_callbacks::post_read()` methods is called.

This callback method is invoked when the `vmm_ral_reg_backdoor::post_read()` method is invoked.

vmm_ral_reg_frontdoor

Virtual base class for user-defined access to registers through a physical interface.

By default, different registers are mapped to different addresses in the address space of the block instantiating them. If registers are physically accessed using a non-linear, non-mapped mechanism, this base class must be user-extended to provide the physical access to these registers. See "User-Defined Register Access" on page 114 for an example.

Summary

- vmm_ral_reg_frontdoor::read() ..... page B-437
- vmm_ral_reg_frontdoor::write() ..... page B-439

vmm_ral_reg_frontdoor::read()

Performs a physical register read.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]  data,
   integer         data_id,
   integer         scenario_id,
   integer        stream_id)
```

Description

Performs a physical read access of the register corresponding to the instance of this class. Returns the content of the register and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_reg::read()" method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_reg_frontdoor::write()

Performs a physical write to a register.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Performs a physical write access to the register corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the "vmm_ral_reg::write()" method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See "User-Defined Register Access" on page 114 for an example.

vmm_ral_sys

System descriptor class derived from "vmm_ral_block_or_sys".

Summary

- vmm_ral_sys::new() ..... page B-442
- vmm_ral_sys::get_blocks() ..... page B-444
- vmm_ral_sys::get_all_blocks() ..... page B-446
- vmm_ral_sys::get_block_by_name() ..... page B-448
- vmm_ral_sys::get_subsys() ..... page B-449
- vmm_ral_sys::get_all_subsys() ..... page B-451
- vmm_ral_sys::get_subsys_by_name() ..... page B-453

vmm_ral_sys::new()

Creates an instance of a RAL model.

SystemVerilog

```
function new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

OpenVera

```
task new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

Description

Creates an instance of a RAL model with the corresponding system as the top-level structural element.

The `cover_on` argument specifies the functional coverage models to be enabled in the RAL model. Multiple functional coverage models may be specified by adding their symbolic names. Only functional coverage models that were generated by `ralgen` using the `-c` option can be enabled. Because the functional coverage models affect the memory footprint and runtime performance of a RAL model, they should be enabled only when relevant. See "Predefined Functional Coverage Models" on page 121 for more details.

It is not possible to enable a functional coverage model at a later time, but it is possible to turn the measurement of a functional coverage model off and on using the "vmm_ral_block_or_sys::set_cover()" method.

Example

Example B-219

```
ral_sys_mysys ral_model = new(vmm_ral::ALL_COVERAGE);
```

vmm_ral_sys::get_blocks()

Returns all blocks in system.

SystemVerilog

```
virtual function void get_blocks(
   ref vmm_ral_block blocks[],
   ref string        domains[],
   input string     domain = "")
```

OpenVera

```
virtual task get_blocks(
   var vmm_ral_block blocks[*],
   var string        domains[*],
   string           domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and their domain name for all of the blocks directly instantiated in the system. The order in which the blocks are located in the array is not specified.

The name returned in `domains[i]` corresponds to the domain name within `blocks[i]` that is instantiated in the system.

If a domain name is specified, only the blocks instantiated in the specified domain are included.

The order in which the blocks are located in the array is not specified.

Example

Example B-220

```
  vmm_ral_block blks[];
  string domains [];
  ral_model.get_blocks(blks,domains);
  foreach (blks[i]) begin
    $display(" block name = %s :: domain name =
%s",blks[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_all_blocks()

Returns all blocks in the system and subsystems.

SystemVerilog

```
virtual function void get_all_blocks(
   ref vmm_ral_block blocks[],
   ref string        domains[],
   input string     domain = "")
```

OpenVera

```
virtual task get_all_blocks(
   var vmm_ral_block blocks[*],
   var string        domains[*],
   string           domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the blocks instantiated in the system and subsystems it contains.

The name returned in `domains[i]` corresponds to the domain name within `blocks[i]` that is instantiated in the system.

If a domain is specified, only those blocks accessible in that domain are returned.

The order in which the blocks are located in the array is not specified.

Example

Example B-221

```
vmm_ral_block blks[];
```

```
  string domains [];
  ral_model.get_all_blocks(blks,domains);
  foreach (blks[i]) begin
    $display(" block name = %s :: domain name =
%s",blks[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_block_by_name()

Returns the block with the specified name in this system.

SystemVerilog

```
virtual function vmm_ral_block get_block_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_block get_block_by_name(
   string name)
```

Description

Finds a block with the specified name in the system and returns its descriptor. If no blocks are found, returns null.

Block name uniqueness is guaranteed only within a single system. Therefore, if used on a system with more than one block having the same name but in different subsystems, this method returns the first block found.

Example

Example B-222

```
  vmm_ral_block blk;
  blk = ral_model.get_block_by_name("block1");
 if (blk == null) `vmm_error(log, "specified block doesn't
exists");
```

vmm_ral_sys::get_subsys()

Returns all subsystems in system.

SystemVerilog

```
virtual function void get_subsys(
   ref vmm_ral_sys subsys[],
   ref string      domains[],
   input string   domain = "")
```

OpenVera

```
virtual task get_subsys(
   var vmm_ral_sys subsys[*],
   var string      domains[*],
   string         domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the subsystems directly instantiated in the system.

The name returned in `domains[i]` corresponds to the domain name within `subsys[i]` that is instantiated in the system.

If a domain is specified, only those subsystems accessible in that domain are returned.

The order in which the subsystems are located in the array is not specified.

Example

Example B-223

```
  vmm_ral_sys sub_sys[];
  string domains [];
  ral_model.get_subsys(sub_sys,domains);
  foreach (sub_sys[i]) begin
    $display(" Sub-system name = %s :: domain name =
%s",sub_sys[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_all_subsys()

Returns all subsystems in system and subsystems.

SystemVerilog

```
virtual function void get_all_subsys(
   ref vmm_ral_subsys subsys[],
   ref string         domains[],
   input string      domain = "")
```

OpenVera

```
virtual task get_all_subsys(
   var vmm_ral_sys subsys[*],
   var string      domains[*],
   string         domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the subsystems instantiated in the system and subsystems it contains.

The name returned in `domains[i]` corresponds to the domain name within `subsys[i]` that is instantiated in the system.

If a domain is specified, only those subsystems accessible in that domain are returned.

The order in which the subsystems are located in the array is not specified.

Example

Example B-224

```
  vmm_ral_sys sub_sys[];
  string domains [];
  ral_model.get_all_subsys(sub_sys,domains);
  foreach (sub_sys[i]) begin
    $display(" Sub-system name = %s :: domain name =
%s",sub_sys[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_subsys_by_name()

Returns the subsystem with the specified name in this system.

SystemVerilog

```
virtual function vmm_ral_sys get_subsys_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_sys get_subsys_by_name(
   string name)
```

Description

Finds a subsystem with the specified name in the system and returns its descriptor. If no system is found, returns `null`.

Subsystem name uniqueness is guaranteed only within a single system. Therefore, if used on a system with more than one subsystem having the same name but in different subsystems, this method returns the first subsystem found.

Example

Example B-225

```
vmm_ral_sys sys;
  sys = ral_model.get_subsys_by_name("sys1");
  if (sys == null) `vmm_error(log, "specified subsystem
doesn't exists");
```

vmm_ral_vfield

Field descriptors.

Summary

- vmm_ral_vfield::log ..... page B-455
- vmm_ral_vfield::get_name() ..... page B-456
- vmm_ral_vfield::get_fullname() ..... page B-457
- vmm_ral_vfield::get_register() ..... page B-458
- vmm_ral_vfield::get_lsb_pos_in_register() ..... page B-459
- vmm_ral_vfield::get_n_bits() ..... page B-460
- vmm_ral_vfield::get_access() ..... page B-461
- vmm_ral_vfield::display() ..... page B-463
- vmm_ral_vfield::psdisplay() ..... page B-464
- vmm_ral_vfield::read() ..... page B-465
- vmm_ral_vfield::write() ..... page B-467
- vmm_ral_vfield::peek() ..... page B-469
- vmm_ral_vfield::poke() ..... page B-471
- vmm_ral_vfield::append_callback() ..... page B-473
- vmm_ral_vfield::prepend_callback() ..... page B-474
- vmm_ral_vfield::unregister_callback() ..... page B-475

vmm_ral_vfield::log

Message service interface.

SystemVerilog

```
vmm_log log
```

Description

Message service interface instance for the field descriptor. A single message service interface instance is shared by all virtual field abstraction class instances.

Example

Example B-226

```
  vmm_log log = new("Test", "Main");

 ral_model.status_reg.virtual_field.read(0, status, data);
  if (status != vmm_rw::IS_OK) begin
     `vmm_error(log, "Non-OK status when reading through
virtual field");
  end

  log.report();
```

vmm_ral_vfield::get_name()

Returns the name of the field.

SystemVerilog

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

Description

Returns the name of the field corresponding to the instance of the descriptor.

Example

Example B-227

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Virtual Field Name: %s\n",
vfields[i].get_name()));
  end
```

vmm_ral_vfield::get_fullname()

Returns the fully-qualified name of the field.

SystemVerilog

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the field corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-228

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Virtual Field Hierarchical
Name: %s\n", vfields[i].get_fullname()));
  end
```

vmm_ral_vfield::get_register()

Returns the virtual register that instantiates this field.

SystemVerilog

```
virtual function vmm_ral_vreg get_register()
```

Description

Returns a reference to the descriptor of the virtual register that includes the field corresponding to the descriptor instance.

Example

Example B-229

```
vmm_ral_vreg vreg[];
vmm_ral_vfield vfields[];
ral_model.status_vreg.get_fields(vfields);
foreach (vfields[i]) begin
   vreg[i] = vfields[i].get_register();
   vreg[i].display();
end
```

vmm_ral_vfield::get_lsb_pos_in_register()

Returns the offset of the least-significant bit of the field.

SystemVerilog

```
virtual function int unsigned get_lsb_pos_in_register()
```

Description

Returns the index of the least significant bit of the field in the virtual register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the virtual register.

Example

Example B-230

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
     `vmm_note(log, $psprintf("Offset of LSB of the field:
%s\n", vfields[i].get_lsb_pos_in_register()));
  end
```

vmm_ral_vfield::get_n_bits()

Returns the width of the field.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

Description

Returns the width, in number of bits, of the field.

Example

Example B-231

```
  integer count[];
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Width of the field: %d\n",
vfields[i].get_n_bits()));
       count[i] = vfields[i].get_n_bits();
  end
```

vmm_ral_vfield::get_access()

Returns the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the field when written and read through the optionally-specified domain.

If the register containing the field is shared across multiple domains, a domain must be specified. The access mode of a field in a specific domain may be restricted by the domain access rights of the memory implementing the field. For example, a RW field may only be writable through one of the domains and read-only through all of the other domains.

Example

Example B-232

```
  vmm_ral::access_e access;
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
    access = vfields[i].get_access();
    if (access == vmm_ral::RW) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::RW\n");
    else if (access == vmm_ral::RO) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::RO\n");
    else if (access == vmm_ral::WO) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::WO\n");
    else `vmm_note(log,
```

```
$psprintf("vmm_ral_vfield::get_access = %d\n", access));
  end
```

vmm_ral_vfield::display()

Displays a description of the field to stdout.

SystemVerilog

```
virtual function void display(string prefix = "")
```

Description

Displays the image created by the "vmm_ral_field::psdisplay()" method on the standard output.

Example

Example B-233

```
vmm_ral_vfield vfields[];
ral_model.status_vreg.get_fields(vfields);
foreach (vfields[i]) begin
    vfields[i].display();
end
```

vmm_ral_vfield::psdisplay()

Creates a human-readable description of the field.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the field and its current mirrored value. Each line of the description is prefixed with the specified prefix.

Example

Example B-234

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
   `vmm_note(log, $psprintf("psdisplay of the field: %s\n",
vfields[i].psdisplay()));
  end
```

vmm_ral_vfield::read()

Reads a virtual field value from the design.

SystemVerilog

```
virtual task read(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the field in the virtual register specified by the index from the associated memory using the specified access path.

If the field is located in a memory shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to

execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-235

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.status_vreg.vfield.read(0,status,data);
```

vmm_ral_vfield::write()

Sets a virtual field value in the design.

SystemVerilog

```
virtual task write(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Writes the specified field value in the virtual register specified by the index into the associated memory using the specified access path. If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, a read-only field will not be written.

If the virtual field is located in a memory shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the memory location where this virtual field is physically located contains other fields, a read-modify-write process is used to update the field value without modifying the others.

Example

Example B-236

```
vmm_rw::status_e status;
ral_model.status_vreg.vfield.write(0,status,'hABCD);
```

vmm_ral_vfield::peek()

Peek a virtual field value from the design.

SystemVerilog

```
virtual task peek(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Peek the current value of the virtual field from the associated memory using a back-door access. The value of the field in the design is not modified, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-237

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.status_vreg.vfield.peek(0,status,data);
```

vmm_ral_vfield::poke()

Poke a field value in the design.

SystemVerilog

```
virtual task poke(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Deposit the specified field value in the associated memory using a back-door access. The value of the field is updated, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the memory location where this field is physically located contains other fields, the current value of the other fields are peeked first then poked back in.

Example

Example B-238

```
vmm_rw::status_e status;
ral_model.status_vreg.vfield.poke(0,status,'hABCD);
```

vmm_ral_vfield::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(vmm_ral_vfield callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the order of registration.

Note that field callback methods will be invoked before their corresponding "vmm_ral_vreg" callback methods.

Example

Example B-239

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.append_callback(cb);
```

vmm_ral_vfield::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(vmm_ral_vfield_callbacks
cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the reverse order of registration.

Note that field callback methods will be invoked before their corresponding "vmm_ral_vreg" callback methods.

Example

Example B-240

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.prepend_callback(cb);
```

vmm_ral_vfield::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(vmm_ral_vfield_callbacks
cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this field descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-241

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.unregister_callback(cb);
```

vmm_ral_vfield_callbacks

Field descriptors.

Summary

- vmm_ral_vfield_callbacks::pre_write() ..... page B-477
- vmm_ral_vfield_callbacks::post_write() ..... page B-479
- vmm_ral_vfield_callbacks::pre_read() ..... page B-481
- vmm_ral_vfield_callbacks::post_read() ..... page B-483

vmm_ral_vfield_callbacks::pre_write()

OOP callback invoked before writing a field.

SystemVerilog

```
virtual task pre_write(vmm_ral_vfield      field,
                       longint unsigned    idx,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                         ref string            domain)
```

Description

This callback method is invoked before a value is written to a field in the DUT. The written value, if modified, changes the actual value that will be written. The path and domain used to write to the field can also be modified.

This callback method is only invoked when the "vmm_ral_vfield::write()" or "vmm_ral_vreg::write()" method is used to write to the field inside the DUT. This callback method is not invoked when the memory location is directly written to using the "vmm_ral_mem::write()" method.

Because writing a field causes the memory location to be written, and, therefore all of the other fields it contains to also be written, all registered "vmm_ral_vfield_callbacks::pre_write()" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_write()" methods will also be invoked as a side effect.

Example

Example B-242

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method
called\n",name,pretext);
 endtask

 task pre_write(vmm_ral_vfield                    field,
                longint unsigned                  idx,
                ref bit[`VMM_RAL_DATA_WIDTH-1:0] wdat,
                ref vmm_ral::path_e               path,
                ref string                        domain);
   cb("pre_write");
 endtask

endclass
```

vmm_ral_vfield_callbacks::post_write()

OOP callback invoked after writing a field.

SystemVerilog

```
virtual task post_write(vmm_ral_vfield        field,
                        longint unsigned      idx,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                          ref vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is written to a virtual field in the DUT.

This callback method is only invoked when the "vmm_ral_vfield::write()" or "vmm_ral_vreg::write()" method is used to write to the field inside the DUT. This callback method is not invoked when the memory location is directly written to using the "vmm_ral_mem::write()" method.

Because writing a field causes the memory location to be written, and, therefore all of the other fields it contains to also be written, all registered "vmm_ral_vfield_callbacks::post_write()" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::post_write()" methods will also be invoked as a side effect.

Example

Example B-243

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method
called\n",name,pretext);
 endtask

 task  post_write(vmm_ral_vfield                field,
                        longint unsigned             idx,
                      bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                       vmm_ral::path_e             path,
                      string                     domain,
                     ref vmm_rw::status_e        status);

   cb("post_write");
 endtask

endclass
```

vmm_ral_vfield_callbacks::pre_read()

OOP callback invoked before reading a field.

SystemVerilog

```
virtual task pre_read(vmm_ral_vfield      field,
                      longint unsigned    idx,
                      ref vmm_ral::path_e path,
                        ref string            domain)
```

Description

This callback method is invoked before a value is read from a field in the DUT. The path and domain used to read from the field can be modified.

This callback method is only invoked when the "vmm_ral_vfield::read()" method is used to read the field inside the DUT. This callback method is not invoked when the memory location containing the field is read directly using the "vmm_ral_mem::read()" method.

Because reading a field causes the memory location to be read, and, therefore all of the other fields it contains to also be read, all registered "vmm_ral_vfield_callbacks::pre_read()" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_read()" methods will also be invoked as a side effect.

Example

Example B-244

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method
called\n",name,pretext);
 endtask

 task pre_read(vmm_ral_vfield        field,
                         longint unsigned      idx,
                         ref vmm_ral::path_e   path,
                         ref string            domain);
   cb("pre_read");
 endtask

endclass
```

vmm_ral_vfield_callbacks::post_read()

OOP callback invoked after reading a field.

SystemVerilog

```
virtual task post_read(input vmm_ral_vfield    field,
                       input longint unsigned  idx,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                         ref   vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is read from a virtual field in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the "`vmm_ral_vfield::read()`" method and they can be modified.

This callback method is only invoked when the "`vmm_ral_vfield::read()`" method is used to read the field inside the DUT. This callback method is not invoked when the memory location containing the field is read directly using the "`vmm_ral_mem::read()`" method.

Because reading a field causes the memory location to be read, and, therefore all of the other fields it contains to also be read, all registered "`vmm_ral_vfield_callbacks::post_read()`" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "`vmm_ral_mem_callbacks::post_read()`" methods will also be invoked as a side effect.

Example

Example B-245

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method
called\n",name,pretext);
 endtask

 task post_read(vmm_ral_vfield                     field,
                longint unsigned                   idx,
                ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                vmm_ral::path_e                    path,
                string                             domain,
                ref vmm_rw::status_e               status);
   cb("post_read");
 endtask

endclass
```

vmm_ral_vreg

Base class for virtual register descriptors.

Summary

- vmm_ral_vreg::log ..... page B-486
- vmm_ral_vreg::get_name() ..... page B-487
- vmm_ral_vreg::get_fullname() ..... page B-488
- vmm_ral_vreg::get_block() ..... page B-489
- vmm_ral_vreg::implement() ..... page B-490
- vmm_ral_vreg::allocate() ..... page B-492
- vmm_ral_vreg::get_region() ..... page B-493
- vmm_ral_vreg::release_region() ..... page B-494
- vmm_ral_vreg::get_memory() ..... page B-495
- vmm_ral_vreg::get_n_domains() ..... page B-496
- vmm_ral_vreg::get_domains() ..... page B-497
- vmm_ral_vreg::get_access() ..... page B-498
- vmm_ral_vreg::get_rights() ..... page B-499
- vmm_ral_vreg::get_offset_in_memory() ..... page B-500
- vmm_ral_vreg::get_address_in_system() ..... page B-501
- vmm_ral_vreg::get_size() ..... page B-502
- vmm_ral_vreg::get_n_bytes() ..... page B-503
- vmm_ral_vreg::get_n_memlocs() ..... page B-504
- vmm_ral_vreg::get_incr() ..... page B-505
- vmm_ral_vreg::display() ..... page B-506
- vmm_ral_vreg::psdisplay() ..... page B-507
- vmm_ral_vreg::psdisplay_domain() ..... page B-508
- vmm_ral_vreg::get_fields() ..... page B-509
- vmm_ral_vreg::get_field_by_name() ..... page B-510
- vmm_ral_vreg::read() ..... page B-511
- vmm_ral_vreg::write() ..... page B-513
- vmm_ral_vreg::peek() ..... page B-515
- vmm_ral_vreg::poke() ..... page B-517
- vmm_ral_vreg::append_callback() ..... page B-518
- vmm_ral_vreg::prepend_callback() ..... page B-519
- vmm_ral_vreg::unregister_callback() ..... page B-520

vmm_ral_vreg::log

Message service interface.

SystemVerilog

```
vmm_log log
```

Description

Message service interface instance for the virtual register descriptor. A single message service interface instance is shared by all virtual register abstraction class instances.

Example

Example B-246

```
  vmm_log log = new("Test", "Main");

  ral_model.virtual_reg.read(0, status, data);
  if (status != vmm_rw::IS_OK) begin
     `vmm_error(log, "Non-OK status when reading through
virtual register");
  end

  log.report();
```

vmm_ral_vreg::get_name()

Returns the name of the virtual register.

SystemVerilog

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

Description

Returns the name of the register corresponding to the instance of the descriptor.

Example

Example B-247

```
  vmm_ral_vreg vregs[];
  ral_model.get_virtual_registers(vregs)
  foreach (vregs[i]) begin
    `vmm_note(log, $psprintf("Virtual register Name: %s\n",
vregs[i].get_name()));
  end
```

vmm_ral_vreg::get_fullname()

Returns the fully-qualified name of the virtual register.

SystemVerilog

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the register corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-248

```
  vmm_ral_vreg vregs[];
  ral_model.get_virtual_registers(vregs)
  foreach (vregs[i]) begin
    `vmm_note(log, $psprintf("Virtual register Name: %s\n",
vregs[i].get_fullname()));
  end
```

vmm_ral_vreg::get_block()

Returns the block instantiating the virtual register.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the register corresponding to the descriptor instance.

Example

Example B-249

```
vmm_ral_block blk;
blk = ral_model.virtual_register.get_block();
blk.display();
```

vmm_ral_vreg::implement()

Dynamically implement a set of virtual registers.

SystemVerilog

```
virtual function bit implement(
     longint unsigned n,
     vmm_ral_mem      mem    = null,
     bit [63:0]       offset = 0,
     int unsigned       incr    = 0)
```

Description

Dynamically implement, resize or relocate a set of virtual registers of the specified size, in the specified memory and offset. If an offset increment is specified, each virtual register is implemented at the specified offset from the previous one. If an offset increment of 0 is specified, virtual registers are packed as closely as possible in the memory. If no memory is specified, the virtual register set is in the same memory, at the same offset using the same offset increment as originally implemented.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory now implementing them.

Returns `TRUE` if the memory can implement the number of virtual registers at the specified offset and increment. Returns `FALSE` if the memory cannot implement the specified virtual register set.

The memory region used to implement a set of virtual registers is reserved to prevent it from being allocated for another purpose by the memory's default memory allocation manager.

Statically-implemented virtual registers cannot be implemented, resized nor relocated.

vmm_ral_vreg::allocate()

Dynamically implement a set of virtual registers.

SystemVerilog

```
virtual function vmm_mam_region allocate(
     longint unsigned n,
     vmm_mam           mam)
```

Description

Dynamically implement, resize or relocate a set of virtual registers of the specified size to a randomly allocated region of the appropriate size in the address space managed by the specified memory allocation manager.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory region now implementing them.

Returns a reference to a memory region descriptor if the memory allocation manager was able to allocate a region that can implement the number of virtual registers. Returns `null` if the memory allocation manager cannot allocate a suitable region.

Statically-implemented virtual registers cannot be implemented, resized nor relocated.

vmm_ral_vreg::get_region()

Returns the region descriptor where virtual registers are implemented.

SystemVerilog

```
virtual function vmm_mam_region get_region()
```

Description

Returns a reference to a memory region descriptor that implements the set of virtual registers. Returns `null` if the virtual registers are not currently implemented.

A region implementing a set of virtual registers must not be released using the `vmm_mam::release_region()` method. It must be released using the "`vmm_ral_vreg::release_region()`" method.

Example

Example B-250

```
  vmm_mam_region mr;
  mr = ral_model.virtual_register.get_region();
  if (mr == null) `vmm_note(log, "failed to get_region on
virtual register");
```

vmm_ral_vreg::release_region()

Free the memory used to implement a set of virtual registers.

SystemVerilog

```
virtual function void release_region()
```

Description

Release the memory region used to implement the set of virtual registers and return it to the pool of available memory that can be allocated by the memory's default allocation manager. The virtual registers are subsequently considered as unimplemented and can no longer be accessed.

Statically-implemented virtual registers cannot be released.

Example

Example B-251

```
vmm_mam_region mr;
ral_model.virtual_register.release_region();
mr = ral_model.virtual_register.get_region();
if (mr != null) `vmm_note(log, "'release_region' Failed");
```

vmm_ral_vreg::get_memory()

Returns the memory that implements this register.

SystemVerilog

```
virtual function vmm_ral_mem get_memory()
```

Description

Returns a reference to the memory abstraction class for the memory that implements the set of virtual registers corresponding to the descriptor instance.

Example

Example B-252

```
vmm_ral_mem mem;
mem = ral_model.virtual_register.get_memory();
```

vmm_ral_vreg::get_n_domains()

Returns the number of domains sharing this virtual register.

SystemVerilog

```
function int get_n_domains()
```

Description

Returns the number of domains that share the memory implementing this set of virtual registers. The name of the domains can be obtained with the "vmm_ral_reg::get_domains()" method.

Example

Example B-253

```
 int num_domains;
num_domains = ral_model.virtual_register.get_n_domains();
```

vmm_ral_vreg::get_domains()

Returns the name of the domains sharing this virtual register.

SystemVerilog

```
function void get_domains(ref string domains[])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access the memory implementing this set of virtual registers. The order of the domain names is not specified.

Example

Example B-254

```
  string domains[];
  ral_model.virtual_register.get_domains(domains);
  foreach (domains[i]) begin
     `vmm_note(log, $psprintf("Domain %d = %s\n", i,
domains[i]));
  end
```

vmm_ral_vreg::get_access()

Returns the access mode of the virtual register.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the memory used to implement the set of virtual register when written and read. If the memory is shared across more than one domain, a domain name must be specified.

If access restrictions are present when accessing a memory through the specified domain, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through a domain with read-only restrictions would return `vmm_ral::RO`.

Example

Example B-255

```
vmm_ral::access_e access;
access = ral_model.virtual_register.get_access();
```

vmm_ral_vreg::get_rights()

Returns the access rights of this virtual register.

SystemVerilog

```
function vmm_ral::access_e get_rights(string domain = "")
```

Description

Returns the access rights of the memory implementing this set of virtual registers. Returns `vmm_ral::RW`, `vmm_ral::RO` or `vmm_ral::WO`. See "vmm_ral_mem::get_rights()" for more details.

If the memory implementing this set of virtual registers is shared in more than one domain, a domain name must be specified. If the memory is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-256

```
vmm_ral::access_e rights;
rights = ral_model.virtual_register.get_rights();
```

vmm_ral_vreg::get_offset_in_memory()

Returns the address of the virtual register within the memory address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_memory(
     longint unsigned idx)
```

Description

Returns the offset of the virtual register in the overall address space of the memory that implements it.

If the virtual register occupies more than one memory location, the lowest offset value is returned.

Example

Example B-257

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr =
ral_model.virtual_register.get_offset_in_memory(3);
```

vmm_ral_vreg::get_address_in_system()

Returns the address of the virtual register within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
     longint unsigned idx,
     string            domain = "")
```

Description

Returns the address of the virtual register in the overall address space of the design. If the memory implementing the virtual register is shared between multiple physical interfaces, a domain must be specified.

If the virtual register is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-258

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr =
ral_model.virtual_register.get_address_in_system(2);
```

vmm_ral_vreg::get_size()

Returns the size of the virtual register array.

SystemVerilog

```
virtual function int unsigned get_size()
```

Description

Returns the number of virtual registers in the virtual register array.

Example

Example B-259

```
int size;
size = ral_model.virtual_register.get_size();
```

vmm_ral_vreg::get_n_bytes()

Returns the width of the virtual register.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

Description

Returns the width, in number of bytes, of the virtual register.

The width of a virtual register is always a multiple of the width of the memory locations used to implement it. For example, a virtual register containing two 1-byte fields implemented in a memory with 4-bytes memory locations is 4-byte wide.

Example

Example B-260

```
int width;
width = ral_model.virtual_register.get_n_bytes();
```

vmm_ral_vreg::get_n_memlocs()

Returns the size of the virtual register array.

SystemVerilog

```
virtual function int unsigned get_n_memlocs()
```

Description

Returns the number of memory locations used by a single virtual register.

Example

Example B-261

```
int size;
size = ral_model.virtual_register.get_n_memlocs();
```

vmm_ral_vreg::get_incr()

Returns the distance between two virtual registers.

SystemVerilog

```
virtual function int unsigned get_incr()
```

Description

Returns the number of memory locations between two individual virtual registers in the same array.

Example

Example B-262

```
int dist;
dist = ral_model.virtual_register.get_incr();
```

vmm_ral_vreg::display()

Displays a description of the virtual register to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

Description

Displays the image created by the "vmm_ral_vreg::psdisplay()" method to the standard output.

Example

Example B-263

```
ral_model.virtual_register.display();
```

vmm_ral_vreg::psdisplay()

Creates a human-readable description of the virtual register.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

Example

Example B-264

```
`vmm_note(log, $psprintf("Virtual Register description =
%s\n", ral_model.virtual_register.psdisplay()));
```

vmm_ral_vreg::psdisplay_domain()

Creates a human-readable description of the virtual register.

SystemVerilog

```
virtual function string psdisplay_domain(string prefix = "",
                                         string domain = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the address of the register within that domain is used.

Example

Example B-265

```
`vmm_note(log, $psprintf("Virtual Register description =
%s\n", ral_model.virtual_register.psdisplay_domain()));
```

vmm_ral_vreg::get_fields()

Returns all virtual fields in this virtual register.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_vfield fields[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields contained in the virtual register. Fields are ordered from least-significant position to most-significant position within the register.

Example

Example B-266

```
vmm_ral_vfield vfields[];
ral_model.virtual_register.get_fields(vfields);
```

vmm_ral_vreg::get_field_by_name()

Returns the virtual field with the specified name in this virtual register.

SystemVerilog

```
virtual function vmm_ral_vfield get_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name in the register and returns its descriptor. If no fields are found, returns `null`.

Example

Example B-267

```
  vmm_ral_vfield vfield;
  vfield =
ral_model.virtual_register.get_field_by_name("virtual_fiel
d_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_vreg::read()

Reads a virtual register from the design.

SystemVerilog

```
virtual task read(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the specified virtual register from the design using the specified access path. If the memory implementing the virtual register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

`data_id`
`scenario_id`
`stream_id`

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-268

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.virtual_reg.read(0,status,data);
```

vmm_ral_vreg::write()

Sets a register in the design.

SystemVerilog

```
virtual task write(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Writes the specified value in the specified virtual register in the design using the specified access path. If the memory implementing the virtual register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

`data_id`
`scenario_id`
`stream_id`

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "vmm_rw_access" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-269

```
vmm_rw::status_e status;
ral_model.virtual_reg.write(0,status,'hABCD);
```

vmm_ral_vreg::peek()

Peek a virtual register from the design.

SystemVerilog

```
virtual task peek(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the specified virtual register from the design using a back-door access. The memory implementing the virtual register must provide a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-270

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.virtual_reg.peek(0,status,data);
```

vmm_ral_vreg::poke()

Poke a virtual register in the design.

SystemVerilog

```
virtual task poke(
   input  ilongint unsigned idx,
   output vmm_rw::status_e  status,
   input  bit [63:0]        value,
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int                  stream_id = -1)
```

Description

Deposit the specified value in the specified virtual register in the design, as-is, using a back-door access. The memory implementing the virtual register must provide a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-271

```
vmm_rw::status_e status;
ral_model.virtual_reg.poke(0,status,'hABCD);
```

vmm_ral_vreg::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_vreg_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this virtual register descriptor. Callbacks are invoked in the order of registration.

Note that the corresponding "vmm_ral_vfield" callback methods will be invoked before the virtual register callback methods.

Example

Example B-272

```
  vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.append_callback(cb);

//See "vmm_ral_vreg_callbacks" classes for complete
//definition of vreg_callback.
```

vmm_ral_vreg::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(
   vmm_ral_vreg_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the reverse order of registration.

Note that the corresponding "vmm_ral_vfield" callback methods will be invoked before the virtual register callback methods.

Example

Example B-273

```
  vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.prepend_callback(cb);

// See "vmm_ral_vreg_callbacks" classes for complete
// definition of vreg_callback.
```

vmm_ral_vreg::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(
   vmm_ral_vreg_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this register descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-274

```
vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.unregister_callback(cb);

//See "vmm_ral_vreg_callbacks" classes for the definition
//of vreg_callback.
```

vmm_ral_vreg_callbacks

Base class for virtual register descriptors.

Summary

- vmm_ral_vreg_callbacks::pre_write() ..... page B-522
- vmm_ral_vreg_callbacks::post_write() ..... page B-524
- vmm_ral_vreg_callbacks::pre_read() ..... page B-526
- vmm_ral_vreg_callbacks::post_read() ..... page B-528

vmm_ral_vreg_callbacks::pre_write()

OOP callback invoked before writing a register.

SystemVerilog

```
virtual task pre_write(vmm_ral_vreg        rg,
                       longint unsigned    idx,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                         ref string            domain)
```

Description

This callback method is invoked before a value is written to a virtual register in the DUT. The written value, if modified, changes the actual value that is written. The path and domain used to write to the register can also be modified.

This callback method is only invoked when the "vmm_ral_vreg::write()" method is used to write to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is written to using the "vmm_ral_mem::write()" method.

Because writing a register causes all of the fields it contains to be written, all registered "vmm_ral_vfield_callbacks::pre_write()" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_write()" methods will also be invoked as a side effect.

Example

Example B-275

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
          else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
           else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task pre_write(vmm_ral_vreg                  rg,
                       longint unsigned               idx,
                    ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                      ref vmm_ral::path_e            path,
                     ref string                   domain);
              $display("vreg_callback: Executing Callback
task pre_write for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Writing Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, wdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: pre_write

endclass
```

vmm_ral_vreg_callbacks::post_write()

OOP callback invoked after writing a register.

SystemVerilog

```
virtual task post_write(vmm_ral_vreg          rg,
                        longint unsigned      idx,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                          ref vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is successfully written to a register in the DUT. If a physical write access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is only invoked when the "`vmm_ral_vreg::write()`" method is used to write to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is written to using the "`vmm_ral_mem::write()`" method.

Because writing a register causes all of the fields it contains to be written, all registered "`vmm_ral_vfield_callbacks::post_write()`" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "`vmm_ral_mem_callbacks::post_write()`" methods will also be invoked as a side effect.

Example

Example B-276

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
          else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
           else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task post_write(vmm_ral_vreg                 rg,
                          longint unsigned             idx,
                        bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                         vmm_ral::path_e              path,
                        string                      domain,
                      ref vmm_rw::status_e        status);
              $display("vreg_callback: Executing Callback
task post_write for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Wrote Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, wdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: post_write

endclass
```

vmm_ral_vreg_callbacks::pre_read()

OOP callback invoked before reading a virtual register.

SystemVerilog

```
virtual task pre_read(vmm_ral_vreg        rg,
                      longint unsigned    idx,
                      ref vmm_ral::path_e path,
                        ref string            domain)
```

Description

This callback method is invoked before a value is read from a register in the DUT. The path and domain used to read the register can be modified.

This callback method is only invoked when the "vmm_ral_vreg::read()" method is used to read to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is read to using the "vmm_ral_mem::read()" method.

Because reading a register causes all of the fields it contains to be written, all registered "vmm_ral_vfield_callbacks::pre_read()" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_read()" methods will also be invoked as a side effect.

Example

Example B-277

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
          else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
           else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task pre_read(vmm_ral_vreg         rg,
                        longint unsigned     idx,
                        ref vmm_ral::path_e  path,
                        ref string           domain);
              $display("vreg_callback: Executing Callback
task pre_read for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: pre_read

endclass
```

vmm_ral_vreg_callbacks::post_read()

OOP callback invoked after reading a virtual register.

SystemVerilog

```
virtual task post_read(input vmm_ral_vreg       rg,
                       input longint unsigned   idx,
                       ref   bit [63:0]         rdat,
                       input vmm_ral::path_e    path,
                       input string             domain
                         ref   vmm_rw::status_e     status)
```

Description

This callback method is invoked after a value is successfully read from a register in the DUT. The `rdat` and `status` values are the values that will be ultimately returned by the "`vmm_ral_vreg::read()`" method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is only invoked when the "`vmm_ral_vreg::read()`" method is used to read to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is read to using the "`vmm_ral_mem::read()`" method.

Because reading a register causes all of the fields it contains to be written, all registered "`vmm_ral_vfield_callbacks::post_read()`" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory

implementing the virtual field is accessed through its own abstraction class, all of its registered s methods will also be invoked as a side effect.

Example

Example B-278

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
          else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
           else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task post_read(vmm_ral_vreg                    rg,
                      longint unsigned                  idx,
                    ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                      input vmm_ral::path_e            path,
                     input string                    domain,
                    ref vmm_rw::status_e            status);
              $display("vreg_callback: Executing Callback
task post_read for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Read Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, rdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: post_read

endclass
```

vmm_ral_tests

Summary

- vmm_ral_tests::bit_bash() ..... page B-531
- vmm_ral_tests::hw_reset() ..... page B-532
- vmm_ral_tests::mem_access() ..... page B-533
- vmm_ral_tests::mem_walk() ..... page B-534
- vmm_ral_tests::reg_access() ..... page B-535
- vmm_ral_tests::shared_access() ..... page B-536

vmm_ral_tests::bit_bash()

Verify the bits in a registers.

SystemVerilog

```
static task bit_bash(vmm_ral_block blk,
     string domain,
     vmm_log log);
```

OpenVera

```
Not supported
```

Description

Exercise every bit in the registers found in the specified domain in the specified block to make sure it behaves as specified. Bits in fields with a mode specified as USER or OTHER are not verified. If the domain is specified as "", all registers are exercised. Registers with an attribute named "NO_BIT_BASH_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up and in the reset state. The state of the block must remain idle and stable so the register values do no change from their initial state.

vmm_ral_tests::hw_reset()

Verify initial reset values of registers.

SystemVerilog

```
static task hw_reset(vmm_ral_block blk,
     string domain,
     vmm_log log);
```

OpenVera

```
Not supported
```

Description

Read every register in the specified domains of the specified block and verify that the value read corresponds to the specified reset value. If the domain is specified as "", all registers are exercised. Registers with an attribute named "NO_HW_RESET_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up and in the reset state. The state of the block must remain idle and stable so the register values do no change from their initial state.

vmm_ral_tests::mem_access()

Verify the correctness of memory accesses

SystemVerilog

```
static task mem_access(vmm_ral_block blk,
     vmm_log log);
```

OpenVera

```
Not supported
```

Description

Write every memory location through every available front-door domain and read the corresponding value through the backdoor and vice-versa. Memories with no backdoor access defined or with an attribute named "NO_MEM_ACCESS_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up and in the reset state. The state of the block must remain idle and stable so the memory locations do no change from their initial state.

vmm_ral_tests::mem_walk()

Verify access to memories.

SystemVerilog

```
static task mem_walk(vmm_ral_block blk,
     string domain,
     vmm_log log);
```

OpenVera

```
Not supported
```

Description

Write then read back every location in every memories in the specified domains of the specified block and verify that the value read corresponds to the previously-written value. If the domain is specified as "", all memories are exercised. Memories with an attribute named "NO_MEM_WALK_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up. The content of the memories must remain unchanged and stable so the values do no change from their previously-written state.s

vmm_ral_tests::reg_access()

Verify the correctness of register accesses

SystemVerilog

```
static task reg_access(vmm_ral_block blk,
    vmm_log log);
```

OpenVera

```
Not supported
```

Description

Write every register through every available front-door domain and read the corresponding value through the backdoor and vice-versa. Registers with no backdoor access defined or with an attribute named "NO_REG_ACCESS_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up and in the reset state. The state of the block must remain idle and stable so the register values do no change from their initial state.

vmm_ral_tests::shared_access()

Verify the correctness of shared accesses

SystemVerilog

```
static task shared_access(vmm_ral_block blk,
    vmm_log log);
```

OpenVera

```
Not supported
```

Description

Write every shared register and shared memory locations through every available front-door domain and read the corresponding value through every other available front door (and backdoor if available) and vice-versa. Registers and memories that are not shared across multiple domains or with an attribute named "NO_SHARED_ACCESS_TEST" or "NO_RAL_TESTS" are skipped.

This method must be invoked only when the block has been fully powered-up and in the reset state. The state of the block must remain idle and stable so the register values and memory locations do no change from their initial state.

vmm_rw

Utility class for global symbolic values. Each set of symbolic values is specified using enumerated types. The symbolic values are accessed using a fully-qualified name, such as `vmm_rw::IS_OK`.

A separate, encapsulated class is used to minimize the length of these identifiers and to make them easier to share across classes.

Summary

- `vmm_rw::kind_e` ..... page B-538
- `vmm_rw::status_e` ..... page B-540

vmm_rw::kind_e

Symbolic values identifying the kind of transaction to perform on a physical interface.

SystemVerilog

```
vmm_rw::READ
vmm_rw::WRITE
vmm_rw::EXPECT
```

OpenVera

```
vmm_rw::READ
vmm_rw::WRITE
vmm_rw::EXPECT
```

Description

`vmm_rw::READ`

Performs a read operation from a specified address and returns a value.

`vmm_rw::WRITE`

Performs a write operation of a specified value at a specified address.

`vmm_rw::EXPECT`

Performs a read operation from a specified address and compares the value read against a specified value.

Example

Example B-279

```
class my_rw_master extends vmm_rw_xactor;
   ...
   my_data data;
   virtual task execute_single(vmm_rw_access tr);
      ...
      if(tr.kind == vmm_rw::READ)
         data.addr == tr.addr << 2;
         ...
      ...
   endtask
   ...
endclass
```

vmm_rw::status_e

Symbolic values identifying the completion status of a transaction on a physical interface.

SystemVerilog

```
vmm_rw::status_e;
vmm_rw::IS_OK
vmm_rw::ERROR
vmm_rw::RETRY
vmm_rw::HAS_X
```

OpenVera

```
vmm_rw::IS_OK
vmm_rw::ERROR
vmm_rw::RETRY
```

Description

`vmm_rw::IS_OK`

The transaction completed successfully.

`vmm_rw::ERROR`

The transaction did not complete or completed with an error indication.

`vmm_rw::RETRY`

The transaction completed with a `retry` indication.

`vmm_rw:HAS_X`

The transaction has detected an ’X’ on the databus. This status should be set by the user.

Example

Example B-280

```
class my_rw_master extends vmm_rw_xactor;
   ...
   virtual task execute_single(vmm_rw_access tr);
      my_data data;
      ...
      data = new();
      ...
      if(data.status == my_data::OK)
         tr.status = vmm_rw::IS_OK;
      ...
   endtask
   ...
endclass
```

vmm_rw_access

Descriptor for generic bus transactions. Derived from `vmm_data/rvm_data`.

Summary

- vmm_rw_access::kind ..... page B-543
- vmm_rw_access::addr ..... page B-544
- vmm_rw_access::data ..... page B-545
- vmm_rw_access::n_bits ..... page B-546
- vmm_rw_access::status ..... page B-548
- vmm_rw_access::new() ..... page B-549
- vmm_rw_access::psdisplay() ..... page B-550

vmm_rw_access::kind

Defines the type of transaction described by this instance.

SystemVerilog

```
rand vmm_rw::kind_e kind
```

OpenVera

```
rand vmm_rw::kind_e kind
```

Description

Identifies the type of transaction that is described by this generic bus transaction descriptor instance.

Example

Example B-281

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
     kind == vmm_rw::READ;
   }
   ...
endclass
`vmm_channel(my_rw_access)
...
my_rw_access tr;
`vmm_note(log,$psprintf("vmm_rw_access::kind==>
%0d",tr.kind));
...
```

vmm_rw_access::addr

Address to read from or write to.

SystemVerilog

```
rand bit [63:0] addr
```

OpenVera

```
rand bit [63:0] addr
```

Description

Address that is the target of the transaction. If it is a `read` transaction, the specified address is read. If it is a `write` transaction, the specified address is written to.

Example

Example B-282

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      addr > 'h0;
      addr <= 'hFFFF;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::addr==>
%0d",tr.addr));
...
```

vmm_rw_access::data

Data to write or that has been read.

SystemVerilog

```
rand bit [63:0] data
```

OpenVera

```
rand bit [63:0] data
```

Description

If it is a read transaction, it is the value that was read at the specified address. If it is a write transaction, the specified value is written at the specified address.

Example

Example B-283

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      data = addr + 1;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::data==>
%0d",tr.data));
...
```

vmm_rw_access::n_bits

Number of valid data bits.

SystemVerilog

```
rand int n_bits
```

OpenVera

```
rand int n_bits
```

Description

Specifies the number of valid bits in the read or write cycle. The valid bits are always right justified.

This class property is constrained to be in the 1..64 range.

Example

Example B-284

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      n_bits > 0;
      n_bits < 10;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::nbits==>
%0d",tr.n_bits));
...
```

vmm_rw_access::status

Completion status of the transaction.

SystemVerilog

```
vmm_rw::status_e status
```

OpenVera

```
vmm_rw::status_e status
```

Description

Specifies the completion status of the transaction.

Example

Example B-285

```
   ...
   my_rw_access tr;
   `vmm_note(log,$psprintf("vmm_rw_access::status==>
%0d",tr.status));
   ...
```

vmm_rw_access::new()

Creates a new instance of a transaction descriptor.

SystemVerilog

```
function new(vmm_object parent = null, string name = "")
```

OpenVera

```
task new()
```

Description

Creates a new instance of a transaction descriptor.

Example

Example B-286

```
class my_rw_access extends vmm_rw_access;
   vmm_log log
   ...
   function new();
      super.new(this.log,parent,name);
      log = new("my_rw_access","class");
      `vmm_note(log,"vmm_rw_access:: instance created.");
   endfunction
   ...
endclass
```

vmm_rw_access::psdisplay()

Creates an image representing the described transaction.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable image of the content of the transaction descriptor. Every line in the image is prefixed by the specified prefix. The image is returned as a string terminated with a newline.

Example

Example B-287

```
class my_rw_access extends vmm_rw_access;
   ...
  //Creates an image representing the described transaction.
   `vmm_note(log,$psprintf("%0s",this.psdisplay("RW
Access")));
   ...
endclass
```

vmm_rw_burst

Descriptor for generic bus burst transactions. Derived from "vmm_rw_access".

Summary

- vmm_rw_burst::n_beats ..... page B-552
- vmm_rw_burst::incr_addr ..... page B-553
- vmm_rw_burst::max_addr ..... page B-555
- vmm_rw_burst::data ..... page B-557
- vmm_rw_burst::user_data ..... page B-559

vmm_rw_burst::n_beats

Defines the length of the burst.

SystemVerilog

```
rand int n_beats
```

OpenVera

```
rand integer n_beats
```

Description

Specifies the number of beats or transfers in a burst transaction that is described by this generic burst transaction descriptor instance.

Example

Example B-288

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      n_beats <= 1024;
   }
   ...
endclass
...
my_rw_burst br;
`vmm_note(log,$psprintf("vmm_ral_mem_burst::n_beats ==>
%0d",br.n_beats));
...
```

vmm_rw_burst::incr_addr

Address increment between beats.

SystemVerilog

```
rand bit [63:0] incr_addr
```

OpenVera

```
rand bit [63:0] incr_addr
```

Description

Implicit or explicit address increment between individual beats of a burst transaction. The first beat reads or writes the address specified by the "vmm_rw_access::addr" class property. The nth beat reads or writes the address specified by
"vmm_rw_access::addr" + (*n*-1)
"vmm_rw_burst::incr_addr".

A value of 0 implies a burst access that repeatedly accessed the same location.

Example

Example B-289

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      incr_addr inside {0, 1, 2, 4, 8, 16, 32};
   }
   ...
endclass
...
```

```
my_rw_burst br;
`vmm_note(log,$psprintf("vmm_ral_mem_burst::incr_addr ==>
%0d",br.incr_addr));
...
```

vmm_rw_burst::max_addr

Maximum address for the burst.

SystemVerilog

```
rand bit [63:0] max_addr
```

OpenVera

```
rand bit [63:0] max_addr
```

Description

Limits the address range of the burst and causes wrapping if subsequent beats would access an address past the specified address.

Example

Example B-290

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      n_beats <= 1024;
      incr_addr inside {0, 1, 2, 4, 8, 16, 32};
   }

   constraint linear {
      incr_addr == 1;
      max_addr == addr + n_beats - 1;
   }
   ...
endclass
...
my_rw_burst br;
```

```
`vmm_note(log,$psprintf("vmm_ral_mem_burst::max_addr==>
%0d",br.max_addr));
...
```

vmm_rw_burst::data

Data to write or that has been read.

SystemVerilog

```
rand bit [63:0] data[]
```

OpenVera

```
rand bit [63:0] data[*]
```

Description

If it is a read transaction, it is the values that were read during the burst-read operation. If it is a write transaction, the specified values are written during the burst-write operation.

The number of elements in the array must be equal to the number of beats specified by "vmm_rw_burst::n_beats".

Important: this class property hides "vmm_rw_access::data".

Example

Example B-291

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint vmm_rw_burst_valid {
      n_beats > 0;
      data.size() == n_beats;
   }
   ...
endclass
...
my_rw_burst br;
```

```
foreach(br.data[i])
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::data ==>
%0h",br.data[i]));
...
```

vmm_rw_burst::user_data

Additional burst configuration information.

SystemVerilog

```
vmm_data user_data
```

OpenVera

```
rvm_data user_data
```

Description

Provides a mechanism for passing additional burst configuration information to the "`vmm_rw_xactor::execute_burst()`" method. Any reference to additional user information is passed through transparently. The additional information can be recovered by using `$cast()` or `cast_assign()`.

Example

Example B-292

```
class my_rw_xactor extends vmm_rw_xactor;
   ...
   my_rw_burst burst;
   virtual protected task execute_burst(vmm_rw_burst br)
     ...
     $cast(burst.user_data,br);
     ...
   endtask
   ...
endclass
```

vmm_rw_xactor

Base class, derived from `vmm_xactor/rvm_xactor`, for bus-functional models executing generic bus transactions described using `vmm_rw_access` descriptors.

Summary

- vmm_rw_xactor::exec_chan ..... page B-561
- vmm_rw_xactor::new() ..... page B-563
- vmm_rw_xactor::execute_burst() ..... page B-564
- vmm_rw_xactor::execute_single() ..... page B-566
- vmm_rw_xactor::notifications_e ..... page B-568
- vmm_rw_xactor::pre_single() ..... page B-570
- vmm_rw_xactor::pre_burst() ..... page B-571
- vmm_rw_xactor::post_single() ..... page B-572
- vmm_rw_xactor::post_burst() ..... page B-573

vmm_rw_xactor::exec_chan

Transaction execution input channel.

SystemVerilog

```
vmm_rw_access_channel exec_chan
```

OpenVera

```
vmm_rw_access_channel exec_chan
```

Description

This channel is used to supply a stream of generic bus transaction descriptors to be executed on a physical interface. This channel implements a blocking completion model. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor.

Not usually used by RAL users and is used by "vmm_ral_access" instances for executing physical accesses to read and write registers or memory locations in the design.

Example

Example B-293

```
class my_rw_access extends vmm_rw_access;
   ...
endclass
`vmm_channel(my_rw_access)
...
class my_rw_xactor extends vmm_rw_xactor;
   ...
   my_rw_access_channel exec_chan;
```

```
   ...
   function new(string                inst,
                int unsigned          stream_id,
                my_rw_access_channel exec_chan = null);
      super.new("RAL Master", inst, stream_id, exec_chan);
      ...
   endfunction
   ...
endclass
```

vmm_rw_xactor::new()

Creates a new instance of the base class.

SystemVerilog

```
function new(
   string                 name,
   string                 instance,
   int                    stream_id = -1,
   vmm_rw_access_channel exec_chan = null)
```

OpenVera

```
task new(
   string                 name,
   string                 instance,
   int                    stream_id = -1,
   vmm_rw_access_channel exec_chan = null)
```

Description

Creates a new instance of this base class, with the specified name and instance name and optional stream identifier. If a channel is specified, it is assigned to the `vmm_ral_xactor::exec_chan` class property and reconfigured with a full level of `1`. Otherwise, a new channel instance is allocated and used.

Example

Example B-294

```
function ral_wb_master::new(string name,
                            string instance,
                            int    stream_id = -1);
   super.new(name, instance, stream_id);
endfunction: new
```

vmm_rw_xactor::execute_burst()

Executes a burst transfer transaction.

SystemVerilog

```
virtual protected task execute_burst(vmm_rw_burst tr)
```

OpenVera

```
virtual protected task execute_burst_t(vmm_rw_burst tr)
```

Description

This method may be overloaded. It is called by the base class to request that a generic burst bus transaction be executed on a physical interface. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor then the task should return.

By default, burst transactions are executed as a series of single transactions, using "vmm_rw_xactor::execute_single()". If this transactor is mapped to a physical protocol that supports burst transactions, this method should be overloaded.

This method is ultimately used to execute the following operations:

"vmm_ral_mem::burst_read()"
"vmm_ral_mem::burst_write()"
"vmm_ral_access::burst_read()"
"vmm_ral_access::burst_write()"

Example

Example B-295

```
class my_rw_master extends vmm_rw_xactor;
   ...
   function new(string                inst,
                int unsigned          stream_id,
                my_rw_access_channel exec_chan = null);
      super.new("RW Master", inst, stream_id, exec_chan);
      ...
   endfunction
   ...
   task execute_burst(vmm_rw_burst tr);
      bit [`VMM_RW_ADDR_WIDTH-1:0] addr;
      addr = tr.addr;
      tr.status = vmm_rw::IS_OK;
      ...
   endtask
   ...
endclass
...
my_rw_burst br;
my_rw_master my_xactor;
...
this.my_xactor.execute_single(tr);
...
```

vmm_rw_xactor::execute_single()

Executes a single transfer transaction.

SystemVerilog

```
virtual protected task execute_single(vmm_rw_access tr)
```

OpenVera

```
virtual protected task execute_single_t(vmm_rw_access tr)
```

Description

This method must be overloaded. It is called by the base class to request that a generic bus transaction be executed on a physical interface. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor then the task should return.

If you require concurrency in executing generic bus transaction to the physical interface, refer to Example 6-2 on page 6 for a demonstration of the recommended way of extending/implementing the `execute_single()` task in that scenario.

Example

Example B-296

```
class my_rw_master extends vmm_rw_xactor;
   ...
   my_rw_access_channel exec_chan;
   ...
   virtual task execute_single(vmm_rw_access tr);
      wb_cycle cyc;
      ...
```

```
      cyc = new();
      ...
   endtask
   ...
endclass
...
my_rw_access tr;
my_rw_master my_xactor;
...
this.my_xactor.execute_single(tr);
...
```

vmm_rw_xactor::notifications_e

Symbolic values identifying notifications indicated by the transactor.

SystemVerilog

```
vmm_rw_xactor::BURST_DONE
vmm_rw_xactor::SINGLE_DONE
```

OpenVera

```
vmm_rw_xactor::BURST_DONE
vmm_rw_xactor::SINGLE_DONE
```

Description

`vmm_rw_xactor::BURST_DONE`

Indicates that a burst cycle has been completed. The burst cycle descriptor is available as the status of the indication.

`vmm_rw_xactor::SINGLE_DONE`

Indicates that a single cycle has been completed. The single cycle descriptor is available as the status of the indication.

Example

Example B-297

```
class my_rw_master extends vmm_rw_xactor;
   ...
   typedef enum {BURST_DONE = 99990,SINGLE_DONE}
notifications_e;
   ...
   function new(string                inst,
                int unsigned          stream_id,
                my_rw_access_channel exec_chan = null);
```

```
      super.new("RW Master", inst, stream_id, exec_chan);
   endfunction
   ...
endclass
```

vmm_rw_xactor::pre_single()

AOP callback invoked before the execution of a single cycle.

SystemVerilog

N/A

OpenVera

```
task pre_single_t(vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a single cycle, before the corresponding OOP callback method is invoked, and before the `vmm_rw_xactor::execute_single()` method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

vmm_rw_xactor::pre_burst()

AOP callback invoked before the execution of a burst cycle.

SystemVerilog

N/A

OpenVera

```
task pre_burst_t(vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a burst cycle, before the corresponding OOP callback method is invoked, and before the "vmm_rw_xactor::execute_burst()" method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

vmm_rw_xactor::post_single()

AOP callback invoked after the execution of a single cycle.

SystemVerilog

```
N/A
```

OpenVera

```
task post_single_t(vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a single cycle, after the "vmm_rw_xactor::execute_single()" method has returned, but before the corresponding OOP callback method is invoked. The transaction descriptor must not be modified.

vmm_rw_xactor::post_burst()

AOP callback invoked after the execution of a burst cycle.

SystemVerilog

N/A

OpenVera

```
task post_burst_t(vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a burst cycle, after the "vmm_rw_xactor::execute_burst()" method has returned, but before the corresponding OOP callback method is invoked. The transaction descriptor must not be modified.

vmm_rw_xactor_callbacks

Summary

- vmm_rw_xactor_callbacks::pre_single() ..... page B-575
- vmm_rw_xactor_callbacks::pre_burst() ..... page B-577
- vmm_rw_xactor_callbacks::post_single() ..... page B-579
- vmm_rw_xactor_callbacks::post_burst() ..... page B-581

vmm_rw_xactor_callbacks::pre_single()

OOP callback invoked before the execution of a single cycle.

SystemVerilog

```
virtual task pre_single(vmm_rw_xactor xact,
                        vmm_rw_access tr)
```

OpenVera

```
virtual task pre_single_t(vmm_rw_xactor xact,
                          vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a single cycle, before the "vmm_rw_xactor::execute_single()" method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-298

```
class my_xactor extends vmm_rw_xactor;
   ...
   my_data data; //my_data extends from vmm_rw_access
   this.wb.exec_chan.put(data);
   //This transaction object "data" is not a burst so it
main() of vmm_rw_xactor
   //will call execute_single() method and pre_single() &
post_single() Methods
   //will be invoked.

   virtual function void start_xactor();
      super.start_xactor();
```

```
      this.wb.start_xactor();
      ...
   endfunction: start_xactor
   ...
endclass
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task pre_single(vmm_rw_xactor xact,
                              vmm_rw_access tr);
         `vmm_note(log,{"pre_single method is called for
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
   ...
endprogram
```

vmm_rw_xactor_callbacks::pre_burst()

OOP callback invoked before the execution of a burst cycle.

SystemVerilog

```
virtual task pre_burst(vmm_rw_xactor xact,
                       vmm_rw_access tr)
```

OpenVera

```
virtual task pre_burst_t(vmm_rw_xactor xact,
                         vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a burst cycle, before the "vmm_rw_xactor::execute_burst()" method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-299

```
class my_xactor extends vmm_rw_xactor;
   ...
   my_rw_burst burst;
   this.wb.exec_chan.put(burst);
  //This transaction object "burst" is burst type instance
so it main() of
   //vmm_rw_xactor will call execute_burst() method and
pre_burst() & post_burst()
   //Methods will be invoked.
   ...
endclass
...
```

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task pre_burst(vmm_rw_xactor xact,
                             vmm_rw_access tr);
         `vmm_note(log,{"pre_burst method is called for
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
   ...
endprogram
```

vmm_rw_xactor_callbacks::post_single()

OOP callback invoked after the execution of a single cycle.

SystemVerilog

```
virtual task post_single(vmm_rw_xactor xact,
                         vmm_rw_access tr)
```

OpenVera

```
virtual task post_single_t(vmm_rw_xactor xact,
                           vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a single cycle, after the "vmm_rw_xactor::execute_single()" method has returned. The transaction descriptor must not be modified.

Example

Example B-300

```
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task post_single(vmm_rw_xactor xact,
                               vmm_rw_access tr);
         `vmm_note(log,{"post_single method is called for
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
```

```
    env.my_xactor.append_callback(cb);
    ...
endprogram
```

vmm_rw_xactor_callbacks::post_burst()

OOP callback invoked after the execution of a burst cycle.

SystemVerilog

```
virtual task post_burst(vmm_rw_xactor xact,
                        vmm_rw_access tr)
```

OpenVera

```
virtual task post_burst_t(vmm_rw_xactor xact,
                          vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a burst cycle, after the "vmm_rw_xactor::execute_burst()" method has returned. The transaction descriptor must not be modified.

Example

Example B-301

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task post_burst(vmm_rw_xactor xact,
                              vmm_rw_access tr);
         `vmm_note(log,{"post_burst method is called for
vmm_ral_xactor_",
                    "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
```

```
    ...
endprogram
```

C

RAL C API Macros and Functions

This appendix provides detailed documentation of the macros and functions that compose the C API to a RAL model.

The C API is composed of two sets of macros and functions. A first set is generic to all RAL models and is provided by the RAL package. The names of the macros and functions in the first set is static and identical for all RAL models. A second set is specific to a RAL model and is generated by ralgen. The names of the macros and functions in the second set contain DUT-specific identifiers, as explained in "API Naming" .

For each set, this appendix documents the macros and functions in alphabetical order, and provides a summary of all available macros and functions in each set, with cross references to their detailed documentation.

This section contains the following topics:

- “Generic API” on page 2
- “DUT-Specific Generated API” on page 5

Generic API

Macros and functions that are generic across all RAL models. They are provided by the `vmm_ral.h` file.

The names of the macros and functions are as specified.

Macros and functions are not identified as being macros or functions because the implementation of a specific procedure in the C API may use a macro in one implementation and a function in another. For example, the pure-C implementation of the C API tends to use mainly macros to improve runtime performance of the final object code, whereas the DPI implementation tends to use functions.

Summary

- “VMM_RAL_DATA_BUS_WIDTH” on page 2
- “VMM_RAL_ADDR_GRANULARITY” on page 3
- “vmm_ral_status” on page 4

VMM_RAL_DATA_BUS_WIDTH

Number of bytes in a physical access.

Usage

```
% gcc ... -DVMM_RAL_DATA_BUS_WIDTH=2 ...
```

Description

Specifies the number of bytes transferred in a single physical read or write access in the target system.

If left undefined, physical accesses are assumed to be 4 bytes (32 bits).

This macro must be defined before the `vmm_ral.h` file is compiled and must be consistently defined for all compiled files that include the `vmm_ral.h` file.

Example

Example C-1

```
#define VMM_RAL_DATA_BUS_WIDTH 2
#include "vmm_ral.h"
```

VMM_RAL_ADDR_GRANULARITY

Number of bytes individually addressed.

Usage

```
% gcc ... -DVMM_RAL_ADDR_GRANULARITY=1 ...
```

Description

Specifies the number of bytes that can be individually addressed. In a 1-byte (BYTE) granularity system, each increment of the physical address accesses the next byte in the address space. In a 2-byte (WORD) granularity system, each increment of the physical address

accesses the next 16-bit value in the address space. In a 4-byte (DWORD) granularity system, each increment of the physical address accesses the next 32-bit value in the address space.

A 1-byte granularity system with DWORD-aligned accesses is equivalent to a 4-byte granularity system with the address values shifted by 2-bits left.

If left undefined, address granularity is assumed to be 4 bytes (DWORD access).

This macro must be defined before the `vmm_ral.h` file is compiled and must be consistently defined for all compiled files that include the `vmm_ral.h` file.

Example

Example C-2

```
#define VMM_RAL_ADDR_GRANULARITY 1
#include "vmm_ral.h"
```

---

vmm_ral_status

Status of the last RAL operation.

Usage

```
int vmm_ral_status;
```

Description

Global variable containing the status of the last RAL operation. A value of 0 indicates a successful completion. Any other value indicates an error has occurred.

Example

Example C-3

```
ral_write_X_in_myblk(blk, &x);
if (vmm_ral_status) printf(stderr, "ERROR: ...");
```

DUT-Specific Generated API

Macros and functions that are specific to a RAL model. These macros are used to access a specific field, register, or memory location in the DUT.

The macros and functions are generated by ralgen and contain DUT-specific identifiers. Their names are unique to each DUT and are documented in a generic fashion.

Macros and functions are not identified as being macros or functions because the implementation of a specific procedure in the C API may use a macro in one implementation and a function in another. For example, the pure-C implementation of the C API tends to use mainly macros to improve runtime performance of the final object code, whereas the DPI implementation tends to use functions.

API Naming

The name of the DUT-specific macros and functions are specified using placeholders that, when used, must be replaced with DUT-specific identifiers. These placeholders are specified in *italic* font, enclosed between angle brackets ('<' and '>'), and must be entirely replaced, including the angle brackets. The DUT-specific identifier must be concatenated with the preceding identifier and any following underscore character. For example, the API specified as

`ral_addr_of_<reg>_in_<blk>` must be used as `ral_addr_of_R1_in_myblk` to access a register named `R1` instantiated in a block named `myblk`.

Names are “scoped” to avoid collision between identical names declared in different contexts. A “scoped” name is prefixed with the names of all of its enclosing RAL elements separated with underscores. For example, the RALF file shown in Example C-4, defines the following “scoped” names:

rg
rg_f
sys
sys_sub
sys_sub_blk
sys_sub_blk_r

Note that domains do not create a scope.

Example C-4 RAL file with “scoped” names

```
register rg {
   field f { ... }
}

system sys {
   system sub {
      block blk {
         register rg=r;
      }
   }
}
```

Placeholders

<sys>

The name of a system or subsystem targeted by the access.

<blk>

The name of a block targeted by the access.

<reg>

Name of the register to be accessed.

<mem>

Name of the memory to be accessed.

<fld>

Name of the field to be accessed.

Summary

- “ral_addr_of_<sys>_in_<sys>()” on page 8
- “ral_addr_of_<blk>_in_<sys>()” on page 8
- “ral_addr_of_<reg>_in_<blk>()” on page 9
- “ral_addr_of_<mem>_in_<blk>()” on page 10
- “ral_read_<reg>_in_<blk>()” on page 11
- “ral_write_<reg>_in_<blk>()” on page 12
- “ral_read_<fld>_in_<reg>()” on page 13
- “ral_write_<fld>_in_<reg>()” on page 13
- “ral_read_<fld>_in_<blk>()” on page 14
- “ral_write_<fld>_in_<blk>()” on page 14

ral_addr_of_<*sys*>_in_<*sys*>()

Address of a subsystem in a system.

Usage

```
size_t ral_addr_of_<sys>_in_<sys>(size_t sys);

size_t ral_addr_of_<sys>_in_<sys>(size_t sys, int idx);
```

Description

Returns the address of the specified subsystem in the specified immediately-enclosing system. The address of the immediately enclosing system is specified as the first argument.

If the subsystem is instantiated as a subsystem array in the immediately enclosing system, the index of the subsystem of interest must be specified as the second argument.

Example

Example C-5

```
void C_func_name(size_t top)
{
   size_t s1_base;
   s1_base = ral_addr_of_s2_in_s1(
                ral_addr_of_s1_in_top(top));
}
```

ral_addr_of_<*blk*>_in_<*sys*>()

Address of a block in a system.

Usage

```
size_t ral_addr_of_<blk>_in_<sys>(size_t sys);

size_t ral_addr_of_<blk>_in_<sys>(size_t sys, int idx);
```

Description

Returns the address of the specified block in the specified immediately-enclosing system. The address of the immediately enclosing system is specified as the first argument.

If the block is instantiated as a block array in the immediately enclosing system, the index of the block of interest must be specified as the second argument.

Example

Example C-6

```
size_t b1_base =
   ral_addr_of_b_in_s1(ral_addr_of_s1_in_top(top), 1);
```

---

ral_addr_of_<*reg*>_in_<*blk*>()

Address of a register in a block.

Usage

```
size_t ral_addr_of_<reg>_in_<blk>(size_t blk);

size_t ral_addr_of_<reg>_in_<blk>(size_t blk, int idx);
```

Description

Returns the address of the specified register in the specified immediately-enclosing block. The address of the immediately enclosing block is specified as the first argument.

If the register is instantiated as a register array in the immediately enclosing block, the index of the register of interest must be specified as the second argument.

Example

Example C-7

```
size_t r1_b3addr = ral_addr_of_r1_in_top_b(
                       ral_addr_of_top_b_in_top(top, 3));
```

---

ral_addr_of_<*mem*>_in_<*blk*>()

Base address of a memory in a block.

Usage

```
size_t ral_addr_of_<mem>_in_<blk>(size_t blk);
```

Description

Returns the base address of the specified memory in the specified immediately-enclosing block. The address of the immediately enclosing block is specified as the first argument.

Example

Example C-8

```
size_t m_addr = ral_addr_of_m_in_b(top);
```

ral_read_<*reg*>_in_<*blk*>()

Read an entire register in a block.

Usage

```
void ral_read_<reg>_in_<blk>(size_t blk, int* bfr);

void ral_read_<reg>_in_<blk>(size_t blk,
     int idx,
     int* bfr);
```

Description

Read the entire content of the specified register in the specified immediately-enclosing block. The address of the immediately enclosing block is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the register. The content of the register is stored in little endian order.

If the register is instantiated as a register array in the immediately enclosing block, the index of the register of interest must be specified as the second argument.

The status of the operation is stored in vmm_ral_status.

Example

Example C-9

```
int r[3];
ral_read_r_in_b(addr_of_b_in_s(s), r);
```

ral_write_<*reg*>_in_<*blk*>()

Write an entire register in a block.

Usage

```
void ral_write_<reg>_in_<blk>(size_t blk, int* bfr);

void ral_write_<reg>_in_<blk>(size_t blk,
     int idx,
     int* bfr);
```

Description

Write the entire content of the specified register in the specified immediately-enclosing block. The address of the immediately enclosing block is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the register. The content of the register is written in little-endian order.

If the register is instantiated as a register array in the immediately enclosing block, the index of the register of interest must be specified as the second argument.

The status of the operation is stored in vmm_ral_status.

Example

Example C-10

```
int r;
ral_write_r_in_b(b, &r);
```

ral_read_<*fld*>_in_<*reg*>()

Read a field in a register.

Usage

```
void ral_read_<fld>_in_<reg>(size_t reg, int* bfr);
```

Description

Read the specified field in the specified register. The address of the register is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the field. The content of the field is stored in little-endian order.

The status of the operation is stored in vmm_ral_status.

ral_write_<*fld*>_in_<*reg*>()

Write a field in a register.

Usage

```
void ral_write_<fld>_in_<reg>(size_t reg, int* bfr);
```

Description

Write the specified field in the specified register. The address of the register is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the field. The content of the field is written in little-endian order.

The status of the operation is stored in vmm_ral_status.

---

ral_read_<*fld*>_in_<*blk*>()

Read a field in a block.

Usage

```
void ral_read_<fld>_in_<blk>(size_t blk, int* bfr);
```

Description

Read the specified field in the specified immediately-enclosing block. The address of the immediately enclosing block is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the field. The content of the field is stored in little-endian order.

The status of the operation is stored in vmm_ral_status.

This function is available only if the field is unique within the block.

---

ral_write_<*fld*>_in_<*blk*>()

Write a field in a block.

Usage

```
void ral_write_<fld>_in_<blk>(size_t blk, int* bfr);
```

Description

Write the particular field in the specified, immediately-enclosing, block. The address of the immediately enclosing block is specified as the first argument.

The last argument is a pointer to an array of `int` that is assumed to be long enough to hold the entire content of the field. The content of the field is written in little-endian order.

The status of the operation is stored in vmm_ral_status.

This function is available only if the field is unique within the block.

D

RALF C++ API Class Reference

This appendix describes the C++ classes and methods that comprise the RALF C++ API.

RALF C++ API classes are encapsulated inside namespace "ralf".

Class Summary

- ralf::vmm_ralf_exception Class Reference .....page D-19
- ralf::vmm_ralf_error Class Reference .....page D-22
- ralf::vmm_ralf_invalid_oprn Class Reference .....page D-24
- ralf::vmm_ralf_hdl_path::const_hdl_path_iter Class Referencepage D-26
- ralf::vmm_ralf_hdl_path Class Reference .....page D-32
- ralf::vmm_ralf_constraint Class Reference .....page D-35
- ralf::vmm_ralf_fld_typ Class Reference .....page D-38
- ralf::vmm_ralf_fld_instance Class Reference .....page D-45
- ralf::vmm_ralf_blk_component_instance Class Reference .....page D-50
- ralf::vmm_ralf_reg_typ::const_fld_iter Class Reference ....page D-55
- ralf::vmm_ralf_reg_typ Class Reference .....page D-60
- ralf::vmm_ralf_reg_instance Class Reference .....page D-69
- ralf::vmm_ralf_mem_typ Class Reference .....page D-77
- ralf::vmm_ralf_mem_instance Class Reference .....page D-85
- ralf::vmm_ralf_rfile_typ::const_reg_iter Class Reference ..page D-90
- ralf::vmm_ralf_rfile_typ Class Reference .....page D-95
- ralf::vmm_ralf_rfile_instance Class Reference .....page D-100
- ralf::vmm_ralf_domain Class Reference .....page D-105

- ralf::vmm_ralf_blk_domain::const_component_iter Class Referencepage D-110
- ralf::vmm_ralf_blk_domain Class Reference .....page D-115
- ralf::vmm_ralf_blk_or_sys_typ Class Reference .....page D-117
- ralf::vmm_ralf_blk_typ::const_domain_iter Class Reference page D-120
- ralf::vmm_ralf_blk_typ Class Reference .....page D-125
- ralf::vmm_ralf_domain_instance Class Reference .....page D-127
- ralf::vmm_ralf_blk_domain_instance Class Reference .....page D-136
- ralf::vmm_ralf_sys_domain::const_component_iter Class Referencepage D-138
- ralf::vmm_ralf_sys_domain Class Reference .....page D-143
- ralf::vmm_ralf_sys_typ::const_domain_iter Class Reference page D-145
- ralf::vmm_ralf_sys_typ Class Reference .....page D-150
- ralf::vmm_ralf_sys_domain_instance Class Reference .....page D-152
- ralf::vmm_ralf Class Reference .....page D-154

ralf::vmm_ralf_exception Class Reference

RALF exception class.

Synopsis

```
namespace ralf {
   class vmm_ralf_exception : public std::exception {
      …
   };
};
```

Description

Top level RALF exception class.

Any user defined exception class should be derived from this class.

Public Member Functions

```
explicit vmm_ralf_exception (const char * msg = NULL);
virtual const std::string &
getMessage() const throw();
```

Summary

- ralf::vmm_ralf_exception::vmm_ralf_exception(…) .....page D-20
- ralf::vmm_ralf_exception::getMessage() .....page D-21

ralf::vmm_ralf_exception::vmm_ralf_exception(...)

Default constructor.

Synopsis

```
explicit vmm_ralf_exception(const char * msg = NULL);
```

Description

This constructor shall create an exception object of class `vmm_ralf_exception` with the specified exception `msg`.

Return Value

N/A

ralf::vmm_ralf_exception::getMessage()

Gets the exception message.

Synopsis

```
virtual const std::string & getMessage() const throw();
```

Description

Gets the message, if any, associated with the exception.

Return Value

Reference to string constant (storing the exception message).

ralf::vmm_ralf_error Class Reference

RALF exception class for errors.

Synopsis

```
namespace ralf {
   class vmm_ralf_error : public vmm_ralf_exception {
      …
   };
};
```

Description

Any user defined error class should be derived from this class.

Public Member Functions

```
explicit vmm_ralf_error(const char * error_msg = NULL);
```

Summary

- ralf::vmm_ralf_error::vmm_ralf_error(…) .....page D-23

ralf::vmm_ralf_error::vmm_ralf_error(...)

Default constructor.

Synopsis

```
explicit vmm_ralf_error(const char * error_msg = NULL);
```

Description

This constructor shall create an exception object of class `vmm_ralf_error` with the specified error message `error_msg`.

Return Value

N/A

ralf::vmm_ralf_invalid_oprn Class Reference

RALF exception class for errors of type "Invalid operation performed".

Synopsis

```
namespace ralf {
   class vmm_ralf_invalid_oprn : public vmm_ralf_error {
      …
   };
};
```

Description

Any user defined exception class of this type should be derived from this class.

Public Member Functions

```
explicit vmm_ralf_invalid_oprn(const char * why = NULL);
```

Summary

- ralf::vmm_ralf_invalid_oprn::vmm_ralf_invalid_oprn(…) .....page D-25

ralf::vmm_ralf_invalid_oprn::vmm_ralf_invalid_oprn(...)

Default constructor.

Synopsis

```
explicit vmm_ralf_invalid_oprn(const char * why = NULL);
```

Description

This constructor shall create an exception object of class `vmm_ralf_invalid_oprn` with the specified reason `why`.

Return Value

N/A

ralf::vmm_ralf_hdl_path::const_hdl_path_iter Class Reference

RALF HDL path iterator.

Synopsis

```
namespace ralf {
   class vmm_ralf_hdl_path {
      class const_hdl_path_iter {
         …
      };
   };
};
```

Description

HDL path can either be a concatenated expression of multiple RTL signals like

{ *signal1* , *signal2* , … *signalN* }

Or HDL path can just be one single signal like *signal1*.

So, the data structure representing HDL path, needed to be generic enough for being able to support both these kinds of HDL path expressions.

The data structure chosen (for representing HDL path) is kind of a sequence of strings. The sequence would be the same as it is in the concatenation expression. So, for the concatenation expression shown above, the sequence would be: *signal1*, *signal2*, … *signalN*.

And, for the case, where HDL path has got just one signal (no concatenation expression), the sequence would just have one string/ signal.

For either case, strings (storing the individual signals) can be accessed using this iterator class `const_hdl_path_iter`.

As the nomenclature (`const_*`) of this iterator class indicates, this class cannot be used for modifying the objects pointed to.

Public Member Functions

```
virtual const_hdl_path_iter & operator++() = 0;
virtual const std::string * operator*() const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator++() page D-28
- ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator*() .page D-29
- ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isBegin() ...page D-30
- ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isEnd() .....page D-31

ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_hdl_path_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next HDL path signal, assuming that the HDL path is actually a concatenation expression. If the iterator is already pointing to the last HDL path signal, invocation of this function would result in this iterator pointing to one past the last HDL path signal, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf::vmm_ralf_hdl_path::const_hdl_path_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const std::string * operator*() const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the HDL path signal string (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to a string constant (storing the `hdl_path`).

ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isBegin()

Checks if the iterator is pointing to the first element.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Return Value

Returns true if the iterator is pointing to the first element. Else returns false.

ralf::vmm_ralf_hdl_path::const_hdl_path_iter::isEnd()

Checks if the iterator is pointing to the end.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_hdl_path Class Reference

RALF HDL path.

Synopsis

```
namespace ralf {
   class vmm_ralf_hdl_path {
      …
   };
};
```

Description

Descriptor of RALF HDL path

Public Member Functions

```
virtual const_hdl_path_iter * createHdlPathIter() const = 0;
virtual bool isVhdlPath() const = 0;
```

Summary

- ralf::vmm_ralf_hdl_path::createHdlPathIter() .....page D-33
- ralf::vmm_ralf_hdl_path::isVhdlPath() .....page D-34

ralf::vmm_ralf_hdl_path::createHdlPathIter()

Creates an HDL path iterator.

Synopsis

```
virtual const_hdl_path_iter * createHdlPathIter() const = 0;
```

Description

Creates an HDL path iterator to iterate over the RTL signals composing the HDL path.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first or left-most RTL signal.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_hdl_path::isVhdlPath()

Checks if the target signal has been implemented in VHDL.

Synopsis

```
virtual bool isVhdlPath() const = 0;
```

Description

Checks if the target variable or signal has been implemented in VHDL.

Verilog implementation is assumed by default.

Return Value

Returns true if implemented in VHDL, else false.

ralf::vmm_ralf_constraint Class Reference

RALF constraint.

Synopsis

```
namespace ralf {
   class vmm_ralf_constraint {
      …
   };
};
```

Description

Descriptor of RALF constraint specification

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const std::string & getBody() const = 0;
```

Summary

- ralf::vmm_ralf_constraint::getName() .....page D-36
- ralf::vmm_ralf_constraint::getBody() .....page D-37

ralf::vmm_ralf_constraint::getName()

Gets the name of the constraint.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the constraint.

Return Value

Reference to the string constant (storing the constraint name).

ralf::vmm_ralf_constraint::getBody()

Gets the constraint body.

Synopsis

```
virtual const std::string & getBody() const = 0;
```

Description

Gets the body of the constraint

Return Value

Reference to the string constant (storing the constraint body).

ralf::vmm_ralf_fld_typ Class Reference

RALF `field` type.

Synopsis

```
namespace ralf {
   class vmm_ralf_fld_typ {
      …
   };
};
```

Description

Descriptor of a RALF `field` type.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const size_t & getNBits() const = 0;
virtual access_e getAccess() const = 0;
virtual const std::string & getHardReset() const = 0;
virtual const std::string & getSoftReset() const = 0;
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Summary

- ralf::vmm_ralf_fld_typ::getName() .....page D-39
- ralf::vmm_ralf_fld_typ::getNBits() .....page D-40
- ralf::vmm_ralf_fld_typ::getAccess() .....page D-41
- ralf::vmm_ralf_fld_typ::getHardReset() .....page D-42
- ralf::vmm_ralf_fld_typ::getSoftReset() .....page D-43
- ralf::vmm_ralf_fld_typ::getConstraint() .....page D-44

ralf::vmm_ralf_fld_typ::getName()

Gets the name of the RALF `field` type.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF `field` type.

Return Value

Reference to string constant (storing name of the RALF `field` type).

ralf::vmm_ralf_fld_typ::getNBits()

Gets the field size in bits.

Synopsis

```
virtual const size_t & getNBits() const = 0;
```

Description

Gets the field size in bits.

Return Value

Field size in bits.

ralf::vmm_ralf_fld_typ::getAccess()

Gets access specification.

Synopsis

```
enum access_e {
      RW,
      RO,
      WO,
      W1,
      RU,
      RC,
      W1C,
      A0,
      A1,
      DC,
      OTHER,
      USER0,
      USER1,
      USER2,
      USER3
};
virtual access_e getAccess() const = 0;
```

Description

Gets access specification of the RALF `field` type.

Return Value

Enumeration constant of type `access_e` to indicate the field access specification.

ralf::vmm_ralf_fld_typ::getHardReset()

Gets hard reset value.

Synopsis

```
virtual const std::string & getHardReset() const = 0;
```

Description

Gets hard reset value of the RALF `field` type.

Return Value

Reference to string constant (storing the hard reset value).

ralf::vmm_ralf_fld_typ::getSoftReset()

Gets soft reset value.

Synopsis

```
virtual const std::string & getSoftReset() const = 0;
```

Description

Gets the soft reset value of the RALF `field` type.

Return Value

Reference to string constant (storing the soft reset value).

ralf::vmm_ralf_fld_typ::getConstraint()

Gets the constraint.

Synopsis

```
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Description

Gets the constraint, if any specified, in the corresponding RALF description of the RALF `field` type.

Return Value

Pointer to const constraint data structure, if the constraint was specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_fld_instance Class Reference

RALF field instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_fld_instance {
      …
   };
};
```

Description

RALF descriptor of an instantiated field, as instantiated in a register.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const vmm_ralf_fld_typ & getType() const = 0;
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
virtual const size_t & getLSBPosition() const = 0
```

Summary

- ralf::vmm_ralf_fld_instance::getName() .....page D-46
- ralf::vmm_ralf_fld_instance::getName() .....page D-46
- ralf::vmm_ralf_fld_instance::getType() .....page D-47
- ralf::vmm_ralf_fld_instance::getHdlPath() .....page D-48
- ralf::vmm_ralf_fld_instance::getLSBPosition() .....page D-49

ralf::vmm_ralf_fld_instance::getName()

Gets the name of the RALF field instance.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF field instance.

Return Value

Reference to string constant (storing the instance name).

ralf::vmm_ralf_fld_instance::getType()

Gets the corresponding RALF `field` type.

Synopsis

```
virtual const vmm_ralf_fld_typ & getType() const = 0;
```

Description

Gets the corresponding RALF `field` type data structure.

Return Value

Reference to const RALF `field` type data structure.

ralf::vmm_ralf_fld_instance::getHdlPath()

Gets the HDL path.

Synopsis

```
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
```

Description

Gets the HDL path, if any specified, in the corresponding RALF description.

Return Value

Pointer to const HDL path data structure, if the HDL path has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_fld_instance::getLSBPosition()

Gets the LSB position of the field.

Synopsis

```
virtual const size_t & getLSBPosition() const = 0;
```

Description

Gets the LSB position of the field in the parent register.

Return Value

LSB position of the field instance.

ralf::vmm_ralf_blk_component_instance Class Reference

Abstract, common base class of all RALF block components.

Synopsis

```
namespace ralf {
   class vmm_ralf_blk_component_instance {
      …
   };
};
```

Description

Abstract, common base class of all direct structural components of a RALF block.

Direct structural components of RALF block are `register`, `memory`, and `regfile`. A field is not a direct structural component of block since it cannot be directly instantiated (structurally in RALF) inside a block.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
virtual const size_t & getOffset() const = 0;
virtual const size_t & getAddrSpaceSize() const = 0;
```

Summary

- ralf::vmm_ralf_blk_component_instance::getName() .....page D-51
- ralf::vmm_ralf_blk_component_instance::getHdlPath() .....page D-52
- ralf::vmm_ralf_blk_component_instance::getOffset() .....page D-53
- ralf::vmm_ralf_blk_component_instance::getAddrSpaceSize() .page D-54

ralf::vmm_ralf_blk_component_instance::getName()

Gets the instance name.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the instance name of the RALF block component.

Return Value

Reference to string constant (storing the instance name).

ralf::vmm_ralf_blk_component_instance::getHdlPath()

Gets the HDL path.

Synopsis

```
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
```

Description

Gets the HDL path, if any specified, in the corresponding RALF description.

Return Value

Pointer to const HDL path data structure, if the HDL path has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_blk_component_instance::getOffset()

Gets the offset.

Synopsis

```
virtual const size_t & getOffset() const = 0;
```

Description

Gets the offset/address of the block component's instance in the parent RALF block.

Return Value

Offset/address of the instance.

ralf::vmm_ralf_blk_component_instance::getAddrSpaceSize()

Gets the address space size.

Synopsis

```
virtual const size_t & getAddrSpaceSize() const = 0;
```

Description

Gets the address space size of the instance in the parent block domain.

Return Value

Address space size in the parent block domain.

ralf::vmm_ralf_reg_typ::const_fld_iter Class Reference

RALF field iterator.

Synopsis

```
namespace ralf {
   class vmm_ralf_reg_typ {
      class const_fld_iter {
         …
      };
   };
};
```

Description

Iterator to iterate over all the fields of a RALF register, in the same sequence in which the fields are defined in the corresponding RALF description.

Public Member Functions

```
virtual const_fld_iter & operator++() = 0;
virtual const vmm_ralf_fld_instance * operator*() const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf::vmm_ralf_reg_typ::const_fld_iter::operator++() .....page D-56
- ralf::vmm_ralf_reg_typ::const_fld_iter::operator*() .....page D-57
- ralf::vmm_ralf_reg_typ::const_fld_iter::isBegin() .....page D-58
- ralf::vmm_ralf_reg_typ::const_fld_iter::isEnd() .....page D-59

ralf::vmm_ralf_reg_typ::const_fld_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_fld_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next RALF field instance. If the iterator is already pointing to the last RALF field instance, then invocation of this function would result in this iterator pointing to one past the last RALF field instance, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf::vmm_ralf_reg_typ::const_fld_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_fld_instance * operator*() const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the RALF field instance descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to a const RALF field instance descriptor.

ralf::vmm_ralf_reg_typ::const_fld_iter::isBegin()

Checks if the iterator is pointing to the first RALF field.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first RALF field.

Return Value

Returns true if the iterator is pointing to the first RALF field. Else returns false.

ralf::vmm_ralf_reg_typ::const_fld_iter::isEnd()

Checks if the iterator is pointing to the end.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last RALF field) of the sequence.

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_reg_typ Class Reference

RALF `register` type.

Synopsis

```
namespace ralf {
   class vmm_ralf_reg_typ {
      …
   };
};
```

Description

Descriptor of a RALF `register` type.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const std::string & getScopeName() const = 0;
virtual const size_t & getNBytes() const = 0;
virtual const_fld_iter * createFldIter() const = 0;
virtual bool isLeft2Right() const = 0;
virtual bool isShared() const = 0;
virtual const vmm_ralf_constraint * getConstraint() const =
0;
virtual const vmm_ralf_hdl_path * getSharedHdlPath() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Summary

- ralf::vmm_ralf_reg_typ::getName() .....page D-61
- ralf::vmm_ralf_reg_typ::getScopeName() .....page D-62
- ralf::vmm_ralf_reg_typ::getNBytes() .....page D-63
- ralf::vmm_ralf_reg_typ::createFldIter() .....page D-64
- ralf::vmm_ralf_reg_typ::isLeft2Right() .....page D-65
- ralf::vmm_ralf_reg_typ::isShared() .....page D-66
- ralf::vmm_ralf_reg_typ::getConstraint() .....page D-67
- ralf::vmm_ralf_reg_typ::getSharedHdlPath() .....page D-68

ralf::vmm_ralf_reg_typ::getName()

Gets the name.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF `register` type.

Return Value

Reference to string constant (storing the name of the RALF `register` type).

ralf::vmm_ralf_reg_typ::getScopeName()

Gets the scope name.

Synopsis

```
virtual const std::string & getScopeName() const = 0;
```

Description

Gets the scope name of the RALF `register` type.

Return Value

Reference to string constant (storing scope name of the RALF `register` type).

ralf::vmm_ralf_reg_typ::getNBytes()

Gets the size of the register in bytes.

Synopsis

```
virtual const size_t & getNBytes() const = 0;
```

Description

Gets the size of the register in bytes.

Return Value

Size of the register (in bytes).

ralf::vmm_ralf_reg_typ::createFldIter()

Creates the field iterator.

Synopsis

```
virtual const_fld_iter * createFldIter() const = 0;
```

Description

Creates an iterator to iterate over the list of all field instances in a register.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first field instance.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_reg_typ::isLeft2Right()

Checks if the field alignment is left to right.

Synopsis

```
virtual bool isLeft2Right() const = 0;
```

Description

Checks if alignment of field instances in the register is from left (MSB) to right (LSB).

Return Value

Returns true if yes, else false.

ralf::vmm_ralf_reg_typ::isShared()

Checks if the register is shared.

Synopsis

```
virtual bool isShared() const = 0;
```

Description

Checks if the register is sharable among multiple domains of a block.

Return Value

Returns true if sharable, else false.

ralf::vmm_ralf_reg_typ::getConstraint()

Gets the constraint.

Synopsis

```
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Description

Gets the constraint, if any specified, in the corresponding RALF description.

Return Value

Pointer to the const constraint data structure, if the constraint has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_reg_typ::getSharedHdlPath()

Gets shared HDL path.

Synopsis

```
virtual const vmm_ralf_hdl_path * getSharedHdlPath()
   const throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the shared HDL path, if any specified, in the corresponding RALF description.

Throws exception `vmm_ralf_invalid_oprn()` if this method is invoked for a register that is not shared.

Return Value

Pointer to `const hdl_path` data structure, if shared `hdl_path` has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_reg_instance Class Reference

Descriptor of a RALF register instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_reg_instance
   : public virtual vmm_ralf_blk_component_instance {
      …
   };
};
```

Description

RALF descriptor of an instantiated register, as instantiated in a regfile or block.

Public Member Functions

```
virtual const vmm_ralf_reg_typ & getType() const = 0;
virtual bool isArray() const = 0;
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual bool isUnMapped() const = 0;
virtual const size_t & getOffset() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual shared_access_e getSharedAccess() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Summary

- ralf::vmm_ralf_reg_instance::getType() .....page D-70
- ralf::vmm_ralf_reg_instance::isArray() .....page D-71
- ralf::vmm_ralf_reg_instance::getArraySize() .....page D-72
- ralf::vmm_ralf_reg_instance::getArrayIncr() .....page D-73
- ralf::vmm_ralf_reg_instance::isUnMapped() .....page D-74
- ralf::vmm_ralf_reg_instance::getOffset() .....page D-75
- ralf::vmm_ralf_reg_instance::getSharedAccess() .....page D-76

ralf::vmm_ralf_reg_instance::getType()

Gets the corresponding RALF register type.

Synopsis

```
virtual const vmm_ralf_reg_typ & getType() const = 0;
```

Description

Gets the corresponding RALF register type.

Return Value

Reference to const RALF register type descriptor.

ralf::vmm_ralf_reg_instance::isArray()

Checks if the instance is an array instance.

Synopsis

```
virtual bool isArray() const = 0;
```

Description

Checks if the instance is an array instance.

Return Value

Returns true if yes, else false.

ralf::vmm_ralf_reg_instance::getArraySize()

Gets the size of the array.

Synopsis

```
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the size of the array.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Size of the array.

ralf::vmm_ralf_reg_instance::getArrayIncr()

Gets the array increment.

Synopsis

```
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the array increment.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Array increment.

ralf::vmm_ralf_reg_instance::isUnMapped()

Checks if the register instance is unmapped in the parent block domain.

Synopsis

```
virtual bool isUnMapped() const = 0;
```

Description

Checks if the register instance is unmapped in the parent block domain.

Return Value

Returns true if unmapped, else false.

ralf::vmm_ralf_reg_instance::getOffset()

Gets the offset.

Synopsis

```
virtual const size_t & getOffset() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the offset/address of the instance in the parent RALF block domain or regfile.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on an unmapped register instance.

Return Value

Offset/address of the instance.

ralf::vmm_ralf_reg_instance::getSharedAccess()

Gets the shared access specification.

Synopsis

```
enum shared_access_e {
      DEFAULT,    //!< Default is RW shared access.
      READ,
      WRITE
};
virtual shared_access_e getSharedAccess() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the shared access specification of the shared register instance. This access specification indicates if the shared register has been instantiated to have read-only, write-only or default read-write shared access in the parent block domain.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a register which is not sharable.

Return Value

Enumeration constant of type `shared_access_e` to indicate the shared access specification.

ralf::vmm_ralf_mem_typ Class Reference

RALF `memory` type.

Synopsis

```
namespace ralf {
   class vmm_ralf_mem_typ {
      …
   };
};
```

Description

Descriptor of a RALF `memory` type.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const std::string & getScopeName() const = 0;
virtual const size_t & getSize() const = 0;
virtual const size_t & getNBits() const = 0;
virtual access_e getAccess() const = 0;
virtual bool isShared() const = 0;
virtual const vmm_ralf_hdl_path * getSharedHdlPath() const
= 0;
```

Summary

- ralf::vmm_ralf_mem_typ::getName() .....page D-78
- ralf::vmm_ralf_mem_typ::getScopeName() .....page D-79
- ralf::vmm_ralf_mem_typ::getSize() .....page D-80
- ralf::vmm_ralf_mem_typ::getNBits() .....page D-81
- ralf::vmm_ralf_mem_typ::getAccess() .....page D-82
- ralf::vmm_ralf_mem_typ::isShared() .....page D-83
- ralf::vmm_ralf_mem_typ::getSharedHdlPath() .....page D-84

ralf::vmm_ralf_mem_typ::getName()

Gets the memory name.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF `memory` type.

Return Value

Reference to string constant (storing the name of the RALF `memory` type).

ralf::vmm_ralf_mem_typ::getScopeName()

Gets the scope name.

Synopsis

```
virtual const std::string & getScopeName() const = 0;
```

Description

Gets the scope name of the RALF `memory` type.

Return Value

Reference to string constant (storing the scope name of the RALF `memory` type).

ralf::vmm_ralf_mem_typ::getSize()

Gets the size.

Synopsis

```
virtual const size_t & getSize() const = 0;
```

Description

Gets the size (number of addressable locations) in the memory.

Return Value

Size of the memory.

ralf::vmm_ralf_mem_typ::getNBits()

Gets the number of bits in each memory location.

Synopsis

```
virtual const size_t & getNBits() const = 0;
```

Description

Gets the number of bits in each memory location.

Return Value

Number of bits in each memory location.

ralf::vmm_ralf_mem_typ::getAccess()

Gets the access specification of the memory.

Synopsis

```
enum access_e {
      RW,          //!< Corresponds to RAM
      RO           //!< Corresponds to ROM
};
virtual access_e getAccess() const = 0;
```

Description

`RW` access specification indicates a RAM while `RO` access specification indicates a ROM.

Return Value

Enumeration constant of type `access_e` to indicate if the memory is a RAM or ROM.

ralf::vmm_ralf_mem_typ::isShared()

Checks if the memory is shared.

Synopsis

```
virtual bool isShared() const = 0;
```

Description

Checks if the memory is sharable among multiple domains of a block.

Return Value

Returns true if sharable, else false.

ralf::vmm_ralf_mem_typ::getSharedHdlPath()

Gets shared HDL path.

Synopsis

```
virtual const vmm_ralf_hdl_path * getSharedHdlPath() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the shared HDL path, if any specified, in the corresponding RALF description.

Throws exception `vmm_ralf_invalid_oprn`(), if this method is invoked on a memory that is not shared.

Return Value

Pointer to const HDL path data structure, if shared HDL path has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_mem_instance Class Reference

RALF memory instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_mem_instance
   : public virtual vmm_ralf_blk_component_instance {
      …
   };
};
```

Description

RALF descriptor of an instantiated memory, as instantiated in a block.

Public Member Functions

```
virtual const vmm_ralf_mem_typ & getType() const = 0;
virtual bool isUnMapped() const = 0;
virtual const size_t & getOffset() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual shared_access_e getSharedAccess() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Summary

- ralf::vmm_ralf_mem_instance::getType() .....page D-86
- ralf::vmm_ralf_mem_instance::isUnMapped() .....page D-87
- ralf::vmm_ralf_mem_instance::getOffset() .....page D-88
- ralf::vmm_ralf_mem_instance::getSharedAccess() .....page D-89

ralf::vmm_ralf_mem_instance::getType()

Gets the corresponding RALF memory type.

Synopsis

```
virtual const vmm_ralf_mem_typ & getType() const = 0;
```

Description

Gets the corresponding RALF memory type.

Return Value

Reference to const RALF memory type descriptor.

ralf::vmm_ralf_mem_instance::isUnMapped()

Checks if the memory instance is unmapped.

Synopsis

```
virtual bool isUnMapped() const = 0;
```

Description

Checks if the memory instance is unmapped in the parent block domain.

Return Value

Returns true if unmapped, else false.

ralf::vmm_ralf_mem_instance::getOffset()

Gets the offset.

Synopsis

```
virtual const size_t & getOffset() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the offset/address of the instance in the parent RALF block.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on an unmapped memory instance.

Return Value

Offset/address of the memory instance.

ralf::vmm_ralf_mem_instance::getSharedAccess()

Gets the shared access specification.

Synopsis

```
enum shared_access_e {
      DEFAULT,    //!< Defauilt is RW shared access.
      READ,
      WRITE
};
virtual shared_access_e getSharedAccess() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the shared access specification of the shared memory instance. This access specification indicates if the shared memory has been instantiated to have read-only, write-only or default read-write access in the parent block domain.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a memory which is not shared.

Return Value

Enumeration constant of type `shared_access_e` to indicate access specification.

ralf::vmm_ralf_rfile_typ::const_reg_iter Class Reference

RALF register iterator.

Synopsis

```
namespace ralf {
   class vmm_ralf_rfile_typ {
      class const_reg_iter {
         …
      };
   };
};
```

Description

Descriptor of RALF register iterator

Public Member Functions

```
virtual const_reg_iter & operator++() = 0;
virtual vmm_ralf_reg_instance * operator*() const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf::vmm_ralf_rfile_typ::const_reg_iter::operator++() ....page D-91
- ralf::vmm_ralf_rfile_typ::const_reg_iter::operator*() .....page D-92
- ralf::vmm_ralf_rfile_typ::const_reg_iter::isBegin() .....page D-93
- ralf::vmm_ralf_rfile_typ::const_reg_iter::isEnd() .....page D-94

ralf::vmm_ralf_rfile_typ::const_reg_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_reg_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next register instance. If the iterator is already pointing to the last register instance, invocation of this function would result in this iterator pointing to one past the last register instance, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf::vmm_ralf_rfile_typ::const_reg_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_reg_instance * operator*() const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the register instance descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to const RALF register instance descriptor.

ralf::vmm_ralf_rfile_typ::const_reg_iter::isBegin()

Checks if the iterator is pointing to the first register instance.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first register instance.

Return Value

Returns true if the iterator is pointing to the first register instance. Else returns false.

ralf::vmm_ralf_rfile_typ::const_reg_iter::isEnd()

Checks if the iterator is pointing to the end.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last register instance).

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_rfile_typ Class Reference

RALF `regfile` type.

Synopsis

```
namespace ralf {
   class vmm_ralf_rfile_typ {
      …
   };
};
```

Description

Descriptor of a RALF `regfile` type.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const std::string & getScopeName() const = 0;
virtual const_reg_iter * createRegIter() const = 0;
virtual const vmm_ralf_constraint * getConstraint() const =
0;
```

Summary

- ralf::vmm_ralf_rfile_typ::getName() .....page D-96
- ralf::vmm_ralf_rfile_typ::getScopeName() .....page D-97
- ralf::vmm_ralf_rfile_typ::createRegIter() .....page D-98
- ralf::vmm_ralf_rfile_typ::getConstraint() .....page D-99

ralf::vmm_ralf_rfile_typ::getName()

Gets the regfile name.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF `regfile` type.

Return Value

Reference to string constant (storing name of the RALF `regfile` type)

ralf::vmm_ralf_rfile_typ::getScopeName()

Gets the scope name.

Synopsis

```
virtual const std::string & getScopeName() const = 0;
```

Description

Gets the scope name of the RALF `regfile` type.

Return Value

Reference to string constant (storing scope name of the RALF `regfile` type).

ralf::vmm_ralf_rfile_typ::createRegIter()

Creates a register iterator.

Synopsis

```
virtual const_reg_iter * createRegIter() const = 0;
```

Description

Creates an iterator to iterate over the list of all register instances in the register file.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first register instance.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_rfile_typ::getConstraint()

Gets the constraint.

Synopsis

```
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Description

Gets the constraint, if any specified, in the corresponding RALF description.

Return Value

Pointer to the const constraint data structure, if the constraint has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_rfile_instance Class Reference

RALF regfile instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_rfile_instance
   : public virtual vmm_ralf_blk_component_instance {
      …
   };
};
```

Description

RALF descriptor of an instantiated regfile, as instantiated in a block.

Public Member Functions

```
virtual const vmm_ralf_rfile_typ & getType() const = 0;
virtual bool isArray() const = 0;
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Summary

- ralf::vmm_ralf_rfile_instance::getType() .....page D-101
- ralf::vmm_ralf_rfile_instance::isArray() .....page D-102
- ralf::vmm_ralf_rfile_instance::getArraySize() .....page D-103
- ralf::vmm_ralf_rfile_instance::getArrayIncr() .....page D-104

ralf::vmm_ralf_rfile_instance::getType()

Gets the corresponding RALF `regfile` type.

Synopsis

```
virtual const vmm_ralf_rfile_typ & getType() const = 0;
```

Description

Gets the corresponding RALF `regfile` type.

Return Value

Reference to const RALF `regfile` type descriptor.

ralf::vmm_ralf_rfile_instance::isArray()

Checks if the instance is an array instance.

Synopsis

```
virtual bool isArray() const = 0;
```

Description

Checks if the instance is an array instance.

Return Value

Returns true if yes, else false.

ralf::vmm_ralf_rfile_instance::getArraySize()

Gets the size of the array.

Synopsis

```
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the size of the array.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Size of the array.

ralf::vmm_ralf_rfile_instance::getArrayIncr()

Gets the array increment.

Synopsis

```
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the array increment.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Array increment.

ralf::vmm_ralf_domain Class Reference

Abstract base class of RALF block and system domain descriptors.

Synopsis

```
namespace ralf {
   class vmm_ralf_domain {
      …
   };
};
```

Description

Abstract base class of RALF block and system domain descriptors.

Public Member Functions

```
virtual const size_t & getNBytes() const = 0;
virtual const std::string & getName() const = 0;
virtual endianness_e getEndian() const = 0;
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Summary

- ralf::vmm_ralf_domain::getNBytes() .....page D-106
- ralf::vmm_ralf_domain::getName() .....page D-107
- ralf::vmm_ralf_domain::getEndian() .....page D-108
- ralf::vmm_ralf_domain::getConstraint() .....page D-109

ralf::vmm_ralf_domain::getNBytes()

Gets the domain width in bytes.

Synopsis

```
virtual const size_t & getNBytes() const = 0;
```

Description

Gets the domain width in bytes.

Return Value

Domain width in bytes.

ralf::vmm_ralf_domain::getName()

Gets the name of the domain.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the domain.

If the domain is unnamed, an empty string will be returned.

Return Value

Reference to string constant (storing the name of the domain).

ralf::vmm_ralf_domain::getEndian()

Gets the endian-ness.

Synopsis

```
enum endianness_e {
      RALF_NO_ENDIAN,
      RALF_LITTLE_ENDIAN,
      RALF_BIG_ENDIAN,
      RALF_LITTLE_FIFO,
      RALF_BIG_FIFO
};
virtual endianness_e getEndian() const = 0;
```

Description

Gets the endian-ness, if any specified, in the corresponding RALF description.

By default, RALF_NO_ENDIAN will be returned.

Return Value

Enumeration constant of type `endianness_e` to indicate the domain’s "endian" specification.

ralf::vmm_ralf_domain::getConstraint()

Gets the constraint.

Synopsis

```
virtual const vmm_ralf_constraint * getConstraint() const=0;
```

Description

Gets the constraint, if any specified, in the corresponding RALF description.

Return Value

Pointer to the const constraint data structure, if the constraint has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_blk_domain::const_component_iter Class Reference

RALF block component iterator.

Synopsis

```
namespace ralf {
  class vmm_ralf_blk_domain : public virtual vmm_ralf_domain
  {
    class const_component_iter {
      …
    };
  };
};
```

Description

Iterator to iterate over all components (that is, registers, memories and regfiles) in a RALF block domain.

Public Member Functions

```
virtual const_component_iter & operator++() = 0;
virtual const vmm_ralf_blk_component_instance * operator*()
const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf:: vmm_ralf_blk_domain::const_component_iter::operator++()page D-111
- ralf:: vmm_ralf_blk_domain::const_component_iter::operator*()page D-112
- ralf::vmm_ralf_blk_domain::const_component_iter::isBegin()page D-113
- ralf::vmm_ralf_blk_domain::const_component_iter::isEnd() .page D-114

ralf::vmm_ralf_blk_domain::const_component_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_component_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next block component instance. If the iterator is already pointing to the last block component instance, then invocation of this function would result in this iterator pointing to one past the last block component instance, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf:: vmm_ralf_blk_domain::const_component_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_blk_component_instance * operator*()
const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the block component instance descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to constant block component instance descriptor, currently getting pointed to, by the iterator.

ralf::vmm_ralf_blk_domain::const_component_iter::isBegin()

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Return Value

Returns true if the iterator is pointing to the first element. Else returns false.

ralf::vmm_ralf_blk_domain::const_component_iter::isEnd()

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_blk_domain Class Reference

RALF block domain.

Synopsis

```
namespace ralf {
  class vmm_ralf_blk_domain : public virtual vmm_ralf_domain
  {
    …
  };
};
```

Description

RALF descriptor of a block domain.

Public Member Functions

```
virtual const_component_iter * createComponentIter()
const=0;
```

Summary

- ralf::vmm_ralf_blk_domain::createComponentIter() .....page D-116

ralf::vmm_ralf_blk_domain::createComponentIter()

Creates an iterator to iterate over the list of all block component instances.

Synopsis

```
virtual const_component_iter * createComponentIter()
const=0;
```

Description

Creates an iterator to iterate over the list of all block component instances in the block domain.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first block component instance.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_blk_or_sys_typ Class Reference

Abstract base class for RALF descriptors of `block` and `system` types.

Synopsis

```
namespace ralf {
   class vmm_ralf_blk_or_sys_typ {
      …
   };
};
```

Description

Abstract base class for RALF descriptors of `block` and `system` types.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const std::string & getScopeName() const = 0;
```

Summary

- ralf::vmm_ralf_blk_or_sys_typ::getName() .....page D-118
- ralf::vmm_ralf_blk_or_sys_typ::getScopeName() .....page D-119

ralf::vmm_ralf_blk_or_sys_typ::getName()

Gets the name of the RALF `block` or `system` type.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the name of the RALF `block` or `system` type.

Return Value

Reference to string constant (storing name of the RALF `block` or `system` type).

ralf::vmm_ralf_blk_or_sys_typ::getScopeName()

Gets the scope name of the RALF `block` or `system` type.

Synopsis

```
virtual const std::string & getScopeName() const = 0;
```

Description

Gets the scope name of the RALF `block` or `system` type.

Return Value

Reference to string constant (storing scope name of the RALF `block` or `system` type).

ralf::vmm_ralf_blk_typ::const_domain_iter Class Reference

RALF block domain iterator.

Synopsis

```
namespace ralf {
   class vmm_ralf_blk_typ : public virtual
   vmm_ralf_blk_or_sys_typ {
      class const_domain_iter {
         …
      };
   };
};
```

Description

Iterator for iterating over all domains of a RALF block.

Public Member Functions

```
virtual const_domain_iter & operator++() = 0;
virtual const vmm_ralf_blk_domain * operator*() const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf::vmm_ralf_blk_typ::const_domain_iter::operator++() ..page D-121
- ralf::vmm_ralf_blk_typ::const_domain_iter::operator*() ...page D-122
- ralf::vmm_ralf_blk_typ::const_domain_iter::isBegin() .....page D-123
- ralf::vmm_ralf_blk_typ::const_domain_iter::isEnd() .....page D-124

ralf::vmm_ralf_blk_typ::const_domain_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_domain_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next block domain. If the iterator is already pointing to the last block domain, then invocation of this function would result in this iterator pointing to one past the last block domain, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf::vmm_ralf_blk_typ::const_domain_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_blk_domain * operator*() const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the RALF block domain descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to const RALF block domain descriptor, currently getting pointed to, by the iterator.

ralf::vmm_ralf_blk_typ::const_domain_iter::isBegin()

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Return Value

Returns true if the iterator is pointing to the first element. Else returns false.

ralf::vmm_ralf_blk_typ::const_domain_iter::isEnd()

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_blk_typ Class Reference

RALF `block` type.

Synopsis

```
namespace ralf {
   class vmm_ralf_blk_typ : public virtual
vmm_ralf_blk_or_sys_typ {
      …
   };
};
```

Description

RALF descriptor of a `block` type.

Public Member Functions

```
virtual const_domain_iter * createDomainIter() const = 0;
```

Summary

- ralf::vmm_ralf_blk_typ::createDomainIter() .....page D-126

ralf::vmm_ralf_blk_typ::createDomainIter()

Creates an iterator to iterate over the list of all domains.

Synopsis

```
virtual const_domain_iter * createDomainIter() const = 0;
```

Description

Creates an iterator to iterate over the list of all domains in the block.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first block domain.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_domain_instance Class Reference

Abstract base class descriptor of a block or system domain instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_domain_instance {
      …
   };
};
```

Description

Abstract base class of a block or system domain instance, as instantiated in the parent system.

Public Member Functions

```
virtual const std::string & getName() const = 0;
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
virtual const std::string & getInstantiatedDomainName()
const = 0;
virtual bool isArray() const = 0;
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual const size_t & getOffset() const
throw(vmm_ralf_invalid_oprn) = 0;
virtual const size_t & getAddrSpaceSize() const = 0;
```

Summary

- ralf::vmm_ralf_domain_instance::getName() .....page D-128
- ralf::vmm_ralf_domain_instance::getHdlPath() .....page D-129
- ralf:: vmm_ralf_domain_instance::getInstantiatedDomainName()page D-130
- ralf::vmm_ralf_domain_instance::isArray() .....page D-131
- ralf::vmm_ralf_domain_instance::getArraySize() .....page D-132
- ralf::vmm_ralf_domain_instance::getArrayIncr() .....page D-133
- ralf::vmm_ralf_domain_instance::getOffset() .....page D-134
- ralf::vmm_ralf_domain_instance::getAddrSpaceSize() .....page D-135

ralf::vmm_ralf_domain_instance::getName()

Gets the instance name.

Synopsis

```
virtual const std::string & getName() const = 0;
```

Description

Gets the instance name of the instantiated block or system domain.

Return Value

Reference to string constant (storing the instance name).

ralf::vmm_ralf_domain_instance::getHdlPath()

Gets the HDL path.

Synopsis

```
virtual const vmm_ralf_hdl_path * getHdlPath() const = 0;
```

Description

Gets the HDL path, if any specified, in the corresponding RALF description.

Return Value

Pointer to the const HDL path data structure, if the HDL path has been specified in the corresponding RALF description. Else returns NULL.

ralf::vmm_ralf_domain_instance::getInstantiatedDomainName()

Gets the name of the instantiated domain.

Synopsis

```
virtual const std::string & getInstantiatedDomainName()
const = 0;
```

Description

Gets the name of the instantiated domain.

Return Value

Reference to string constant (storing the name of the instantiated domain).

ralf::vmm_ralf_domain_instance::isArray()

Checks if the instance is an array instance.

Synopsis

```
virtual bool isArray() const = 0;
```

Description

Checks if the instance is an array instance.

Return Value

Returns 'true' if yes, else 'false'.

ralf::vmm_ralf_domain_instance::getArraySize()

Gets the size of the array.

Synopsis

```
virtual const size_t & getArraySize() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the size of the array

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Size of the array.

ralf::vmm_ralf_domain_instance::getArrayIncr()

Gets the array increment.

Synopsis

```
virtual const size_t & getArrayIncr() const
throw(vmm_ralf_invalid_oprn) = 0;
```

Description

Gets the array increment.

Throws exception `vmm_ralf_invalid_oprn`, if this method is invoked on a non-array instance.

Return Value

Array increment value.

ralf::vmm_ralf_domain_instance::getOffset()

Gets the offset.

Synopsis

```
virtual const size_t & getOffset() const = 0;
```

Description

Gets the offset/address of the domain instance in the parent system domain.

Return Value

Offset/address of the domain instance.

ralf::vmm_ralf_domain_instance::getAddrSpaceSize()

Gets the address space size.

Synopsis

```
virtual const size_t & getAddrSpaceSize() const = 0;
```

Description

Gets the address space size of the domain instance in the parent system domain.

Return Value

Address space size.

ralf::vmm_ralf_blk_domain_instance Class Reference

RALF descriptor of a block domain instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_blk_domain_instance
   : public virtual vmm_ralf_domain_instance {
      …
   };
};
```

Description

RALF descriptor of a block domain instance, as instantiated in the parent system domain.

Public Member Functions

```
virtual const vmm_ralf_blk_typ & getType() const = 0;
```

Summary

- ralf::vmm_ralf_blk_domain_instance::getType() .....page D-137

ralf::vmm_ralf_blk_domain_instance::getType()

Gets the corresponding RALF `block` type.

Synopsis

```
virtual const vmm_ralf_blk_typ & getType() const = 0;
```

Description

Gets the corresponding RALF `block` type.

Return Value

Reference to const RALF `block` type descriptor.

ralf::vmm_ralf_sys_domain::const_component_iter Class Reference

RALF system component iterator.

Synopsis

```
namespace ralf {
  class vmm_ralf_sys_domain : public virtual vmm_ralf_domain
  {
    class const_component_iter {
      …
    };
  };
};
```

Description

Iterator to iterate over all components (that is, blocks and subsystems) of a RALF system domain.

Public Member Functions

```
virtual const_component_iter & operator++() = 0;
virtual const vmm_ralf_domain_instance * operator*() const
= 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf:: vmm_ralf_sys_domain::const_component_iter::operator++()page D-139
- ralf:: vmm_ralf_sys_domain::const_component_iter::operator*()page D-140
- ralf:: vmm_ralf_sys_domain::const_component_iter::isBegin()page D-141
- ralf::vmm_ralf_sys_domain::const_component_iter::isEnd() .page D-142

ralf:: vmm_ralf_sys_domain::const_component_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_component_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next block or subsystem domain instance. If the iterator is already pointing to the last block or subsystem domain instance, then invocation of this function would result in this iterator pointing to one past the last block ir subsystem domain instance, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf:: vmm_ralf_sys_domain::const_component_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_domain_instance * operator*()
const=0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the block or subsystem domain instance descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to const block or subsystem domain instance descriptor, currently getting pointed to, by the iterator.

ralf::vmm_ralf_sys_domain::const_component_iter::isBegin()

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Return Value

Returns true if the iterator is pointing to the first element. Else returns false.

ralf::vmm_ralf_sys_domain::const_component_iter::isEnd()

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_sys_domain Class Reference

RALF system domain.

Synopsis

```
namespace ralf {
  class vmm_ralf_sys_domain : public virtual vmm_ralf_domain
  {
    …
  };
};
```

Description

RALF descriptor of a system domain.

Public Member Functions

```
virtual const_component_iter * createComponentIter() const
= 0;
```

Summary

- ralf::vmm_ralf_sys_domain::createComponentIter() .....page D-144

ralf::vmm_ralf_sys_domain::createComponentIter()

Creates an iterator to iterate over the list of all block or subsystem domain instances.

Synopsis

```
virtual const_component_iter * createComponentIter()
const=0;
```

Description

Creates an iterator to iterate over the list of all block or subsystem domain instances in the system domain.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first block or subsystem domain instance.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_sys_typ::const_domain_iter Class Reference

RALF system domain iterator.

Synopsis

```
namespace ralf {
   class vmm_ralf_sys_typ :
         public virtual vmm_ralf_sys_or_sys_typ {
      class const_domain_iter {
        …
      };
   };
};
```

Description

Iterator for iterating over all domains of a RALF system.

Public Member Functions

```
virtual const_domain_iter & operator++() = 0;
virtual const vmm_ralf_sys_domain * operator*() const = 0;
virtual bool isBegin() const = 0;
virtual bool isEnd() const = 0;
```

Summary

- ralf::vmm_ralf_sys_typ::const_domain_iter::operator++() ..page D-146
- ralf::vmm_ralf_sys_typ::const_domain_iter::operator*() ...page D-147
- ralf::vmm_ralf_sys_typ::const_domain_iter::isBegin() .....page D-148
- ralf::vmm_ralf_sys_typ::const_domain_iter::isEnd() .....page D-149

ralf::vmm_ralf_sys_typ::const_domain_iter::operator++()

Prefix increment operator.

Synopsis

```
virtual const_domain_iter & operator++() = 0;
```

Description

Increments the iterator to point to the next system domain. If the iterator is already pointing to the last system domain, then invocation of this function would result in this iterator pointing to one past the last system domain, a state which can be detected by the `isEnd()` method.

Return Value

Reference to the incremented iterator itself.

ralf::vmm_ralf_sys_typ::const_domain_iter::operator*()

Overloaded de-reference operator.

Synopsis

```
virtual const vmm_ralf_sys_domain * operator*() const = 0;
```

Description

This overloaded de-reference operator will let the user de-reference the iterator to get the RALF system domain descriptor (getting pointed to by this iterator), just like the way de-referencing of a pointer is done to access the object it is pointing to.

Return Value

Pointer to const RALF system domain descriptor, currently getting pointed to, by the iterator.

ralf::vmm_ralf_sys_typ::const_domain_iter::isBegin()

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Synopsis

```
virtual bool isBegin() const = 0;
```

Description

Checks if the iterator is pointing to the first element, that is, the beginning of the sequence.

Return Value

Returns true if the iterator is pointing to the first element. Else returns false.

ralf::vmm_ralf_sys_typ::const_domain_iter::isEnd()

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Synopsis

```
virtual bool isEnd() const = 0;
```

Description

Checks if the iterator is pointing to the end (that is, one past the last element) of the sequence.

Return Value

Returns true if the iterator is pointing to the end. Else returns false.

ralf::vmm_ralf_sys_typ Class Reference

RALF system type.

Synopsis

```
namespace ralf {
   class vmm_ralf_sys_typ : public virtual
         vmm_ralf_sys_or_sys_typ {
      …
   };
};
```

Description

RALF descriptor of a system type.

Public Member Functions

```
virtual const_domain_iter * createDomainIter() const = 0;
```

Summary

- ralf::vmm_ralf_sys_typ::createDomainIter() .....page D-151

ralf::vmm_ralf_sys_typ::createDomainIter()

Creates an iterator to iterate over the list of all domains.

Synopsis

```
virtual const_domain_iter * createDomainIter() const = 0;
```

Description

Creates an iterator to iterate over the list of all domains in the system.

Return Value

Pointer to a newly created iterator object. The returned iterator is initialized to point to the first system domain.

The returned iterator object would need to be deleted by the user using `delete`, after using it.

ralf::vmm_ralf_sys_domain_instance Class Reference

RALF descriptor of a system domain instance.

Synopsis

```
namespace ralf {
   class vmm_ralf_sys_domain_instance
         : public virtual vmm_ralf_domain_instance {
      …
   };
};
```

Description

RALF descriptor of a system domain instance, as instantiated in the parent system domain.

Public Member Functions

```
virtual const vmm_ralf_sys_typ & getType() const = 0;
```

Summary

- ralf::vmm_ralf_sys_domain_instance::getType() .....page D-153

ralf::vmm_ralf_sys_domain_instance::getType()

Gets the corresponding RALF `system` type.

Synopsis

```
virtual const vmm_ralf_sys_typ & getType() const = 0;
```

Description

Gets the corresponding RALF `system` type.

Return Value

Reference to const RALF `system` type descriptor.

ralf::vmm_ralf Class Reference

Top level concrete class whose object should be instantiated by the end user.

Synopsis

```
namespace ralf {
   class vmm_ralf {
      …
   };
};
```

Description

Top level concrete class whose object should be instantiated by the end user, for using RALF C++ API(s) or Synopsys' version of the RALF to RTL generator.

Public Member Functions

```
vmm_ralf(…);
virtual void generateRTL() const;
const vmm_ralf_blk_or_sys_typ * getTopLevelBlockOrSys()
const;
```

Summary

- ralf::vmm_ralf::vmm_ralf(…) .....page D-155
- ralf::vmm_ralf::generateRTL() .....page D-156
- ralf::vmm_ralf::getTopLevelBlockOrSys() .....page D-157

ralf::vmm_ralf::vmm_ralf(...)

Constructor.

Synopsis

```
vmm_ralf(const char * ralf_file,
         const char * top,
         const char * rtl_dir = "./",
         const char * inc_dirs = NULL,
         bool quiet_mode = false);
```

Description

Top level constructor end-users would need to use.

Parameters

```
ralf_file - Unix absolute/relative RALF file name.
top
     - Top level block/system's name that needs to be accessed
using the C++ API. Same as what is specified with the '-t'
option of ralgen.
rtl_dir
     - Dir where the RTL code should be generated. Defaults
to current directory.
inc_dirs
     - Include path for 'source'd RALF files. Same as what
can be specified using ralgen's '-I' option.
quiet_mode      - If set to 'true', all ralgen warnings will
be suppressed. Same as what is specified using ralgen's '-
q' option.
```

Return Value

N/A

ralf::vmm_ralf::generateRTL()

RALF to RTL generator.

Synopsis

```
virtual void generateRTL() const;
```

Description

Synopsys version of RALF to RTL code generator.

Return Value

void

ralf::vmm_ralf::getTopLevelBlockOrSys()

Gets the top level block or system data structure (as per `top` specified in the constructor).

Synopsis

```
const vmm_ralf_blk_or_sys_typ * getTopLevelBlockOrSys()
const;
```

Description

Gets the top level block or system data structure (as per `top` specified in the constructor).

Users would need to dynamic_cast the returned `const vmm_ralf_blk_or_sys_typ *` pointer to `const vmm_ralf_blk_typ *` or `const vmm_ralf_sys_typ *` to find out if the RALF top is a block or a system, and then, get access to the rest of the block or system data using that dynamic_cast-ed pointer.

Return Value

Pointer to const RALF block or system type descriptor.